P O L I T E C H N I K A Ł Ó D Z K A

SZYMON GRABOWSKI

NEW ALGORITHMS FOR EXACT AND APPROXIMATE TEXT MATCHING

Ł Ó D Ź 2009

Contents

Streszczenie **7**

Preface **8**

1 Online exact string matching **11**
1.1 Preliminaries . . . 11
1.2 Worst-case and average-case optimized algorithms . . . 12
1.3 Modern approaches: bit-parallel simulations of NFA . . . 18
1.4 Average-optimal Shift-Or algorithm . . . 22
1.4.1 Relaxing q . . . 25
1.4.2 Handling longer patterns . . . 25
1.4.3 Linear worst-case time . . . 26
1.4.4 Implementation . . . 26
1.5 The Aho–Corasick algorithm . . . 27
1.5.1 Optimal Aho–Corasick . . . 28
1.6 Application of the AOSO technique in other algorithms . . . 31
1.6.1 Pattern matching with swaps . . . 31
1.6.2 Pattern matching with all circular shifts . . . 32
1.6.3 (δ, γ)-matching . . . 33
1.7 Experimental results . . . 33
1.7.1 Shift-Or and Shift-Add experiments . . . 33
1.7.2 Aho–Corasick experiments . . . 36
1.8 Conclusions . . . 38

2 Online approximate string matching **41**
2.1 Similarity measures and their applications . . . 42
2.2 Basic techniques . . . 44
2.2.1 Dynamic programming algorithms . . . 45
2.2.2 Algorithms based on automata . . . 49
2.2.3 Fast Fourier transform based algorithms . . . 49
2.2.4 Algorithms based on bit-parallelism . . . 50
2.2.5 The filtering approach . . . 51

2.3 A new technique for bit-parallel algorithms with counters 52
2.3.1 Shift-Add algorithm 52
2.3.2 Counter-splitting 53
2.3.3 Expanding and contracting counters 55
2.3.4 Matryoshka counters 57
2.4 Average-Optimal Shift-Add for short patterns 58
2.5 Other applications of Matryoshka counters 60
2.5.1 (δ, γ)-matching and (δ, k)-matching 60
2.5.2 Intrusion detection and episode matching 62
2.6 Global similarity measures 65
2.6.1 The LCS problem 65
2.6.2 LCS-related problem variants 67
2.6.3 The LCTS problem, theoretical and practical solutions . . . 69
2.7 Conclusions 76

3 Matching with gaps 77
3.1 Preliminaries 79
3.2 Previous work 80
3.3 Dynamic programming 81
3.4 Row-wise sparse dynamic programming 86
3.4.1 Efficient worst case 86
3.4.2 Efficient average case 87
3.4.3 Faster preprocessing 87
3.4.4 Improved algorithm for large α 90
3.5 Column-wise sparse dynamic programming 92
3.6 Simple algorithm for (δ, α)-matching 93
3.6.1 Sublinear average case 94
3.7 Simple algorithm for (δ, γ, α)-matching 96
3.7.1 Improving the worst case 97
3.8 Bit-parallel dynamic programming for (δ, α)-matching 98
3.8.1 Fast algorithm on average 100
3.8.2 Handling large α in $O(1)$ time 103
3.8.3 Relaxing δ and α 104
3.9 Bit-parallel dynamic programming for (δ, γ, α)-matching 105
3.9.1 Cut-off 108
3.9.2 Lazy preprocessing 109
3.9.3 Improving the worst case for large α 111
3.9.4 Multiple patterns 112
3.9.5 Filtering 113
3.10 Non-deterministic finite automata for (δ, α)-matching 113
3.11 Other models 116
3.11.1 Handling character classes 116
3.11.2 Matching with general gaps 117
3.11.3 Matching with k mismatches 118
3.11.4 $(\delta, k_\Delta, \alpha)$-matching 118

3.12 Transposition invariance . . . 119
3.12.1 Transposition invariant Simple . . . 119
3.12.2 Transposition invariant DP . . . 122
3.13 Experimental results for (δ, α)-matching and related problems . . . 124
3.13.1 Transposition invariance . . . 126
3.13.2 PROSITE patterns . . . 127
3.14 Experimental results for (δ, γ, α)-matching . . . 128

4 Searching in compressed domain 130
4.1 Motivation and brief overview of the area . . . 131
4.2 Search-supporting codes for large alphabets . . . 135
4.3 q-gram based full-text coding with efficient search capabilities . . . 142
4.3.1 Searching for long patterns . . . 143
4.3.2 Searching for short patterns . . . 145
4.3.3 Experimental results . . . 147
4.4 A simple technique for denser encoding of static texts . . . 150
4.4.1 Experimental results . . . 151
4.5 Conclusions and future work . . . 152

5 Compressed full-text indexes 155
5.1 Motivation and problem aspects . . . 155
5.2 Classic indexing data structures . . . 158
5.3 Early compact text indexes . . . 161
5.4 Basic concepts of the FM-index . . . 163
5.4.1 Burrows–Wheeler transform . . . 164
5.4.2 Search mechanism (LF-mapping) . . . 167
5.4.3 Locating occurrences and displaying the text . . . 168
5.5 *Rank* and *select* in theory . . . 170
5.5.1 Constant-time *rank* . . . 170
5.5.2 Constant-time *select* . . . 171
5.5.3 *Rank* and *select* for compressed sequences . . . 173
5.6 *Rank* and *select* in practice . . . 173
5.6.1 *Rank* queries via popcounting . . . 173
5.6.2 *Rank* queries using a single level plus sequential scan . . . 175
5.6.3 *Select* queries . . . 176
5.6.4 $SelectNext$ queries . . . 179
5.7 The wavelet tree . . . 180
5.8 FM-Huffman and its variants . . . 182
5.8.1 Basic idea . . . 183
5.8.2 Locate and display . . . 186
5.8.3 K-ary Huffman . . . 190
5.8.4 Kautz–Zeckendorf coding . . . 192
5.8.5 Other space-time tradeoffs . . . 192
5.9 Experimental results . . . 195
5.9.1 Space results . . . 196

5.9.2 Counting queries 197
5.9.3 Reporting queries 198
5.9.4 Displaying text 198
5.9.5 Analysis of results 200
5.10 Recent advancements in compressed indexes 202

6 Conclusions **205**

Bibliography **208**

List of Symbols and Abbreviations **236**

List of Figures **238**

List of Tables **240**

Summary **241**

Charakterystyka zawodowa autora **243**

Streszczenie

W rozprawie skupiono się na wybranych problemach wyszukiwania dokładnego i przybliżonego w tekście. Pojęcie tekstu winno być rozumiane szeroko, obejmując dane w językach naturalnych, sekwencje bioinformatyczne oraz zapisy utworów muzycznych (nutowe).

Praca składa się z pięciu rozdziałów, z których każdy poświęcony jest osobnemu zagadnieniu. W kolejności, są to: wyszukiwanie dokładne, wyszukiwanie przybliżone, wyszukiwanie sekwencji z przerwami (ang. *gaps*), wyszukiwanie *online* w tekście skompresowanym oraz pełnotekstowe indeksy skompresowane. Rozprawa wnosi wkład w rozwój każdego z tych problemów. Każdy rozdział zaczyna się jednak od przedstawienia odnośnego stanu wiedzy.

Wiele z zaproponowanych w pracy algorytmów wykorzystuje równoległość bitową, nowoczesną technikę obliczeń z wykorzystaniem poszczególnych bitów rejestru procesora. W szczególności, zaprezentowano dwie nowe techniki równoległości bitowej, jedną mającą na celu optymalizację przypadku średniego w wyszukiwaniu dokładnym, drugą redukującą złożoność w przypadku najgorszym w algorytmach wykorzystujących liczniki. Te dość ogólne techniki algorytmiczne zostały pomyślnie zastosowane w szeregu konkretnych znanych algorytmów wyszukiwania.

W pracy pokazano, iż do wyszukiwania sekwencji z przerwami można stosować rozliczne podejścia, rozszerzając znacznie istniejący arsenał metod dla problemów tej kategorii. Nowe wyniki bazują m. in. na technikach algorytmicznych równoległości bitowej, programowania dynamicznego rzadkiego i oszczędnych bitowo-równoległych symulacjach automatów NFA.

Przedstawiono również algorytmy wyszukiwania w danych skompresowanych, cechujące się zarówno wydajnością, jak i prostotą.

Obok analiz teoretycznych, większość algorytmów zaimplementowano i poddano testom empirycznym, a osiągnięte wyniki zwykle pozwalają zaliczyć nowe metody do najefektywniejszych dla danych problemów.

Preface

Searching is arguably the most important of all problems that computer science deals with. Very broadly speaking, this problem consists in *reporting* occurrences of an object (key) in a database. Reporting may mean in a particular scenario: yielding the number of its occurrences, returning their exact locations, or merely answering 'yes' if at least one match is found, and 'no' in the opposite case.

The term *database* used here is also very general; it can be a collection of records, a list of integers, a catalogue of web pages, a sequence of DNA, etc.

Those very different kinds of objects stored in a database imply equally diverse kinds of sought keys.

The seemingly very obvious notion of matching may be relaxed in many ways, leading to many approximate matching models. Similarly, the keys themselves may be of "generalized" form, e.g., can be described with regular expressions.

Finally, there may be different scenarios under which the search processes will be run. The database may be static, hence it can be preprocessed to make future searches faster, or it may be dynamic, which raises the fundamental question of efficient updates. The searches may be run on a single one-core one-CPU machine, or they may be run on a parallel platform, possibly in a distributed environment. The database may be so large that it cannot be held entirely in main memory, hence the minimization of I/O operations is a crucial task. Additionally, current theoretical models (not to say about experimental works!) more and more often notice the fact that computer memory nowadays is hierarchical rather than a flat (i.e., uniform-access) von Neumann structure.

Among the search problems, a broad and important category seems to be *string matching* problems, where the pattern and the database are *textual*. Text, since the beginning of known history of mankind, till our digital era,

remains one of the most important media of information. The previous century even broadened the definition of text, which nowadays means not only natural language (NL) messages but also bioinformatics data (e.g., DNA and protein sequences), music scores, program sources and structured data, e.g., XML. Moreover, textual algorithms have some importance for searching in and indexing images.

This thesis focuses on various exact and approximate matching problems for textual data. The author's main contributions (usually joint efforts) to the field of string matching can be summarized as follows.

- We proposed a "striding" bit-parallel technique for the problem of exact string matching, and also applications of this technique for several other matching problems and known algorithms (e.g., Aho–Corasick for multiple matching). Our algorithm for exact matching has optimal average time complexity, as long as the pattern is short enough. Experiments confirm that the devised algorithms based on our technique belong to the fastest known for the respective problems.
- We found a counter-splitting technique for the well-known Shift-Add algorithm for matching under Hamming distance, which improves the time complexity of Shift-Add from $O(n\lceil m \log(k)/w \rceil)$ to $O(n\lceil m/w \rceil)$. Again, this technique is very flexible, and we applied it successfully for several other problems.
- We proposed a number of algorithms for (δ, α) and (δ, γ, α)-matching problems, motivated by music information retrieval (MIR) applications. Our algorithms are competitive either in theory, or in practice, or both. For those problems, complex interplays between the preprocessing times and search times, and also average time and worst-case time complexities, have been spotted and analyzed. Our algorithms are based on bit-parallelism, sparse dynamic programming with cut-off, and automata.
- We showed that byte codes used earlier for efficient text compression on words, with support for direct pattern search, can also be successfully applied for non-segmented text. This has natural applications for non-NL texts, like DNA, proteins, music scores (MIDI), but also oriental languages (Chinese, Japanese, Korean), or even some European languages (especially agglutinative ones, like Finnish).
- We presented an extremely simple compressed index based on Huffman coding and Burrows–Wheeler transform. This index belongs to the FM-index family.

The dissertation comprises five chapters, dedicated to various string matching problems. Each chapter contains our own results and almost all introduced algorithms are verified experimentally, on real-world and/or synthetical data.

Chapter 1 presents the most classic problem of online exact string matching.

Chapter 2 is dedicated to approximate string matching, with most new results oriented towards matching under Hamming distance or the k-mismatches problem.

Chapter 3, entitled *Matching with gaps*, deals specifically with a broad class of approximate string matching problems, in which the pattern symbols are allowed to match a wider text area than the pattern length, and some of the symbols inside are ignored. Many matching problems with gaps are there discussed and our own results are presented in detail.

Chapter 4 presents a relatively new paradigm of online matching, which is matching directly in compressed text.

The last chapter covers the area of compressed full-text indexes.

This thesis could not achieve its shape without help of many people. I am grateful to my collaborators. The most notable is Kimmo Fredriksson. Thank you, Kimmo, not only for your ideas and hard work, but also for your immense patience to my incompetence, laziness and frequent blunders. Kimmo Fredriksson is also the author of most implementations of the algorithms presented in the thesis. I also thank Gonzalo Navarro and Sebastian Deorowicz; your professional approach taught me a lot, guys. I am grateful to Sebastian also for helping me with typesetting this book in LaTeX. The implementations in Chapter 5 are due to Alejandro Salinger and Rafał Przywarski. I thank my boss at Computer Engineering Department (KIS), Prof. Dominik Sankowski, for constant encouragement and support. Last but not the least, I thank my wife Barbara and the rest of the family, for the many things the so-called natural language seems helpless to express.

Chapter 1

Online exact string matching

The exact string matching is the oldest and most straightforward problem in the field, with tens of papers published in the last 30 years, presenting new algorithms (often only minor modifications of older ones) or, sometimes, new analyses and insights. Although the problem seems to be (almost) closed from the point of theory now, it is absolutely essential to any researcher in the field to know the basic techniques for exact matching, since they appear in solutions for many more complex problems (e.g., multiple string matching, approximate string matching, phrase retrieval with an inverted index).

Our contribution (joint work with Kimmo Fredriksson) for this problem is the design and empirical evaluation of a novel technique of sampling the text, which gives rise to the fast average-optimal Shift-Or (FAOSO) algorithm [FG05b, FG09a], being a non-trivial modification of the well-known Shift-Or algorithm [BYG89]. FAOSO is optimal in time for the average case, can be modified to yield linear (i.e., optimal) behavior in the worst case (both results require short enough patterns), and achieves very competitive search speed in practice. We show that the novel filtering technique used in this algorithm can also be applied to some other exact and approximate string matching problems [FG09a], the latter of which will be discussed in Chap. 2. In particular, we present the average-optimal Aho–Corasick (AOAC) algorithm, which does not suffer from the limitation on the pattern length in its average-case analysis.

1.1 Preliminaries

The main actors in this study are strings.

A string S is a sequence of symbols (characters) over a known alphabet

Σ. Let the length of S be m; the string S can be written as $s_0 s_1 \ldots s_{m-1}$ or $S[0 \ldots m-1]$ (both notations are equivalent). If not stated otherwise, we assume that the alphabet is integer, contiguous and finite, i.e., $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. The length of S is denoted by $|S|$. The notion of an empty string is sometimes useful, and such string will be denoted with ε. Obviously, $|\varepsilon| = 0$.

The string S_1 of length m *matches* the string S_2 iff $|S_2| = m$ and $S_1[i] = S_2[i]$ for all $0 \leq i < m$.

The concatenation of strings $S_1[0 \ldots m_1 - 1]$ and $S_2[0 \ldots m_2 - 1]$, denoted by $S_1 S_2$, is such a string S of length $m_1 + m_2$ that $S[0 \ldots m_1 - 1]$ matches S_1 and $S[m_1 \ldots m_1 + m_2 - 1]$ matches S_2.

The string $S[0 \ldots j-1]$, for any $0 < j \leq m$, is called a (non-empty) *prefix* of S. Similarly, $S[j \ldots m-1]$, for any $0 \leq j \leq m-1$, is called a (non-empty) *suffix* of S. Finally, $S[i \ldots j]$, $0 \leq i \leq j < m$, is called a *factor* of S. Trivially, any prefix or suffix of S is also a factor of S.

The concatenation of m instances of the same alphabet symbol, c, will be in short written as c^m.

The *exact string matching* problem can be stated in the following way. Given a pattern $P[0 \ldots m-1]$ and text $T[0 \ldots n-1]$, both made up of symbols from an integer alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$, report all the occurrences of P in T, i.e. return the sequence of indexes $j_1 < j_2 < \ldots < j_k$ such that $P[0 \ldots m-1]$ matches $T[j_l \ldots j_l + m - 1]$ for any $l = 1 \ldots k$, or return a negative answer if no match of P has been found.

If not stated otherwise, in average-case analyses we assume that both the text and the pattern are obtained under the same, uniformly random, distribution, and are independent. The probability that a given character of the text or the pattern is c_i, is $1/\sigma$, for any $i = 0 \ldots \sigma - 1$.

The logarithms used throughout this work are usually in base 2 and this base is then omitted. In some cases, however, to stress that the base is important, we do write it explicitly.

1.2 Worst-case and average-case optimized algorithms

The naïve (brute-force) algorithm for exact matching is to match P against each alignment of T, e.g. from left to right, aborting character comparisons as soon as possible. This clearly gives $O(nm)$ worst-case time complexity, but is of linear time on average. To see this, note that the probability that an arbitrary pair of characters matches is $1/\sigma$, which is less or equal $1/2$, and the probability that a prefix of P of length k matches a given text area

is $1/\sigma^k$, hence the expected matching prefix length is constant (and the constant does not exceed 2).

The algorithm's performance gets catastrophic if the pattern is highly periodic (or more precisely, if many prefixes of P are equal to some other factors of P) and the text contains many partial matches, but on typical texts, e.g., natural language ones, it fares quite well.

The danger of falling into a quadratic pathological case is reduced if the naïve algorithm is replaced with the Karp–Rabin (KR) algorithm [KR87]. KR is a classic exact string matching technique based on signatures (although the first application of this old idea in string matching was probably from Harrison in [Har71], where he only checked for a pattern occurrence, not its location, and the scheme required the text to be preprocessed). The key idea of the algorithm is to calculate a signature (hash) based on all the pattern characters, which is an integer stored in a single machine word. Then, using the same hash function, signatures are calculated over sliding text, in (overlapping) slices of length m, and the pattern signature is matched against the text slice signature in $O(1)$ time. Matches have to be verified (usually with a brute-force algorithm) but obviously no match will be missed. An important trait of the algorithm is that the hash function can be calculated incrementally, paying $O(1)$ time per text character. The worst-case time complexity of this algorithm is still $O(nm)$ (and, unfortunately, it is very easy to give examples which imply such behavior, e.g., $P = c^m$, $T = c^n$) but on average it is $O(n)$, with a low constant in the number of examined characters. Still, even in the best case the KR algorithm needs $O(n)$ time. A number of much more practical variations on the theme of Karp–Rabin were recently proposed by Lecroq [Lec07]. He actually adapted the classic Wu–Manber multiple string matching algorithm [WM94] to the case of a single pattern. The basic idea is to calculate hash values over all the possible q-grams, i.e. strings of length q, and use them to shift the pattern appropriately (or test for an occurrence if no shift is implied). The average time behavior is sublinear in n, but the worst case remains quadratic. Still, the family of algorithms has been designed for practical rather than theoretical purposes and the results reported in [Lec07, FL08] are competitive, especially for small alphabets.

In 1977 Rivest showed [Riv77] that any exact string matching algorithm must examine at least $n-m+1$ characters from the text, which constitutes the $\Omega(n)$ worst-case lower bound. A natural question arises then: is this lower bound tight?

The first major algorithm solving (positively) this problem was published by 1977 by Knuth, Morris and Pratt [KMP77], and is often denoted in the

literature with the initials of its authors, KMP.[1] The history of this discovery dates back as early as 1970 and is interesting in itself; the curious reader is referred to [Ste92, p. 8].

The underlying idea of KMP is to avoid backtracking in the text string in the event of a mismatch. The text is scanned from left to right, and at the first mismatching character, an $O(1)$-time lookup is performed into a precomputed *next* table, which tells how far to shift the pattern before the next character comparison. It is shown that precomputing the *next* table can be done in $O(m)$ time, independently of the alphabet size. KMP algorithm works in $O(n)$ worst-case time (more precisely, performs not more than $2n - 1$ character comparisons), hence being a proof that the Rivest's lower bound is tight. The main disappointment with KMP is however that it never compares less than n characters. Ironically, according to [BYN04], on most real-world texts (e.g., English, proteins) it is about twice slower than the naïve algorithm.

Interestingly, another – practical, this time – breakthrough result was published in the same year, 1977. The algorithm of Boyer and Moore (BM) [BM77] was the first one allowing to skip large portions of text during the search. In practice, for not very small alphabets and at least middle-sized patterns, this algorithm is several times faster than e.g. the naïve (or KMP) algorithm. Roughly speaking, if the alphabet size is not less than the pattern length, the BM algorithm reaches $O(n/m)$ time complexity on average. Its worst case is superlinear in n, a theoretical deficiency that was overcome in [AG86]. A novel idea in BM was to match the pattern against a piece of text from right to left. The algorithm is based on two heuristics which make use of knowledge obtained in the pattern preprocessing.

The Boyer–Moore algorithm is the first and one of the most prominent examples of the search technique called *filtering*. The idea of filtering is to slide the pattern quickly over large areas of text using some heuristic (or several heuristics) to discard them as potential matches. The heuristic should be fast but its answers are only negative, i.e. that a given piece of text is not a match. If the heuristic cannot give such an answer, it means that the given text area must be verified, i.e. the pattern must be matched against it using some other algorithm, often a brute-force one. Algorithms based on filtering are often very practical, and used for many search problems. The commonsense requirement for them is to have a both fast and selective heuristic, i.e. such that rejects a large enough fraction of the text, on average.

[1] As a historical remark, we note that essentially the same algorithm was obtained several years earlier by Gosper [BGS72], although no analysis was given.

Alg. 1 BMH(T, n, P, m).

```
1    for i ← 0 to σ − 1 do d[i] ← m
2    for i ← 0 to m − 2 do d[P[i]] ← m − 1 − i
3    i ← −1
4    while i + m < n do
5        while i + m < n and T[i + m] ≠ P[m − 1] do
6            i ← i + d[T[i + m]]
7        if P[0 . . . m − 2] = T[i + 1 . . . i + m − 1] then report match
8    i ← i + d[T[i + m]]
```

_	A	B	C	D	E
3	1	3	2	3	3

Array d
(*i*)

```
BAD_CAB        BAD_CAB        BAD_CAB
CAB               CAB             CAB
0123456        0123456        0123456
```

(*ii*) (*iii*) (*iv*)

Figure 1.1: BMH example

Still, filtering techniques are never winners in the worst case.

The BM algorithm is relatively complicated. In 1980 Horspool presented a simplified variant (BMH) [Hor80], which is based on a single heuristic. The auxiliary table is of the size of the alphabet, and a lookup for the text character aligned with the rightmost pattern character determines the shift (may be 0, and then the text area is inspected character-by-character). The BMH algorithm is $O(n/\min(m, \sigma))$ on average but $O(nm)$ in the worst case. Although BMH yields shorter shifts on average than the original BM, it is somewhat faster in practice, due to reduced complexity. Alg. 1 presents the pseudocode for the Boyer–Moore–Horspool (its variant from [BYN04]), and Fig. 1.2 illustrates its preprocessing and search on an example.

We will comment briefly the example. The pattern P is `CAB`, the text T is `BAD_CAB`. The integer alphabet corresponds to $\{$`_`, `A`, `B`, `C`, `D`, `E`$\}$. Figure (*i*) shows the preprocessing table d. On Fig. (*ii*) we see that the aligned text character (`D`) does not occur in P, hence the window will be shifted by $d[$`D`$] = |P| = 3$. Figure (*iii*) shows a mismatch again, the window will be shifted by $d[$`A`$] = 1$. Finally, on Fig. (*iv*), there is a match at the righmost location, the prefix of P is verified in the current text area, successfully, hence a pattern match is reported. P will be shifted by $d[$`B`$] = 3$, which reaches beyond the text and hence the algorithm terminates.

There are more than ten other algorithms from the Boyer–Moore family [Ste92, CL04] but the performance differences between them are slight and contradictory reports can be found in the literature. Perhaps the main

culprit of this confusion are different hardware platforms (especially CPU's) used in different experiments. Nowadays, it seems crucial to compare a new algorithm against the competitors on at least two different platforms, which is especially important for the exact string matching problem, since the processing speeds for this problem are high enough (often above 1 GB/s) to be vitally dependent on the hardware details.

Several original ideas applicable to the algorithms from the BM family deserve a presentation. One of them is shifting the pattern after a mismatch based not on $T[j]$ aligned with $P[m-1]$ (as in BMH), but on the next character, $T[j+1]$, as no match can be missed in this way [Sun90]. An extension of this idea was presented in [BR99]. Baeza-Yates [BY89b] suggested a careful ordering of pattern symbol comparisons, based on statistical knowledge about a given non-uniformly distributed text (e.g., English), to maximize the shifts or earlier detection of a mismatch, on average. In other words, rarer characters in the pattern (e.g., z or x in English) could be compared before more frequent ones (e.g., e or t). This approach was taken in the just cited work by Sunday [Sun90], in which he examined several variants. Along these lines, Smith developed an adaptive algorithm [Smi91] which works without any prior knowledge about the given text. The skip search algorithm [CLP98] samples the text at regular intervals of size m, and uses lists ("buckets" in the original terminology) of occurrences in the pattern for all the symbols of the alphabet, to brute-force check all the valid alignments. This is, in a way, a reversal of the common behavior of the BM algorithms: the skip search always samples the text in intervals of m characters (other algorithms do it only in the best case), but the verification is performed typically for several alignments of the pattern against a piece of the text (other algorithms do it only once per a shift). The average time complexity of this algorithm is like in BMH, that is, $O(n/\min(m,\sigma))$, and the worst case remains quadratic.

An interesting feature of the algorithms from the BM family is that they run faster with growing pattern length or alphabet size, and they benefit from longer shifts in those cases. The other side of this coin is also that e.g. the BMH algorithm is relatively weak for DNA strings, since its average shift is only approximately four characters (no matter how long the pattern is, provided that $m \geq 4$). A simple technique to mitigate this problem was considered yet in the works of Knuth, Morris and Pratt [KMP77], and Boyer and Moore [BM77]. It is possible to group characters into supercharacters (also called q-grams): pairs, or triples etc., effectively enlarging the alphabet. For example, using triples over DNA increases the effective alphabet size to $4^3 = 64$. The pattern gets shorter this way (but also the text gets respectively shorter, expressing its length in supercharacters), the

preprocessing costs grow, and the search algorithm requires some modifications, but on the overall this strategy pays, if the original alphabet is small [BY89b, KST94, TP97]. It is easy to notice that the search time of the BMH algorithm working on q-grams gets $O(\sigma^q + m + qn/\min(m, \sigma^q))$, which is minimized for $q = \log_\sigma m$ and finally we get the $O(m + n\log_\sigma(m)/m)$ complexity [BYN04], which is optimal on average [Yao79].

Interestingly, there are algorithms achieving $O(n)$ worst-case time using only $O(1)$ additional space. We say "additional", since the pattern itself must be kept in the operating memory, requiring $O(m)$ space. The first such algorithm, with the number of character comparisons bounded with $5n$, was given by Galil and Seiferas in 1983 [GS83]. Further results along these lines decreased the worst-case constant with the number of character comparisons [CP91, Bre93, GPR95]. Finally, Crochemore et al. [CGR99] presented an algorithm with sublinear average time (namely $O(n/\log m)$) with guaranteed linear-time worst case and working with constant additional space, independently of the alphabet used. They also posed a question, still open, what is the minimum amount of needed space for any exact string matching algorithm reaching the average-optimal time complexity, i.e., $O(m + n\log_\sigma(m)/m)$. All of those algorithms are quite complicated and of theoretical interest only. Therefore, researchers are more interested in developing algorithms with optimal average-case performance, or at least being close to optimal in this aspect, and running fast in practice. The space usage, even if superlinear in m, is rarely a hindrance, apart from very large patterns or very large alphabets. All the Boyer–Moore algorithms, presented above, belong to this class, still, the first algorithm reaching the average-case lower bound, $O(m + n\log_\sigma(m)/m)$, was Backward DAWG Matching (BDM) [CCG+94]. Some of its variants, TurboBDM and TurboRF, presented in the same work, are also worst-case optimal. Those algorithms build the suffix automaton of the reverse pattern P^r, that is, a deterministic finite automaton recognizing all suffixes of the reverse pattern, i.e. all prefixes of P. Unfortunately, BDM is also not very practical: for not very small alphabets and short or moderate-length patterns, it is easily outperformed by e.g. BMH [NR00, BYN04]. Another significant algorithm, a more practical alternative to BDM, was Backward Oracle Matching (BOM) [ACR99], which recognizes not only all factors of the reverse pattern, but also some other strings (an automaton with this property is called an oracle in string matching literature). It is not an obstacle, since the only string of length at least m recognized by the oracle is the reverse pattern itself. BOM is average-optimal but quadratic in the worst case in time, and with $O(m)$ preprocessing space and time. Recently, two new variants of BOM were pre-

sented [FL08] (one of them incorporates Sunday's idea of using the forward character for shifting [Sun90] that we mentioned above), which are not only faster than the original idea (sometimes about twice) but in many cases achieve best results among the tested algorithms, in the cited work.

It is interesting that the well-known lower bound on the number of character comparisons, $O(n \log_\sigma(m)/m)$, proved by Yao in 1979 [Yao79], can actually be "broken" in practice, which obviously can be achieved only for the price of changing the model of computation. It is possible to use super-characters, an idea that we presented a few paragraphs above, with a purpose to process $O(\log_\sigma m)$ original symbols in $O(1)$ time. The requirement however is that the input text is packed densely enough, ideally $\lceil \log_2 \sigma \rceil$ bits per character. If this input condition is met, then it is possible to achieve $O(n/m)$ time on average, and this was shown by Fredriksson [Fre02] whose algorithm is a superalphabet simulation of a suffix automaton. Using the same assumptions, very recently Bille [Bil09] showed a KMP variant where symbol packing led to $O(n/\log_\sigma n + m + occ)$ worst-case time, where occ is the number of matches.

1.3 Modern approaches: bit-parallel simulations of NFA

A classical approach to pattern search is using finite automata. The non-deterministic finite automaton (NFA) for recognizing a pattern P has only $|P| + 1$ states and very simple and regular structure. Formally speaking, the automaton recognizes language $\Sigma^* P$, i.e., all strings ending with P. Still, as this automaton is non-deterministic, each state of it can be either active (that is, corresponding to a matching pattern prefix), or inactive, and with each text character all the states have to be updated. Therefore, the algorithm's complexity is $O(nm)$, even in the average case.

A possibility to reduce the search time is to convert the NFA into a deterministic automaton (DFA). Deterministic automata are such that have only one active state at a time. Such a conversion can always be done, but the problem with DFA's is that they may need exponential space (and build time). In our example, building the DFA needs $O(m\sigma)$ both space and time, which is usually acceptable. The worst- and average-case time of this algorithm is thus $O(m\sigma + n)$. Actually, KMP algorithm is basically a smarter variant of such a DFA: what is improved in KMP is the space occupancy, decreased to $O(m)$. Another economical implementation of the DFA recognizing our pattern is Simon's algorithm [Sim93], which has very similar properties to KMP, only the maximum number of comparisons for a

single text character is slightly reduced (but both algorithms have the same total number of character comparisons in the worst case, namely $2n - 1$). Other improvements to KMP algorithm from the early 1990s reduced its total worst-case number of comparisons to $3/2n$ [Col91, AC91], $4/3n$ [GG92], and even $n+(n-m)8/(3(m+1))$ [CH92] (for online algorithms). Note that in the last result the constant in front of n in the number of comparisons gets arbitrarily close to 1, with growing pattern length. Nevertheless, the upper and lower bounds for this problem still do not match [CH97].

Let us get back to the non-deterministic automaton. It is slow because it needs to process many states at a time. A simple but profound idea to speed up this process drastically, was given in Baeza-Yates' PhD in 1989 [BY89a]. The idea, called later *bit-parallelism*, was based on a very simple observation that in a typical computer architecture the CPU works with registers which have many bits (usually 16 in those times, and 32 or 64 nowadays). If we can represent the states of the automaton with individual bits (storing the information if a given state is active or inactive), then the search time complexity can be divided by factor w, denoting the number of bits in a machine word (CPU register). Can we achieve it? The answer is positive. Soon later, Baeza-Yates and Gonnet [BYG89] presented an algorithm called Shift-Or, being a bit-parallel simulation of the NFA, and solving the string matching problem in $O(n\lceil m/w \rceil)$ worst- and average-case time.[2] In 1992, Wu and Manber [WM92b] presented a related algorithm, Shift-And, to solve the same problem, with the same complexity. Note that these algorithms work in linear time as long as the pattern is short enough (no speed penalty even in the worst case, as long as m is not greater than e.g. 32). Both algorithms are very similar in principle, only differ in internal representations of the state vectors and the update (transition) functions. Shift-And is somewhat slower than Shift-Or, thus we are not going to describe it.

Alg. 2 presents the pseudocode of Shift-Or. In the preprocessing, in lines 1–2, a table B, having one bit-mask entry for each alphabet symbol $c \in \Sigma$, is built. The bit-vectors of B obtain unset bits only at the positions of the occurrences of a given character in the pattern string. More formally, for $0 \le i \le m - 1$, the mask $B[c]$ has ith bit set to 0, iff $P[i] = c$. These bits correspond to the transitions of the implicit automaton. That is, if the bit i in $B[c]$ is 0, then there is a transition from the state i to the state $i + 1$ with character c.

Also, the state vector D is initialized. The ith bit of the state vector is

[2] It is not widely known that the idea of Baeza-Yates and Gonnet was a rediscovery of the algorithm by Dömölki, invented yet in 1964 [Döm64].

Alg. 2 Shift-Or(T, n, P, m).

1. **for** $i \leftarrow 0$ **to** $\sigma - 1$ **do** $B[i] \leftarrow \sim 0$
2. **for** $i \leftarrow 0$ **to** $m - 1$ **do** $B[P[i]] \leftarrow B[P[i]]$ **&** $\sim(1 << i)$
3. $D \leftarrow \sim 0$; $mm \leftarrow 1 << (m-1)$; $i \leftarrow 0$
4. **do**
5. $\quad D \leftarrow (D << 1) \mid B[T[i]]$
6. $\quad$ **if** $(D$ **&** $mm) \neq mm$ **then** report match
7. $\quad i \leftarrow i + 1$
8. **while** $i < n$

set to 0, iff the state i is active. Initially each bit is set to 1 (line 3 in Alg. 2). For each text symbol c the vector is updated by $D \leftarrow (D << 1) \mid B[c]$. This simulates all the possible transitions of the non-deterministic automaton in a single step. If after the update the mth bit of D is zero, then there is an occurrence of P.

The Shift-Or preprocessing needs $O(\sigma m)$ bits of space and $O(\sigma\lceil m/w \rceil + m)$ time. Typically, this is negligible compared to the text size. The search loop (lines 4–8) runs in $O(n)$ as long as $m = O(w)$.

Fig 1.3 serves as an illustration. The input data are: $P =$ TTTCATTC, $T =$ AGCTTTTCATTCTGAC, and $w = 8$ is assumed. (i) The preprocessing table B. Overall, it contains $m = 8$ zeros. (ii) Main phase of the algorithm. The columns, appearing from left to right, correspond to the state vector D. The zero in the lowest row denotes a match at position 11.

Since 1989, bit-parallelism has been applied for numerous string matching problems, often leading to very competitive practical algorithms. With-

```
B[A] = 11110111
B[C] = 11101110
B[G] = 11111111
B[T] = 00011001
```

(i)

```
  0123456789012345
  AGCTTTTCATTCTGAC
T 1110000110010111
T 1111000111011111
T 1111100111111111
C 1111111011111111
A 1111111101111111
T 1111111110111111
T 1111111111011111
C 1111111111101111
```

(ii)

Figure 1.2: Shift-Or example

out a doubt, this is one of the most important approaches to designing algorithms in the string matching field.

The elegance of Shift-Or lies in its flexibility. It can be easily adapted for some extended search problems, without affecting the essential search mechanism. For example, handling patterns with character classes (i.e., such that for any $i = 0 \ldots m-1$, $P[i]$ may represent a specified subset of Σ rather than a single symbol) is here trivial, as it only requires a modification in line 2. A related but different problem is multiple pattern matching. Solving this problem with Shift-Or consists in superimposing the patterns (let us assume for simplicity they are of equal length), i.e. storing at each position i the class of characters occurring at position i in at least one pattern from the given set. After doing that, the same procedure as just described (with modification in line 2 of Alg. 2), is run. The only difference now is that the reported positions may be false matches and thus require a verification. Another application of the Shift-Or machinery for multiple matching is possible if the patterns are short, and then the search states for several of them fit a single machine word, without superimposing [HFN05].

Several years ago, Fredriksson [Fre03] adapted Shift-Or for superalphabets, and presented an algorithm running in $O(nm/(qw) + occ)$ time, using $O(\sigma^q)$ space, where q is the number of original symbols in a supercharacter, and occ the number of matches. In his tests, the search performance for DNA and pattern lengths $m = 15$ was improved almost twice for $q = 4$, and even more than sixfold if as many as eight symbols were grouped (the output alphabet size was thus $4^8 = 65\,536$). The latter case, however, needed packing the symbols in text beforehand, so, for most existing DNA repositories, cannot be considered really online (and packing the symbols on the fly is likely to ruin all the speed improvement).

Shift-Or is, in a way, equivalent to KMP algorithm. The latter is a smart implementation of a plain DFA recognizing a given pattern, while the former is a smart (bit-parallel) implementation of an equivalent NFA. The same relation binds BDM and BNDM algorithms. BNDM (Backward Nondeterministic DAWG Matching) [NR00] uses the same BDM algorithm but its implementation consists in simulating the NFA in a bit-parallel manner instead of building the original DFA. BNDM is much simpler than BDM to implement, and also works faster in practice, apart from long patterns. For small alphabets and middle-sized patterns, BNDM belongs to the fastest algorithms. Its average search complexity is $O(\lceil m/w \rceil n \log_\sigma(m)/m)$, i.e. optimal as long as the pattern size is not much larger than the machine word size. Alg. 3 presents the pseudocode for SBNDM (Simplified BNDM) [Nav01b, PT03], a variant of BNDM which offers reduced shifts on average,

Alg. 3 Simplified-BNDM(T, n, P, m).

```
1   for i ← 0 to σ − 1 do B[i] ← 0
2   for i ← 0 to m − 1 do B[P[i]] ← B[P[i]] | (1 << (m − 1 − i))
3   /* let x be the length of the longest prefix of P which is also a suffix of P */
4   s0 ← m − x
5   pos ← 0
6   while pos ≤ n − m do
7       D ← ~0; j ← m
8       do
9           D ← (D << 1) & B[T[pos + j]]
10          j ← j − 1
11      until D = 0 or j = 0
12      if D ≠ 0 then
13          report match
14          pos ← pos + s0
15      else pos ← pos + j + 1
```

but also with a tighter main loop, which resulted in about 10% speedup over BNDM in the cited work. The preprocessing is performed in the first two lines. Note that the B bit-vectors are inverse of the analogous bit-vectors in Shift-Or algorithm. The internal *while* loop is run as long as the state vector D is non-zero. $D = 0$ means that the automaton is in a dead state, when a shift just beyond the non-occurring factor is performed (line 15).

Very recently, Durian et al. [DHPT09] presented an extensive evaluation of several string matching algorithms, in which the best results were usually obtained by q-gram based variations of BNDM and SBNDM, developed by the authors (cf. also [HD05]). This line of research combines the concept of bit-parallel simulation of NFA with using a superalphabet.

A weakness of BNDM is its quadratic behavior in the worst case, which contrasts with e.g. Shift-Or. A few solutions for this problem, i.e., BNDM variants with linear worst case (as long as $m = O(w)$), are known, e.g. LBNDM [HF04].

1.4 Average-optimal Shift-Or algorithm

In this section, we present a Shift-Or variant able to skip text characters, and we show that its average-case time complexity is optimal for short patterns. Then we also show a faster implementation which is called *fast average-optimal Shift-Or* (FAOSO). Experimental results show that our algorithm belongs to the fastest for a wide range of text kinds (DNA, English, proteins) and pattern lengths [FG05b, FG09a].

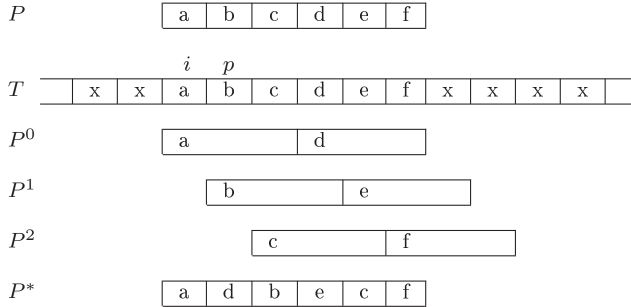

Figure 1.3: AOSO example

Our algorithm takes a parameter q, and from the original pattern we generate a set $\mathcal{P}$ of q new patterns $\mathcal{P} = \{P^0, \ldots, P^{q-1}\}$, each of length $m' = \lfloor m/q \rfloor$, as follows:

$$P^j[i] = P[j + iq], \quad j = 0 \ldots q-1, \quad i = 0 \ldots \lfloor m/q \rfloor - 1.$$

In other words, we generate q different alignments of the original pattern P, each alignment containing only every qth character. The total length of the patterns P^j is $q\lfloor m/q \rfloor \leq m$. For example, if $P =$ `abcdef` and $q = 3$, then $P^0 =$ `ad`, $P^1 =$ `be` and $P^2 =$ `cf`.

Assume now that P occurs at $T[i \ldots i + m - 1]$. From the definition of P^j it directly follows that

$$P^j[h] = T[i + j + hq], \quad j = i \bmod q, \quad h = 0 \ldots m' - 1.$$

This means that we can use the set $\mathcal{P}$ as a filter for the pattern P, and that the filter needs only to scan every qth character of T.

Fig. 1.3 illustrates. Assume that $P =$ `abcdef` occurs at text position $T[i \ldots i + m - 1]$, and that $q = 3$. The current text position is $p = 10$, and $T[p] =$ `b`. The next character the algorithm reads is $T[p + q] = T[13] =$ `e`. This triggers a match of $P^{p \bmod q} = P^1$, and the text area $T[p - 1 \ldots p - 1 + m - 1] = T[i \ldots i + m - 1]$ is verified.

The set of patterns can be searched simultaneously using the Shift-Or algorithm, as long as $qm' \leq w$. All the patterns are preprocessed together, as if they were concatenated. For our example pattern, $P =$ `abcdef`, we effectively preprocess a pattern $P' = P^0\ P^1\ P^2 =$ `adbecf`. Alg. 4 gives the code for preprocessing and filtering algorithms. If the pattern P^j matches, then the $((j + 1)m')$th bit in D is zero. This is detected with $(D \ \&\ mm) \neq mm$, where mm has every $((j + 1)m')$th bit set to 1. These bits have also

Alg. 4 Average-Optimal-Shift-Or(T, n, P, m, q).

```
1   for i ← 0 to σ − 1 do B[i] ← ∼0
2   h ← 0; mm ← 0
3   for j ← 0 to q − 1 do
4       for i ← 0 to ⌊m/q⌋ − 1 do
5           B[P[iq + j]] ← B[P[iq + j]] & ∼(1 << h)
6           h ← h + 1
7       mm ← mm | (1 << (h − 1))
8   D ← ∼0; i ← 0
9   do
10      D ← ((D & ∼mm) << 1) | B[T[i]]
11      if (D & mm) ≠ mm then Verify(T, i, n, P, m, q, D)
12      i ← i + q
13  while i < n
```

Alg. 5 Verify(T, i, n, P, m, q, D).

```
1   D ← (D & mm) ^ mm
2   while D ≠ 0 do
3       s ← ⌊log2(D)⌋
4       c ← −(⌊m/q⌋ − 1) q − ⌊s/⌊m/q⌋⌋
5       if P[0 . . . m − 1] = T[i + c . . . i + c + m − 1] then report match
6       D ← D & ∼(1 << s)
```

to be cleared in D before the shift operation $(D \mathbin{\&} {\sim}mm)$, to correctly initialize the first bit corresponding to each of the successive patterns.

Whenever an occurrence of P^j is found in the text, we must verify if P also occurs, with the corresponding alignment. To efficiently detect which patterns in $\mathcal{P}$ match, we first set $D \leftarrow (D \mathbin{\&} mm) \wedge mm$, i.e. the $((j+1)m')$th bit in D is now one if P^j matches, and all other bits are zero. Now $s \leftarrow \lfloor \log_2 D \rfloor$ gives the index of the highest bit set in D, and therefore j is $\lfloor s/m' \rfloor$, which is our alignment offset, see Fig. 1.3. The corresponding text position is then verified. Finally, we clear the bit s in D. This is repeated until D becomes zero, indicating that there are no more matches. Note that computing $\lfloor \log_2 x \rfloor$ can be done very efficiently in modern computers, e.g. by casting x to real number, and extracting the exponent from the standardized floating point representation. Still, in the RAM model of computation we may instead naïvely erase at most m' bits of D, in total time $O(m')$ over the verification loop, and the analysis below shows it does not hurt the overall complexity. Alg. 5 gives the verification code.

The filtering time of Alg. 4 is $O(n/q)$. The filter searches the exact matches of q patterns, each of length $\lfloor m/q \rfloor$. Assuming that each character occurs with probability $1/\sigma$, the probability that P^j occurs in a given

text position is $(1/\sigma)^{\lfloor m/q \rfloor}$. A brute-force verification cost is in the worst case $O(m)$ (and only $O(1)$ on average, but using the pessimistic bound does not deteriorate the final obtained complexity and simplifies the analysis). To keep the total time at most $O(n/q)$ on average, we select q so that $nm/\sigma^{m/q} = O(n/q)$. This is satisfied for $q = m/(2\log_\sigma m)$, where the verification cost becomes $O(n/m)$ and filtering cost $O(n\log_\sigma(m)/m)$. The total average time is then dominated by the filtering time, i.e. $O(n\log_\sigma(m)/m)$, which is optimal.

In the following sections, we describe efficient applications of the above scheme for several string matching problems.

1.4.1 Relaxing q

The performance of all the algorithms we are going to present depends on the choice of q. Our analyses assume uniform distribution of characters, which is not a problem in most practical cases. For instance, good results for English language can be obtained by assuming uniform distribution for $\sigma \approx 16$ [NR02]. The real problem is that for some texts the distribution could change abruptly, and thus there is no single optimal q.[3]

If this becomes an issue, we can proceed as follows (this will work for all our algorithms):

- Compute the value of q assuming uniform distribution.
- Do all the precomputations for $q' \in \{1, 2, 3, \ldots, q, \ldots, m\}$.
- Initialize $q' = q$, and start searching, using q'.
- During the search, if the verification algorithm is reading significantly more characters than the filtering algorithm, then decrease q'.
- In the opposite case, i.e., if the filtering algorithm is reading significantly more characters than the verification algorithm, then increase q'.

The bookkeeping of the above operations is simple, and adds only a constant factor overhead to the whole algorithm.

1.4.2 Handling longer patterns

If $qm' > w$, we must use more computer words, and the running time is then multiplied by $O(\lceil qm'/w \rceil) = O(\lceil m/w \rceil)$, i.e. the average time becomes $O(n\log_\sigma(m)/w)$.

[3]The problem and the sketch of the solution was pointed out by G. Navarro, priv. comm., 2005.

However, the trick used in [PT03] to make BNDM work with $m > w$ can be applied to our algorithms too. The idea is to partition the pattern into $r = \lfloor m/h \rfloor$ consecutive parts. The length of each part is now $h = \lfloor (m-1)/w \rfloor + 1$. All the h characters of each part are then superimposed into a single character class. The resulting r character classes are then concatenated to form a single pattern of length r. This pattern fits into a single computer word, and it can be searched by reading only every hth character of the text. This turns any algorithm, where it is applied to, into a filter, so the potential matches must be verified. This technique permits long patterns for the average optimal Shift-Or as well. The result is an algorithm with $O(n \log_{\sigma/h}(m)/m)$ time on average. This is not optimal any more, but for $\sigma \gg h$ should work quite well.

1.4.3 Linear worst-case time

The worst-case running time of Alg. 4 is $O(nm)$. However, the verification algorithm is easy to combine with standard Shift-Or, so that the verifications take at most $O(n)$ total time. This is done as follows. Whenever we must verify a pattern occurrence, we do it with Shift-Or. The last text position verified is saved in a variable, as well as the state vector D (for plain Shift-Or). If the next verification area overlaps with the previous, we restore the Shift-Or search state from the previous verification. Otherwise, if the next verification area starts after the previous ended, we reinitialize the Shift-Or search state. The verification algorithm then reads every text character at most once, and therefore the time is at most $O(n)$ (or $O(n\lceil m/w \rceil)$ for long patterns). However, if the verification time becomes an issue, the filter does not work well, and one could use plain Shift-Or just as well.

1.4.4 Implementation

In modern pipelined CPUs branching is costly. In Alg. 2 there are two conditionals in the search code; first to detect the matches, and the second to check the end of the input. A simple way to avoid these to some degree is to unroll the line 5, i.e. repeat the code

$$D \leftarrow (D << 1) \mid B[T[i]]$$

inline several, say U, times (with increasing offsets for the variable i). This means that the bit $m-1$ of D, indicating on occurrence, will be overflowed due to the repeated shifts, and hence in line 6 we must detect if any of the

Alg. 6 Fast-Shift-Or(T, n, P, m).

```
1   for i ← 0 to σ − 1 do B[i] ← ((1 << m) − 1) << (w − U − m)
2   for i ← 0 to m − 1 do B[P[i]] ← B[P[i]] & ~(1 << (w − U − m + i))
3   D ← ~0; i ← 0
4   while i < n do
5       for r ← 0 to U − 1 do D ← (D << 1) | B[T[i + r]]
6       if ~D >> (w − U) ≠ 0 then report matches
7       i ← i + U
```

bits $m-1..m+U-1$ is zero. This means that we need $U-1$ extra bits, and the pattern length is therefore limited to $m \leq w-U+1$.

The second optimization involves detecting the matches. Line 6 in Alg. 2 involves a variable mm. This can be avoided if the bit vectors are aligned so that the highest bit is in position $w-U+1$, instead of in position $m+U-1$. This means that the matches can be detected with $\sim D >> (w-U) \neq 0$, which is efficient if U is constant.

These two simple optimizations (shown in Alg. 6) give about 2–5× speed-up for standard Shift-Or (Alg. 2), depending on the architecture. The line 5 in Alg. 6 is automatically inlined by compilers, for small constant U.

Unrolling speeds-up also the Optimal Shift-Or, but the second optimization cannot be applied in this case, since the bit positions indicating the matches are not consecutive. The unrolling technique uses $U-1$ extra bits per pattern, so we need $q(U-1+\lfloor m/q \rfloor)$ bits in total, which is $O(m(U+\log_\sigma m)/\log_\sigma m)$ with the optimal q. Alg. 7 gives the code.

Finally, observe that while unrolling is well suited to Shift-Or, the benefits are negligible e.g. for BNDM algorithm, since the more complex control logic cannot be avoided.

1.5 The Aho–Corasick algorithm

The Aho–Corasick (AC) algorithm [AC75] is a multiple string matching algorithm that runs in $O(n)$ worst-case time. Our application of the AC technique may serve both for single and multiple exact string matching. We briefly review how the algorithm works, more details can be found in [AC75]. The algorithm builds a finite state automaton recognizing the input pattern set. Basically, the automaton is a trie of all the r patterns, augmented with "fail" transitions. Hence the automaton has $O(rm)$ states. Let $label(s)$ be the label (substring) spelled out by the path from the initial state (root of the trie) to state s (a node of the trie). For a state s, the fail transition leads to state (node in the trie) s', such that $label(s')$ is the longest suffix

Alg. 7 Fast-Average-Optimal-Shift-Or(T, n, P, m, q).

```
1   for i ← 0 to σ − 1 do B[i] ← ∼0
2   h ← 0; mm ← 0
3   for j ← 0 to q − 1 do
4       for i ← 0 to ⌊m/q⌋ − 1 do
5           B[P[iq + j]] ← B[P[iq + j]] & ∼(1 << h)
6           h ← h + 1
7       for r ← 0 to U − 1 do
8           mm ← mm | (1 << (h − 1))
9           h ← h + 1
10      h ← h − 1
11  D ← ∼mm; i ← 0
12  do
13      for r ← 0 to U − 1 do D ← (D << 1) | B[T[i + rq]]
14      if (D & mm) ≠ mm then Verify(T, i, n, P, m, q, U, D)
15      D ← D & ∼mm
16      i ← i + Uq
17  while i < n
```

of $label(s)$ that is also a prefix of some pattern in the set. The resulting automaton can be used to search for every occurrence of the stored patterns from text T; the text symbols are read one by one, and for each symbol we advance using the trie transitions if a matching transition exists, if not, we follow the fail transitions until we reach the initial state, or a matching transition is found. If the automaton goes through a node that corresponds to a stored pattern, an occurrence is found. It is easy to see that the whole process takes only $O(n)$ steps.

Note that for each "fail" transition and alphabet symbol we can precompute the state where the symbol leads. This complicates the preprocessing, but searching algorithm becomes simpler and more efficient (in practice) as every state has an outgoing transition for every alphabet symbol and fail transitions become obsolete. However, the space becomes $O(rm\sigma)$.

Alg. 8 shows the pseudocode for preprocessing. The code builds the trie breadth-first, and simultaneously builds the full automaton directly. In the end, state 0 is the initial state. Each state s has a transition with symbol c stored as $AC.\delta[s][c]$. The set $AC.id[s]$ stores the set of pattern numbers that match for the state s.

1.5.1 Optimal Aho–Corasick

Again recall that we partition P to q patterns, $P^0, \ldots, P^{q-1}$. It should be clear that we can search for each P^i using AC, and adapting it for skipping

Alg. 8 Build-AC($\mathcal{P}, r$).

```
1   State ← 0
2   queue1 ← 1; queue2 ← 0
3   for i ← 0 to r − 1 do L[i] ← 0; ps[i] ← 0
4   done ← false
5   while not(done) do
6       done ← true
7       for i ← 0 to r − 1 do
8           if L[i] < |P^i| then
9               done ← false
10              c ← P^i[L[i]]
11              if AC.δ[ps[i]][c] = fail then
12                  State ← State + 1
13                  AC.δ[ps[i]][c] ← State
14              ps[i] ← AC.δ[ps[i]][c]
15              L[i] ← L[i] + 1
16              if L[i] = |P^i| then AC.id[ps[i]] ← AC.id[ps[i]] ∪ {i}
17      while queue1 ≤ queue2 do
18          for c ← 0 to σ − 1 do
19              s ← AC.δ[queue1][c]
20              if s ≠ fail then
21                  AC.fail[s] ← AC.δ[AC.fail[queue1]][c]
22                  AC.id[s] ← AC.id[s] ∪ AC.id[AC.fail[s]]
23              else
24                  AC.δ[queue1][c] ← AC.δ[AC.fail[queue1]][c]
25          queue1 ← queue1 + 1
26      queue2 ← State
27  return AC
```

text symbols can be done precisely as for Shift-Or. That is, the automaton is built for the q patterns P^i, and only every qth text symbol is read. If some P^i occurs in the text, we invoke verification. The analysis is also the same as for Shift-Or, i.e. the average time becomes $O(n \log_\sigma(m)/m)$. The only difference is that no assumption is made on the pattern length.

We note that we can apply our technique to any number of patterns simultaneously, as AC can search for any number of patterns. (Obviously, the same is true for the bit-parallel algorithms as well, but in practice the number of bits is too small.) The algorithm itself does not change much, the partitioning technique is simply applied to all the r given patterns, and searched for together. However, the verification probability increases, i.e. it is multiplied by r, and hence we must choose $q = O(m/\log_\sigma(rm))$, resulting in $O(n \log_\sigma(rm)/m)$ average time, which is again optimal. Alg. 9 gives the pseudocode for filtering, and Alg. 10 for verification.

Fig. 1.4 presents the full automaton for patterns {`atataa`, `acatta`}, using

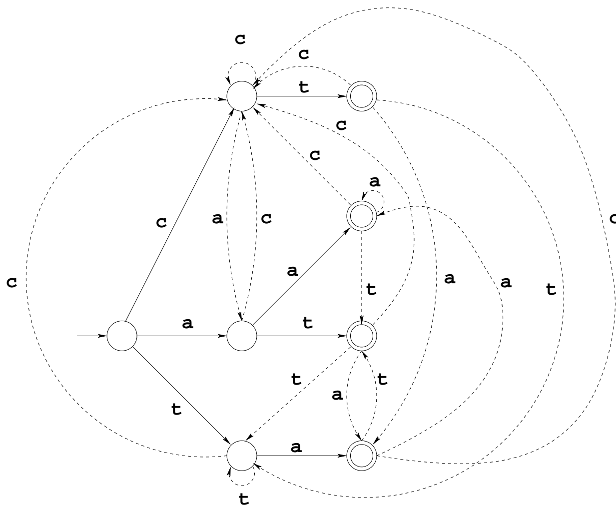

Figure 1.4: AC-automaton example

Alg. 9 Average-Optimal-AC(T, n, P, r, m, q).

1	**for** $i \leftarrow 0$ **to** $r-1$ **do**
2	**for** $j \leftarrow 0$ **to** $q-1$ **do**
3	**for** $k \leftarrow 0$ **to** $\lfloor m/q \rfloor - 1$ **do**
4	$\mathcal{P}^{iq+j}[k] \leftarrow P^i[kq+j]$
5	$AC \leftarrow$ Build-AC$(\mathcal{P}, rq)$
6	$s \leftarrow 0$; $i \leftarrow 0$
7	**while** $i < n$ **do**
8	$s \leftarrow AC.\delta[s][T[i]]$
9	**if** $AC.id[s] \neq \emptyset$ **then** VerifyAOAC$(T, i, n, P, m, q, AC, s)$
10	$i \leftarrow i + q$

the parameter $q = 3$, giving the pattern (multi-)set $\{\texttt{at}, \texttt{ta}, \texttt{aa}, \texttt{at}, \texttt{ct}, \texttt{aa}\}$. The solid arrows correspond to the trie of the set.

The number of states is still $O(rm)$ in the worst case, as in the case of standard AC automaton. However, if r is large as compared to σ, many pattern pieces can share the same prefix, which reduces the number of states in practice. In our case we have qr prefixes, and the pattern lengths are only $\lfloor m/q \rfloor$, and hence in practice the automaton has fewer states than plain AC.

Finally, we note that the worst-case time can be improved from the $O(nmr)$ to just $O(n)$, by using the same trick as for Shift-Or (see Sect. 1.4.3). That is, it is simple to use standard AC algorithm for the verifications, and by saving the search state the total worst-case time can be made $O(n)$.

Alg. 10 VerifyAOAC$(T, i, n, P, m, q, AC, s)$.

```
1   for i ← 0 to |AC.id[s]| − 1 do
2       id ← ⌊AC.id[s][i]/q⌋
3       os ← AC.id[s][i] mod q
4       c ← −(⌊m/q⌋ − 1) q − os
5       if P^id[0...m − 1] = T[i + c...i + c + m − 1] then report match
```

1.6 Application of the AOSO technique in other algorithms

The proposed technique of parallel searching for sampled subpatterns can be applied for several other algorithms as well [FG05b, FG09a]. We present them below, with an exception of the average-optimal Shift-Add algorithm which solves the k-mismatches problem, and its presentation was thus shifted to Chapter 2, dedicated to approximate string matching.

1.6.1 Pattern matching with swaps

The problem of matching with local swaps (also called transpositions) is to report all text substrings $T[i \ldots i{+}m{-}1]$ such that every text symbol $T[i{+}j]$ matches either $P[j]$, or $P[j-1]$, or $P[j+1]$. In other words, any pair of adjacent symbols of P is allowed to be swapped, but no symbol can participate in more than one swap. The best known complexity for this problem is $O(n \log m \log \sigma)$ [ACH$^+$01], and recently also a bit-parallel algorithm was shown [IR08b], with $O(n \log m \lceil m/w \rceil)$ worst-case time complexity. The best bit-parallel algorithms [Fre00] solve the problem in $O(n\lceil m/w \rceil)$ and $O(n \log_\sigma(m)/m)$ worst- and average-case (for $m = O(w)$) times, respectively.

Our average-optimal Shift-Or algorithm can be adapted for this problem as well. The only essential change is in the preprocessing; the mask $B[c]$ has hth bit set to 0, iff $P[iq{+}j] = c$, or $P[iq{+}j{-}1] = c$, or $P[iq{+}j{+}1] = c$ (cf. Alg. 4, line 5). In this way, more verifications are needed compared to the exact matching problem, but it is easy to notice that on average the match probability for a pair of symbols, at any sampled position of T, is upper-bounded by $3/\sigma$, i.e., grows only by a constant, hence the $O(n \log_\sigma(m)/m)$ average time complexity for the simpler problem remains (for $m = O(w)$), as each verification takes $O(m)$ time in the worst case, and only $O(1)$ time on average. (Note that we cannot use the probability estimation of $3/\sigma$ for the case of $\sigma < 4$. The precise probability formula, $p = 1 - (1 - 1/\sigma)^3$, should then be used, which makes showing the average time complexity equally

trivial.) For longer patterns, the trick described in Sect. 1.4.2 can be used again, leading to $O(n \log_{\sigma/h}(m)/m)$ average time.

1.6.2 Pattern matching with all circular shifts

In the problem of matching with all circular shifts, we are interested in reporting matches between a substring of the text and any rotation $P[i \ldots m]$ $P[0 \ldots i{-}1]$ of the pattern [IR08a]. The best known solution needs $O(n \log \sigma)$ time.

Again, we care for the average case rather than the worst case. To this end, we can partition each of the m rotations into q evenly spaced subsequences, and search for all the mq strings with a multiple matching algorithm, namely AC again. Any match (of length $\lfloor m/q \rfloor$) is verified naïvely in $O(m)$ time per matching piece. The analysis is then the same as before, but using $r = mq$, which gives $q = O(m/\log_\sigma(m))$, and $O(n \log_\sigma(m)/m)$ average time.

The problem with that analysis is that now the patterns (i.e., rotations of P) are not independent. Still, we can make use of the analysis of several problems with transposition invariance [FMN06]. In transposition invariance, P matches (exactly) the text substring $T[j \ldots j + m - 1]$, if there exists a $t \in \{-\sigma, \ldots, \sigma\}$ such that $P[i] + t = T[j + i]$ for every i. The problem was solved by generating all the $O(\sigma)$ possible transpositions and resorting to multiple matching. Our case is similar: the generated patterns are not random, but depend on the original pattern. However, the authors of the cited work showed [FMN06, Sect. 5.3] that the average-case complexity analysis that assumes uniform distribution of the pattern symbols is still valid, even if the generated patterns are not independent. This analysis generalizes straightforwardly to our case. We can thus mimic all the steps of their proof, with necessary modifications [FG09a], showing that our $O(n \log_\sigma(m)/m)$ average-case bound is valid. This is also the lower bound for the problem, as otherwise we could solve the exact string matching problem faster, by using some faster algorithm for circular shifts as a filter. Hence our algorithm is optimal on average.

Finally, we note that the average-optimal Shift-Or algorithm could be adapted as well, by generating a pattern $P' = P[0 \ldots m-1]P[0 \ldots m-2]$. The substrings of P' include all the substrings of all the circular rotations of P, and hence we could use P' as a filter (but still using text windows of length m only; note that this trick does not work with all algorithms, but AOSO poses no problems). As $|P'| = O(m)$, the complexity remains the same as for the exact matching.

1.6.3 (δ, γ)-matching

In the problem of (δ, γ)-matching, the pairs of corresponding integer symbols of P and T are allowed to differ by at most δ and the total sum of absolute values of those differences must not exceed γ; see Chapter 3, where similar problems are discussed at length. For this application, we can use our techniques to modify the algorithm from [CIN+05], which runs in $O(n\lceil m(1+\log(\gamma+1))/w\rceil)$ time in the worst case, to obtain a filtering algorithm working in $O(n\lceil m(1+\log(\gamma+1))/w\rceil/q)$ on average. Assuming uniform random distribution of characters, we obtain $O(n \log_2 \gamma \log_{\sigma/\delta}(m)/w)$ asymptotic average time by selecting $q = O(m/\log_{\sigma/\delta} m)$.

1.7 Experimental results

The experiments with our algorithms focus on the average-optimal Shift-Or algorithm, but we give also some results for the average-optimal Shift-Add and Aho–Corasick algorithms. They are not meant to be exhaustive, still they show the potential of our techniques. It can be seen from the results that we can easily beat some of the best previous algorithms.

1.7.1 Shift-Or and Shift-Add experiments

Chronologically, the experiments were first run on a Pentium4 and a UltraSPARC IIIi platform [FG05b], and much later repeated on a more powerful machine, equipped with an Intel Core 2 Duo CPU [FG09a]. We start with the older results (only for exact matching).

All algorithms were implemented in C. The compiler used for the P4 machine was `icc 8.1`. Namely, the experiments were carried out on a 2.4 GHz Pentium4 with 512 MB RAM, 512 KB cache, running GNU/Linux 2.4.20-8. Some experiments were then performed with an 1.28 GHz UltraSPARC IIIi with 16 GB RAM, 1 MB cache, running SunOS 5.9. In this case, the used compiler was Sun ONE Studio 8 C.

We performed the experiments using random ASCII ($\sigma = 96$, $n =$ 10 MB), and several real texts. These are: E.coli DNA sequence (4 638 690 characters) from Canterbury Corpus,[4] real protein data (5 050 292 characters) from TIGR Database (TDB),[5] and natural language text (the collected works of Charles Dickens, 10 192 446 characters), from Silesia Corpus.[6] The

[4]http://corpus.canterbury.ac.nz/descriptions/

[5]http://www.tigr.org/tdb

[6]http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

patterns were randomly extracted from the texts, and each test was repeated 100 times. We report the average speed in megabytes per second.

The algorithms included in the experiments were the following:

- BNDM – The baseline algorithm [NR00] (implemented by Kimmo Fredriksson), one of the best known and most effective of the bit-parallel algorithms,
- SBNDM – Simplified version of BNDM [Nav01b, PT03] (implemented by Kimmo Fredriksson); in practice faster, but examines more text characters,
- AOSO – Our Average-Optimal Shift-Or algorithm,
- FAOSO – Fast variant of AOSO, using unroll factor of $U = 4$,
- Shift-Or – Plain classical Shift-Or algorithm [BYG92],
- Fast Shift-Or – Fast variant of Shift-Or, using unroll factor of $U = 4$.

Moreover, in the newer experiments on the Intel Core 2 Duo, the range of tested algorithms was extended with:

- BNDM2 – The fastest algorithm of the BNDM family [HD05],
- AOSOA – Adaptive version of AOSO,
- AOSA – Our Average-Optimal Shift-Add algorithm for matching under Hamming distance (cf. Sect. 2.3.1).

The variants of BNDM are considered to be the fastest general purpose exact string matching algorithms for $m \leq w$. We also compared against the Boyer–Moore–Horspool algorithm [Hor80], and Boyer–Moore–Horspool–Sunday algorithm [Sun90], but these were not competitive, so we do not report the speeds here.

For AOSO and FAOSO the optimal q value was found experimentally (however, especially FAOSO is not too sensitive to the exact value of q, for long patterns slightly too small q values do not degrade the performance much).

Table 1.1 gives the speeds in megabytes per second for all the texts. AOSO denotes our Average-Optimal Shift-Or algorithm, and FAOSO the fast variant of it, using the unrolling trick. Additional results (not included in the table): Shift-Or processes 131 MB/s, 128 MB/s, 128 MB/s and 132 MB/s, and the fast Shift-Or 776 MB/s, 764 MB/s, 817 MB/s and 820 MB/s for DNA, proteins, natural language and random ASCII, respectively. Note that the speeds for the plain Shift-Or do not depend on the pattern length. For the fast variants, we used unrolling factor $U = 4$, when the representation fitted into a single computer word, otherwise we were forced to use values $1 \ldots 3$.

As it can be seen, our algorithms are clearly the fastest on DNA in all the cases. Interestingly, the fast variant of the plain Shift-Or algorithm beats

DNA				
m, q	AOSO	FAOSO	BNDM	SBNDM
4, 2	321	**503**	181	210
8, 2	539	**763**	312	357
12, 3	702	**941**	438	492
16, 3	1029	**1229**	567	598
20, 4	1079	**1341**	750	804
24, 4	1229	**1525**	1106	1164
28, 5	1427	**1638**	1106	1164

proteins				
m, q	AOSO	FAOSO	BNDM	SBNDM
4, 2	580	**909**	415	512
8, 4	944	**1267**	642	678
12, 4	1120	**1376**	816	926
16, 4	1120	**1459**	963	1025
20, 4	1235	**1376**	1175	1204
24, 5	1267	**1338**	1235	1302
28, 6	1302	1302	1302	1302

natural language				
m, q	AOSO	FAOSO	BNDM	SBNDM
4, 2	579	**884**	368	476
8, 4	1034	**1262**	685	778
12, 4	1144	**1279**	797	845
16, 5	1200	**1389**	831	944
20, 6	1279	**1389**	1013	1092

random ASCII				
m, q	AOSO	FAOSO	BNDM	SBNDM
4, 2	599	952	633	**1053**
8, 4	1124	**1333**	1064	1220
12, 4	1250	**1389**	1299	1282
16, 4	1351	**1389**	1351	**1389**
20, 6	1449	**1471**	1370	1429

Table 1.1: Exact matching. Searching speed in megabytes per second for different algorithms on Pentium4.

our average optimal Shift-Or for $m \leq 8$. The results are quite similar for proteins, but for long patterns BNDM variants have equal performance to our algorithms. Note also that for all cases SBNDM is consistently slightly faster than BNDM. Our approach is faster also in natural language text, while on random ASCII the differences are considerably smaller.

The experiments were repeated on UltraSPARC IIIi for DNA and natural language and can be seen in Table 1.2. Additional results: Shift-Or processes 91 MB/s and 90 MB/s, and the fast Shift-Or 168 MB/s and 165 MB/s on DNA and natural language, respectively. FAOSO is again clearly the fastest alternative, but contrary to the results on Pentium4 the plain AOSO is not competitive.

Now we present newer experimental results. The test machine was changed to an 3.0 GHz Intel Core 2 Duo (E6850) with 2 GB RAM, 4 MB cache, running GNU/Linux 2.6.22.4 and `icc 10.0` compiler. The datasets

m, q	AOSO	FAOSO	BNDM	SBNDM
4, 2	70	**109**	103	92
8, 2	104	**193**	146	141
12, 3	132	**227**	171	164
16, 3	135	**234**	194	192
20, 4	161	**256**	207	207

m, q	AOSO	FAOSO	BNDM	SBNDM
4, 2	104	**198**	142	147
8, 4	157	**250**	193	192
12, 4	160	**256**	217	220
16, 4	175	**267**	232	233
20, 6	189	**275**	244	247

Table 1.2: Exact matching. Searching speed in megabytes per second for different algorithms on UltraSPARC IIIi. Left: DNA; right: natural language.

and the test methodology did not change.

Again, the Boyer–Moore–Horspool algorithm (including optimized variants, using fast skip-loops) and Boyer–Moore–Horspool–Sunday algorithm were not competitive, so we do not report the results.

Besides BNDM2, the authors of [HD05] presented several other BNDM variants, but BNDM2 was clearly the best in our experiments.

The algorithm AOSOA uses an adaptive method for selecting q (see Sect. 1.4.1), implemented as follows: the initial value is $q = m$; every time a verification is invoked, q is decremented by one; after every 256th text access by the filter, q is incremented by one. This simple strategy works well in practice. AOSOA also uses a simple unrolling method (inner loop is simply repeated 4 times), but not the more advanced and efficient method used by FAOSO.

Table 1.3 gives the speeds in megabytes per second for all the texts. The q values reported correspond both to AOSO and FAOSO. As it can be seen, our algorithms are clearly the fastest on DNA in all the cases. Interestingly, the fast variant of the plain Shift-Or algorithm beats our average optimal Shift-Or for short patterns. FAOSO is the best alternative also for natural language, but in some cases the gap against BNDM2 is small. The results for proteins and random ASCII are worse for the proposed algorithm; in some cases BNDM2 wins by a wide margin.

In general, the best algorithms are FAOSO and BNDM2, with only a few exceptions. The main problem with FAOSO (and AOSO) is that q must be an integer, and this forces too small values in some cases. The problem with BNDM2 is that, assuming that an efficient implementation unrolls U times, it can shift only after reading U characters, and the maximum shift is reduced to $m - U + 1$. Our algorithms do not have such limitations.

Finally, Table 1.4 gives the speeds for the average-optimal Shift-Add (AOSA) for Core 2 Duo. Our character skipping technique clearly speeds-up Shift-Add as well, the exception being short patterns or large k on DNA alphabet, where our algorithm essentially degenerates to plain Shift-Add.

1.7.2 Aho–Corasick experiments

We implemented also an AC-automaton, and its average-optimal version (AOAC). The implementation comprises a full automaton without the fail transitions. For a comparison, we used the Backward Set Oracle Matching (BSOM) algorithm [AR99, NR02] (implemented by its authors). This is a simplified version of multiple BDM algorithm, but it has been experimentally shown that BSOM is always faster than BDM [NR02], and one of the fastest

DNA						
Shift-Or: 478, Fast Shift-Or: **1164**						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
4, 2	329	395	508	334	400	567
8, 2	802	**1474**	851	592	707	819
12, 4	983	**2011**	1301	825	1001	1079
16, 4	1474	**2458**	1638	1022	1286	1382
20, 4	1695	**3276**	2011	1222	1563	1638
24, 4	1762	**3597**	2107	1427	1843	1923
28, 4	1777	**3717**	2458	1602	2116	2212

proteins						
Shift-Or: 473, Fast Shift-Or: 1151						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
4, 2	882	**1605**	892	753	917	1473
8, 4	1553	2603	1417	1120	1294	**2992**
12, 4	1720	3676	2189	1416	1738	**4081**
16, 8	2676	3853	2676	1852	2271	**4816**
20, 8	2676	3853	3010	2189	2816	**5235**
24, 8	3211	**5873**	3705	2675	3368	5473
28, 8	3440	**5873**	4014	3211	3947	5734

natural language						
Shift-Or: 476, Fast Shift-Or: 1151						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
4, 2	798	**1450**	817	710	816	1350
8, 4	1495	**2492**	1369	1020	1171	2430
12, 4	1735	**3600**	2160	1275	1555	3240
16, 4	1767	**3641**	2627	1502	1948	3600
20, 6	2558	**4628**	2859	1714	2337	3888
24, 8	3240	**5143**	3600	1921	2730	4050
28, 8	3240	**5282**	4050	2118	3076	4226

random ASCII						
Shift-Or: 477, Fast Shift-Or: 1152						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
4, 2	901	2000	1149	1316	**2150**	1339
8, 4	1786	**3636**	2041	2222	3125	2967
12, 6	2564	**4878**	2941	2941	3636	4367
16, 8	3330	**5556**	3571	3300	4000	5495
20, 10	4000	5714	4348	3703	4348	**5814**
24, 12	4546	5882	4546	4167	4444	**6024**
28, 12	4546	5882	4762	4348	4651	**6289**

Table 1.3: Searching speed in megabytes per second for different algorithms on Core 2 Duo

Alg	AOSA	AOSA	AOSA	AOSA	AOSA	Shift-Add
m k	$m = 8$ $k = 1$	$m = 12$ $k = 1$	$m = 16$ $k = 1$	$m = 8$ $k = 2$	$m = 16$ $k = 2$	$m = 8 \dots 16$ $k = 1 \dots 2$
DNA	379	702	834	379	681	379
Proteins	816	944	1554	438	860	379
NL (ASCII)	784	875	1519	397	860	379
Rnd (ASCII)	855	1613	1724	826	1587	379

Table 1.4: k-mismatches. Searching speed in megabytes per second for Average-Optimal Shift-Add on Core 2 Duo.

algorithms for moderate to long patterns, the competitiveness increasing also for increasing alphabet sizes.

The experiments were run using pattern set sizes $r \in \{1, 16, 64\}$ and pattern lengths $m \in \{8, 16, 64\}$. The results are shown in Table 1.5 for Core 2 Duo. Our algorithm is always faster than BSOM, but loses to plain AC for short DNA patterns for $r = 16, 64$. The reason is that in these cases the optimal q is 1, i.e. AOAC degenerates to plain AC, with additional complexity. However, for larger alphabets and longer patterns our approach is better by far.

1.8 Conclusions

The exact string matching is a fundamental problem with over 30 years of research history. Pondering over this problem, we have managed to find a novel, yet extremely simple, filtering technique which has a surprising number of applications in string matching algorithms. The resulting new algorithms often have optimal running times on average, and have simple implementations, which helps them achieve very competitive speeds in practice, what was demonstrated in comparative experiments on three different hardware platforms (Pentium4, UltraSPARC IIIi and Core 2 Duo). The simplicity comes from a novel forward matching technique (as opposed to backward matching as in most competing algorithms) and from the fact that the pattern shifts are constant. This also leads to simple unrolling trick that boosts the search in modern hardware. This trick cannot be applied so successfully to more complex backward matching algorithms.

With regard to exact string matching, we showed how our technique can be used to modify the well-known bit-parallel algorithm, Shift-Or, to achieve the optimal running time on average, for short patterns. We call our algorithm FAOSO. Interestingly, the same idea can be used for other exact string matching algorithms, as we showed in [FG09a] on the example of

DNA			
m, r, q	AOAC	AC	BSOM
8, 1, 2	**691**	421	394
16, 1, 4	**1341**	421	504
64, 1, 16	**3403**	421	1593
8, 16, 1	385	**417**	127
16, 16, 2	**737**	413	234
64, 16, 9	**1580**	395	725
8, 64, 1	323	**369**	61
16, 64, 2	**481**	357	153
64, 64, 9	**714**	297	433

proteins			
m, r, q	AOAC	AC	BSOM
8, 1, 4	**1338**	422	708
16, 1, 8	**2535**	419	1170
64, 1, 16	**5351**	419	3265
8, 16, 2	**764**	415	303
16, 16, 5	**1505**	412	554
64, 16, 16	**2349**	385	1517
8, 64, 2	**645**	395	216
16, 64, 4	**958**	376	396
64, 64, 16	**1021**	295	775

natural language			
m, r, q	AOAC	AC	BSOM
8, 1, 4	**1312**	419	729
16, 1, 8	**2090**	419	1098
64, 1, 20	**5064**	419	2682
8, 16, 2	**741**	415	250
16, 16, 4	**1361**	414	457
64, 16, 16	**2576**	400	1234
8, 64, 2	**525**	375	141
16, 64, 4	**894**	360	279
64, 64, 16	**1225**	315	680

random ASCII			
m, r, q	AOAC	AC	BSOM
8, 1, 4	**1520**	420	1210
16, 1, 8	**2860**	420	1980
64, 1, 32	**5710**	418	3830
8, 16, 4	**1390**	417	510
16, 16, 8	**2250**	415	1030
64, 16, 32	**3300**	398	2340
8, 64, 4	**940**	405	376
16, 64, 8	**1300**	394	680
64, 64, 20	**1590**	345	1120

Table 1.5: Multiple pattern search. Searching speed in megabytes per second for different algorithms on Intel Core 2 Duo.

modifying the brute-force algorithm. The resulting method was very simple, with sublinear average search time, yet not competitive to FAOSO in practice (due to this reason, we omit the presentation of this algorithm in this thesis). Our best result for exact matching, in asymptotic terms, is based on building the Aho–Corasick automaton for a number of subsequences of the given pattern. The algorithm achieves the optimal $O(m+n\log_\sigma(m)/m)$ average time, without any limitation on the pattern length. Generalizing this algorithm for multiple patterns is straightforward and again optimal average search time is achieved for this problem. We note now that another application along these lines can be modifying the Karp–Rabin algorithm to improve its average time from $O(n)$ to the optimal $O(n\log_\sigma(m)/m)$. The key idea of the original algorithm is to calculate a signature (hash) from all pattern symbols, and compare it to a respective signature for a text substring. Matches have to be verified (usually in a brute-force manner), but mismatches with equal signatures are very rare. An important property of the algorithm is that the hash function can be calculated incrementally, paying $O(1)$ time per a text character. Our technique of skipping charac-

ters in regular intervals can also be applied to the KR algorithm. The only problem is that matching a signature built over m/q text symbols against all q signatures for subsequences of the pattern cannot be done, in brute-force manner, as it would not lead to any improvement in the average time complexity. Instead, we can use a hash table with signatures as keys, and restrict the signatures to, e.g., $\min(m, \log_2(n)/2)$ bits, instead of the standard whole machine word (which has at least $\log_2 n$ bits). In this way, the hash table gets quite cheap both in space and initialization time, and the average lookup time remains $O(1)$.

The introduced technique also handles several approximate matching problems. One is matching under Hamming distance, where we present two algorithms [FG09a], the first based on the Shift-Add algorithm (it will be discussed in Sect. 2.3.1), the other on brute-force; the former of them achieves the optimal average-case complexity for short patterns. Other models are pattern matching with swaps, and pattern matching with all circular shifts, where again our idea easily allows to achieve optimal average-case complexity, for short patterns in case of the former problem, and for any patterns in case of the latter one. Finally, the (δ, γ)-matching problem, motivated by music information retrieval, was discussed. The obtained average-case time is again optimal for short patterns. Our technique could be used with the classic pattern partitioning technique [Nav01a] as well, to solve string matching under Levenshtein distance (allowing k insertions, deletions or substitutions) in $O(nk \log_\sigma(m)/m)$ average time, optimal if $\log_\sigma m = O(1)$.

Some of the proposed algorithms have been shown in thorough experiments, with various real-world and artificial texts, to be very competitive in practice.

Chapter 2

Online approximate string matching

The problem of approximate string matching is to find the occurrences of pattern P in text T when some distortions of the pattern are allowed. More precisely, a distance function (dissimilarity measure) between two sequences must be specified, and all the start (or end) text positions aligned with the pattern where the distance between those two sequences does not exceed the given error level, are reported. The choice of the distance function depends on the application and constitutes the actual problem. The range of considered measures for approximate string matching is very wide. It contains several edit distance variants, Hamming distance, reversals, block distance, q-gram distance, (δ, γ, α)-distance and its particular cases, and many more. The main application areas comprise computational biology, text retrieval and signal processing, but the number of applications is greater, to name anti-virus software, data mining, OCR algorithms or query-by-humming systems retrieving information from music databases.

Approximate string matching, as defined in the previous paragraph, shifts a pattern along a (presumably, much longer) text sequence. A related problem is to calculate a global similarity of two sequences, usually of comparable length. Here, the most widely used measure is the length of the longest common subsequence (LCS), with applications in DNA sequence analysis and program versioning, to name a few.

The outline of this chapter is as follows. In Sect. 2.1 we present several widely used similarity measures and their main applications. Section 2.2 contains information on the basic tools and techniques used for approximate string matching: dynamic programming, automata, FFT, bit-parallelism and filtering. Section 2.3 describes our novel technique [GF08] of splitting counters in Shift-Add, a well-known bit-parallel algorithm for Hamming dis-

tance, to achieve the optimal parallelization. Applications of this idea in several other string matching problems are also presented. In Sect. 2.4 we show how our filtering technique [FG09a] from Chapter 1 can be applied for the k-mismatches problem (which is to report text positions in which the pattern matches with at most k Hamming errors), to achieve an average-optimal algorithm for short patterns. The next section discusses global similarity measures, focusing on the LCS measure and its variants. In particular, we point out our – theoretical and practical – results [NGMD05, GD08, DG09a] for the longest common subsequence with transposition invariance (LCTS) measure, an LCS variant devised for music information retrieval applications.

2.1 Similarity measures and their applications

Historically, the earliest application of approximate string matching was probably the problem of correcting misspelled words [Nav01a]. The first references go back to the 1920s and 1930s [Mas27], and even earlier, in 1918, the famous phonetic algorithm Soundex [Knu73] was developed and patented. The essence and novelty of Soundex was to index names by sound rather than their spelling, and the matching of different names with alike pronunciation in this scheme can be considered a form of (application oriented) approximate matching. In the computer era the problem of approximate matching started to be treated in an algorithmic way in the 1960s. It was noted, for example, that about 80% of spelling errors could be corrected with a single character insertion, deletion, replacement or transposition [Dam64], a claim substantiated also in a more recent experimental work [Kuk92]. The need for correcting garbled words is ubiquitous. So-called typos can be entered manually, due to a writer's carelessness, or lexical incompetence, or orthography ignorance, or they can appear in the OCR process. Obviously, error-correcting capabilities are desirable in text editors, educational (language learning) software, command language interfaces, or programming environments, to name a few applications. Apart from error correction (which often poses language-specific problems and consequently implies language-specific solutions [DC05]), it is desirable to have error-tolerant search procedures implemented. One of the most important targets for approximate matching in the recent years are web searchers. Navarro in [Nav01a] reports that the search for the word "Levenshtein" with Altavista search engine gave more than 30% errors, allowing just a single deletion or transposition.

A related, but much more exhaustive, experiment was performed in 2001

by Dalianis [Dal02]. An examination of approximately one million queries to the web site of Swedish National Tax Board, mostly in Swedish language, revealed that almost 10% of them were misspelled or erroneous. Those examples, we believe, give some motivation for researching the area of approximate string matching. It could be also noted that the current leader in web searchers, the Google engine, suggests the correct spelling (e.g., asks "Did you mean: Levenshtein", if the entered term was "Levenschtein"), i.e., makes use of approximate matching algorithms over its term vocabulary.

To decide if a given sequence matches another, in an approximate sense, in most matching models a similarity measure, or distance, between the sequences, is specified. The distance $d(A, B)$ between strings A and B is the minimal cost of a sequence of operations that convert sequence A into B. Each of the allowed operations has a defined positive cost (weight). If all the operation weights are unary, the minimal-cost path to tranform A into B corresponds to the minimal number of operations to tranform A into B. The minimal-cost path does not have to be unique. If the transformation of A into B is impossible, the distance is ∞. The distance function is often, but not always, symmetric (i.e. $d(A, B) = d(B, A)$).

In most applications, the set of allowed operations contains:

- insertion,
- deletion,
- substitution (replacement),
- transposition.

Now we can present several most widely used distance functions.

Levenshtein *or* edit distance. It allows insertions, deletions and substitutions, usually with unary costs, and is often denoted with $ed(\cdot, \cdot)$. Note that a single transposition error (e.g., *the* – *teh*) is counted as two errors in the Levenshtein distance, as this can be corrected e.g. with a single deletion and a single insertion. Note also for this example that the correction path is not unique since two substitutions are just as good. In the literature the search problem under the Levenshtein distance is also called "string matching with k differences", where k is the number of allowed errors (the maximum distance allowed). This distance is symmetric and $ed(A, B) \leq \max\{|A|, |B|\}$. The volume of works dedicated to this matching model is large and was surveyed by Navarro in [Nav01a].

Indel distance. It allows only insertions and deletions. This distance is also symmetric and $d_{\mathrm{ID}}(A, B) \leq |A| + |B|$. Most algorithms developed for Levenshtein distance can be easily adopted to indel distance.

Hamming distance. It allows only substitutions. Note $d_\mathrm{H}(A,B) = \infty \iff |A| \neq |B|$. The matching problem is also called "string matching with k mismatches", but we should distinguish between reporting the Hamming distance for each position of the text (which clearly needs $\Omega(n)$ time even in the best case in the character model, i.e., if only single characters are read at a time), and reporting the text positions for which the pattern is aligned with at most k errors, which can be done much faster on average. Because the matching pattern and the corresponding text area must be of equal length, those two problems are amenable for FFT-based approaches (see Sect. 2.2.3).

Damerau distance. This model comprises all the Levenshtein errors, i.e., insertions, deletions and substitutions, but also consider a transposition of two adjacent symbols as a single error.

γ (or L_1) distance. This model requires an ordered (typically, integer) alphabet, and the distance between the pattern and its corresponding text excerpt, of equal length, is the sum of absolute differences over all the corresponding pairs of symbols, and should not exceed a given γ. Such a measure is motivated by music information retrieval, where retrieving a melody based on a "pattern" sung or whisted by an untrained human should be tolerant to false notes, as long as the total amount of imprecisions does not exceed a reasonable limit. Another variant of this measure is called (δ, γ), where apart from the total error γ, a limitation δ on individual symbols (music notes) is also imposed.

2.2 Basic techniques

Although the Cartesian product of various approximate matching problems and their possible scenarios (short/long patterns, repetitive/non-repetitive patterns, small/large alphabet, low/medium/high allowed error level, etc.) is huge and makes room for plenty of algorithms (many of which are competitive, in theory or in practice, at least in some niches), almost all the known solutions and techniques can be classified into only five areas. Those fundamental approaches to solve different approximate matching problems are:

- dynamic programming,
- automata,
- FFT,

- bit-parallelism,
- filtering.

They are not totally distinct or independent; for example, many algorithms based on bit-parallelism are specific simulations of non-deterministic finite automata. In the following subsections we take a closer look at each of those four approaches.

2.2.1 Dynamic programming algorithms

This is the oldest area, very important but no longer very active. Still, some of our results (to be presented in Sect. 5.3) are based on dynamic programming. Let us present a simple dynamic programming (DP) algorithm for calculating the Levenshtein distance between two sequences A and B, of length n and m, respectively. The idea is to fill – one by one – the cells of a two-dimensional $(n+1) \times (m+1)$ matrix C, in a way that each cell (i, j), $i, j \geq 0$, gets the value of the Levenshtein distance between the sequence prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$.

Note that empty prefixes correspond to the cells at two adjacent borders of the matrix, let us say, the topmost row and the leftmost column. No matter what A and B are, $C_{i,0} = i$, for any i, and similarly $C_{0,j} = j$, for any j. This is because converting an empty string into a(ny) string of length j requires j operations (insertions only) and converting a(ny) string of length i into an empty string requires i operations (deletions only). This simple observation allows us to fill "blindly" the topmost row and the leftmost column of C with the increasing numbers. For the Levenshtein distance (and many other distances) the value in each cell (except in the leftmost column / topmost row) depends only on its three neighbors: upper, left and upper-left. Precisely, the formula for $C_{i,j}$ is:

$$\begin{aligned} C_{i,j} = \textbf{if } & (A_i = B_j) \textbf{ then } C_{i-1,j-1} \\ & \textbf{else } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}), \end{aligned}$$

and the bottom-right corner cell stores the final answer. Fig. 2.1 illustrates.

To understand the formula, it is convenient to consider scanning in parallel both sequences, A and B, from left to right, possibly making "false moves" from time to time: either move one character forward in A but not in B, or move one character forward in B but not in A, or move one character forward in both sequences but the newly read characters are different. All those cases add up a single error, and only passing over matching characters in both sequences does not increase the overall error. The Levenshtein distance is the minimum error over all such possible traversals.

	ε	R	O	S	E	S
ε	0	1	2	3	4	5
C	1	1	2	3	4	5
O	2	2	1	2	3	4
W	3	3	2	2	3	4
S	4	4	3	2	3	3

Figure 2.1: Dynamic programming for Levenshtein distance calculation, $ed(A, B) = 3$

We go back to the DP formula. If the two prefixes, $A[1 \dots i]$ and $B[1 \dots j]$, end with same character, those last characters could be truncated without affecting the distance between prefixes, hence the first part of the formula is implied. If those last characters are not equal, we must consider three possibilities for the "last step" in a run of operations converting one of the prefixes into the other. One is that $A[i]$ has just been inserted, i.e., the previous state corresponds to the cell $C_{i-1,j}$. The other possibility is that $B[j]$ has just been inserted, and the previous state corresponds to the cell $C_{i,j-1}$. Finally, it is possible that $A[i]$ was replaced with $B[j]$, which directs us to the cell $C_{i-1,j-1}$. The fundamental assumption of dynamic programming is that the previous states are already known, and in this case it means, in particular, that we know the distances between $A[1 \dots i-1]$ and $B[1 \dots j]$, $A[1 \dots i]$ and $B[1 \dots j-1]$, and $A[1 \dots i-1]$ and $B[1 \dots j-1]$. To find $ed(A[1 \dots i], B[1 \dots j])$, we must add the error cost (i.e., 1) of the last operation to the minimum of the mentioned distances, which gives us the formula above.

The described routine only finds the distance (path length), not the mismatching characters. To actually find the sequence of operations converting A into B, we need to traverse back the matrix, jumping up-left to the previous row at the update position. One should remember that the path doesn't have to be unique. The algorithm complexity is $O(nm)$ in time, even for the mere distance calculation. As for finding the sequence of operations, it is also $O(nm)$ in space, as we need to store the entire matrix "or at least an area around the main diagonal" [Nav01a]. The latter observation lets us think that in output-dependent terms the space can be smaller, namely $O(np)$, where $p = ed(A, B)$.

Interestingly, in 1975 Hirschberg [Hir75] presented a brilliant idea based on the divide-and-conquer paradigm, which decreases the space usage to

$O(n)$, while retaining the $O(nm)$ time. Note that for the mere distance calculation achieving the linear space complexity is trivial, since it is enough to always store only the current and the previous row. Alternatively, we can traverse the matrix column-wise, and then the space usage turns into $O(m)$.

Demonstrating dynamic programming on an example of edit distance is popular (due to the importance and simplicity of that distance), but it should be stressed that in general a given cell $C_{i,j}$ value may not depend only on $A[i]$, $B[j]$ and the three neighboring cells. The only requirement, assuming row-wise traversal of the DP matrix, is that we should be able to calculate $d(A[1\ldots i], B[1\ldots j])$ and write it into $C_{i,j}$, knowing all the other distances between prefixes of $A[1\ldots i-1]$ and B, plus the distances between $A[1\ldots i]$ and prefixes of $B[1\ldots j-1]$ (symmetrically, we can specify when column-wise DP calculations are possible). In fact, many more cells than three are typically needed to calculate a cell value in matching with gaps [CCF05a, FG06c]. This undemanding requirement makes the DP approach flexible and working naturally with many similarity measures.

The main problem with DP algorithms is the high computation time. Note that the quadratic complexity is not only for the worst case, but occurs for any pair of patterns. Hence, the research on speeding-up DP algorithms was twofold: focusing on improving the worst case, and focusing on the average case. We assume the Levenshtein distance below, although many of the sketched techniques are of wider use.

As for the worst case, yet in 1980 Masek and Paterson [MP80] gave an algorithm with $O(nm/\log_\sigma^2 n)$ time complexity. Note that the complexity strongly depends on the alphabet size σ (works best for the binary alphabet). For large enough σ, the algorithm turns into the plain DP procedure. The Masek and Paterson algorithm is based on the so-called Four-Russians technique [ADKF75]. They split the matrix into $r \times r$ boxes, where r is small enough (with use of $O(n)$ extra memory, r does not exceed $\log_{3\sigma} n$), and precompute them one by one (in some order, e.g. row-wise and from left to right in rows), where a given box's input are: the corresponding chunks from both sequences, the gathered distance value from the bottom-right cell of the upper-left box (which is already known), and the last column and last row, i.e., the left and the top "walls" of the current box. An important idea is to compactly represent those left and top "walls". This is possible since, from basic properties of the DP matrix in the Levenshtein distance calculation, adjacent cells can differ only by -1, 0 or $+1$.

Overall, there are $m(3\sigma)^{2r}$ different cells to precompute, which, for the r specified earlier, leads to the given subquadratic final complexity.

Ukkonen is the inventor of the approach later called "diagonal transition

algorithms" [Ukk85a]. He noticed that the upper-left to lower-right diagonals are monotonically increasing (with subsequent differences 0 or 1 only), and it is enough to quickly find all the cells in which the values are incremented. A practical implementation finding each such cell in $O(1)$ time is not so simple though, and requires a "heavy" indexing structure, the suffix tree. Still, for two strings A and B, the edit distance between them can be calculated in $O(ed(A, B)^2)$ time. In the search problem variant it is crucial to know if the distance for the current alignment is $\leq k$ (i.e., match). Ukkonen's algorithm checks this condition in $O(k^2)$ time. The following algorithms along these lines were from Landau and Vishkin; in their last work in the series [LV89] they achieved $O(kn)$ worst-case time and $O(n)$ space for the search problem. Note that trivially adapting the Ukkonen's algorithm for this setting would result in $O(nk^2)$ time, which was hardly ever attractive. Other algorithms of this kind are e.g. [Mye86, CL94, CH98]. The last of the mentioned results has the worst-case time complexity of $O(n(1 + k^4/m))$ (for some patterns it is faster), i.e., can be linear for non-constant (but small enough) k.

The average-case oriented algorithms are much more important in practice. Ukkonen [Ukk85b] improved the average time complexity to $O(kn)$ (interestingly, Ukkonen gave only a short note describing this idea and conjecturing the average time complexity; the real proof was given later, in a work by Chang and Lampe [CL92]), while the space remained $O(m)$. Obviously, the DP matrix is traversed column-wise. The algorithm is based on a simple but powerful trick, so-called cut-off heuristic. Values in columns, if we go down, do not ever decrease. The cells with values larger than k are irrelevant, in the sense we do not need to know their exact values. We simply know they are inactive. Hence, it is needed to keep track of the last active cell in each column. Let it be in row i for some particular column. The next column can thus be calculated down only to row $i + 1$ (all the cells below must be inactive). The cut-off trick was adapted by Cantone et al. [CCF05b], and Fredriksson and Grabowski [FG05a, FG06a, FG08a] for (δ, α)-matching and related problems. More details will be given in Chap. 3.

An algorithm from [Mye86] is based on the diagonal transitions, and is a trivial (but quite practical) simplification of the sophisticated algorithms in [Ukk85a] and following ones. This variant is based on brute-force but its $O(kn)$ average time complexity follows immediately. An original approach was taken in [CL92]. The algorithm works column-wise, and partitions each column into runs of strictly increasing cells. As the average length of the runs is $O(\sqrt{\sigma})$ (which was proved only in [Nav01a, pp. 13, 25]), and the algorithm incorporates also the cut-off heuristic, its average search time gets $O(kn/\sqrt{\delta})$, making this algorithm the fastest in its class.

2.2.2 Algorithms based on automata

With this approach it is possible to reach the worst-case lower bound for the Levenshtein distance, which is $O(n)$, but at a cost of space growing exponentially in m and k, hence it is not very practical. Again, this approach was initiated by Ukkonen. In [Ukk85b], he constructed a deterministic finite state automaton (DFA), whose states corresponded to the full possible set of values for the columns in the DP matrix. Once the automaton is built, the text is scanned with it, with clearly $O(1)$ time per input character. Still, of course, the space use was immense. Some optimizations and non-trivial analyses in the same work made possible to bound the number of the automaton states with $O(\min\{3^m, m(2m\sigma)^k\})$, which was, many years later, further refined by Melichar [Mel96]. Nevertheless, in spite of those improvements, the DFA-based algorithm is completely intractable apart from short patterns or very small error levels.

2.2.3 Fast Fourier transform based algorithms

The idea of using Fast Fourier transform (FFT) for approximate text matching is actually quite old [FP74], but recently a couple of new algorithms using this approach have been presented, especially for music information retrieval motivated problems. In particular, it is possible to solve the δ-matching and (δ, γ)-matching problems in $O(\delta n \log m)$ time, and the γ-matching problem in $O(n\sqrt{m \log m})$ time [CCI05]. Another FFT-based technique can reach a better complexity for δ-matching, namely $O(\sqrt{\delta} n \log m)$, and $O(n\gamma \log \gamma)$ for matching with γ total error limit [ALPU05].

For Hamming distance / k-mismatches, convolutions and FFT can be used to get $O(n\sigma \log m)$ [FP74], and with a more refined technique, just $O(n\sqrt{k \log k})$ time [ALP00].

To give a flavor of this technique, here we present briefly the classic algorithm of Fischer and Paterson [FP74] for matching under Hamming distance. For each symbol c from the alphabet we create a bit-vector $B[c]$ of length m, with bits 1 at positions where c occurs in P, and 0s elsewhere. Having this representation, we calculate convolutions between P and a sliding window of T of width m. Using FFT, this can be done in $O(n \log m)$ worst-case time. The result at each text position is a multiplication of two binary vectors, i.e., the number of matching pairs of bits 1. To make the implementation easier, we can run this in parallel for all σ symbols, and wherever the sum of matches over the alphabet (hence the σ multiplicative factor) is at least $m - k$, there we have a match.

FFT-based algorithms often belong to the asymptotically best solutions known today, that is, are very competitive in the worst case for long patterns, but in practice they may lose even to most naïve (brute-force) techniques, due to large constants involved [FMN06].

Very recently, Fredriksson and Grabowski [FG09b] presented a word-level parallelization technique for FFT-based approximate string matching, illustrating it with algorithms for Hamming distance and k-mismatches (similar improvement for several other problems is possible). In particular, they reduced the complexity of the Amir et al. algorithm [ALP00] for k-mismatches to $O(n + n\sqrt{k/w}\log k)$; other results can be found in the cited work.

2.2.4 Algorithms based on bit-parallelism

The bit-parallel approach, presented already in Chapter 1, proved its usefulness also in many approximate matching problems. One of the first algorithms of this kind was Shift-Add [BYG92], which can be considered an ingenious generalization of the Shift-Or algorithm that works for the exact matching. Shift-Add is designed for the Hamming distance and solves the matching problem in $O(n\lceil m\log(k)/w\rceil)$ worst-case time. We postpone describing Shift-Add in detail until Sect. 2.3.1, where our modifications [GF08] of this algorithm, with better time complexity, will also be presented.

The influential paper by Wu and Manber [WM92b], among many other practical ideas, contained the first bit-parallelization of the NFA automaton of Fig. 2.2. In this example (the picture idea was taken from [Nav01a]), the automaton recognizes the pattern "ROSES" with at most two Levenshtein errors. The grey circles correspond to the states active after reading the text "AROUSED". We assume the Levenshtein distance again, but it can be adapted to some other distance measures in a straightforward way. The Wu–Manber idea was to maintain $k+1$ rows of the automaton on the referenced figure, in one machine word each, if the pattern is short enough. Each machine word $R_i, i = 0 \ldots k$, keeps the states (active/inactive information encoded in one bit) for each pattern prefix, assuming exactly i errors in each prefix. The bit-parallel simulations of the automaton transactions are similar to the ones in Shift-Or (in particular, the mechanism for handling R_0 is exactly Shift-Or), but the formula is longer since the new value for $R_i, i > 0$, must be a superposition ("or" operation) of four components, corresponding to a matching character, an insertion error, a deletion error, and a substitution error. On the overall, the algorithm complexity is $O(k\lceil m/w\rceil n)$ in the worst, but also in the best case. The algorithm is not very fast, according to current standards, but its simple and "decomposable" structure allows for

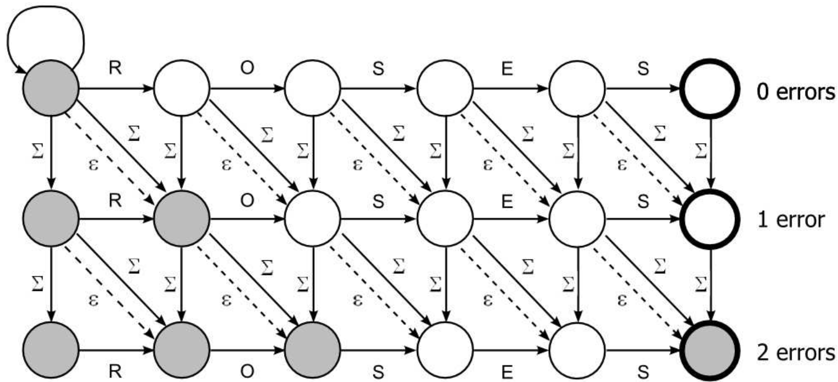

Figure 2.2: An NFA for recognizing the pattern "ROSES" with at most two Levenshtein errors

great flexibility, e.g., to handle classes of characters and regular expressions quite easily. This algorithm is the main building brick of the well-known software package *agrep* from the same authors [WM92a].

Another technique to parallelize the NFA was presented by Baeza-Yates and Navarro in 1996 [BYN96, BYN99], where the time complexity was improved to $O(\lceil km/w \rceil n)$, thanks to using parallelization over the diagonals, not rows, of the automaton. The result seems a mediocre improvement over its predecessor but note that for an important practical case of both small m and small k (e.g., $m = 10$, $k = 2$), this algorithm works in linear time while the Wu–Manber algorithm had to update three machine words for $k = 2$. Nevertheless, a more radical improvement was achieved by Myers in 1998 [Mye98], in an algorithm parallelizing the computation of the DP matrix. This algorithm, with $O(\lceil m/w \rceil n)$ time complexity, still belongs to the fastest ones in practice. Unfortunately, Myers' algorithm is quite complicated, even in the simplified variant by Hyyrö [Hyy01].

2.2.5 The filtering approach

The concept of filtering is extremely simple and natural: use a fast algorithm to discard possibly many text areas which cannot match, and use another algorithm to verify potential matches. The verifier may even use brute-force.

Filters care only for the average case. In their range of applicability, however, they belong to the best choices: for example, for Levenshtein and Hamming distance the average-optimal algorithms [CM94, FN04] belong to filters, and their average time complexity is $O((k + \log_\sigma(m))n/m)$ (with

some limitation on the error level k/m). Also in practice filtering algorithms belong to the fastest, if only the error level is not too high.

The collection of known filtering algorithms for approximate matching is very large and we resign from presenting them; the curious reader is referred to Navarro's survey [Nav01a] and also to a newer work [FN04].

2.3 A new technique for bit-parallel algorithms with counters

In this section we present our technique for an "economical" use of bits in bit-parallel algorithms working with counters. Our considerations will be carried on the example of the well-known Shift-Add algorithm [BYG92] for matching under Hamming distance, and later on we will show how the presented idea can be applied also to some other problems and corresponding algorithms. Most of the content of this section has been presented in [GF08].

As we know from Section 2.2, there are several algorithms with $o(nm)$ worst-case search time for Hamming distance. The most practical algorithm from this group seems to be Shift-Add, based on bit-parallelism.

We propose two new variants of the Shift-Add algorithm, improving its $O(n\lceil m\log(k)/w\rceil)$ worst-case time first to $O(n\lceil m\log\log(k)/w\rceil)$, and then to $O(n\lceil m/w\rceil)$. Note that the better result matches the time of the best known algorithm for searching under edit distance [Mye99], obtaining optimal parallelization.

2.3.1 Shift-Add algorithm

Shift-Add reserves a *counter* of $\ell = \lceil\log_2(k+1)\rceil + 1$ bits for each pattern character in a bit-vector D of length $m\ell$ bits. This bit-vector denotes the search state: the ith counter tells the number of mismatches for the pattern prefix $p_0 \ldots p_i$ for some text position j. If the $(m-1)$th counter is at most k at any time, i.e., $D_{[m-1]\ell} \leq k$, then we know that the pattern occurs with at most k mismatches in the current text position j (more precisely, j is the end position of the matching area).

The preprocessing algorithm builds an array B of bit-vectors. More precisely, we set $B[c]_{[i]\ell} = 0$ iff $p_i = c$, and 1 otherwise. Then we can accumulate the mismatches as

$$D \leftarrow (D << \ell) + B[t_j].$$

It means that the shift operation moves all counters at position i to position $i+1$, and effectively clears the counter at position 0. Recall that the counter i corresponds to the number of mismatches for a pattern prefix $p_0 \ldots p_i$. The $+\ B[t_j]$ operation then adds 0 or 1 to each counter, depending on whether the corresponding pattern characters match t_j.

If $D_{[m-1]\ell} \leq k$, the pattern matches with at most k mismatches. Note that since the pattern length is m, the number of mismatches can even be m, but we have allocated only $\ell = O(\log k)$ bits for the counters. This means that the counters can overflow. The solution is to store the highest bits of the fields in a separate computer word o, and keep the corresponding bits cleared in D:

$$D \leftarrow (D << \ell) + B[t_j]$$
$$o \leftarrow (o << \ell) \mid (D \ \&\ om)$$
$$D \leftarrow D \ \&\ {\sim}om$$

The bit mask om has bit one in the highest bit position of each ℓ-bit field, and zeros elsewhere. Note that if o has bit one in some field, the corresponding counter has reached at least value $k+1$, and hence clearing this bit from D does not cause any problems. There is an occurrence of the pattern whenever

$$(D + o) \ \&\ mm \quad < \quad (k+1) << ((m-1)\ell),$$

i.e. when the highest field is less than $k+1$. The bit mask mm selects the $(m-1)$th field. Shift-Add clearly works in $O(n)$ time, if $m(\lceil \log_2(k+1) \rceil + 1) \leq w$. Otherwise, $\lceil m\ell/w \rceil$ computer words have to be allocated for the counters, and the time becomes $O(n\lceil m \log(k)/w \rceil)$ in the worst-case. Note that on average the time is better, since only the words that are "active", i.e. the words that have at least one counter with a value at most k have to be updated. This implies from the fact that the counters can only increase.

Note that the seemingly harder problem, string matching under edit distance, can be solved more efficiently with bit-parallelism, in $O(n\lceil m/w \rceil)$ worst-case time [Mye99]. Unfortunately, this algorithm cannot be modified for the matching model in question, as it relies on the fact that adjacent cells in the dynamic programming matrix can differ only by -1, 0 or $+1$, which is not the case under Hamming distance.

2.3.2 Counter-splitting

In this section we show how the number of bits for Shift-Add can be reduced. The idea is simple. We use two levels of counters. The top level is as in

plain Shift-Add, i.e. we use $\ell = O(\log k)$ bits. For the second level we use only $\ell' = \log_2(\log_2(k+1)+1)$ bits. The basic idea is then to use a bit-vector D' of $m\ell'$ bits, and accumulate the mismatches as before. However, these counters may overflow every $2^{\ell'}$ steps. We therefore add D' to D at every $2^{\ell'} - 1$ steps, and clear the counters in D'. The result is that updating D' takes only $O(\lceil m \log\log(k)/w \rceil)$ worst-case time per text character, and updating D takes only $O(\lceil m \log(k)/w \rceil / 2^{\ell'}) = O(m/w)$ amortized worst-case time. The total time is then dominated by computing the D' vectors, leading to $O(n \lceil m \log\log(k)/w \rceil)$ total time. It is easy to notice that no $\ell' = o(\log\log k)$ can improve the overall complexity.

Note that as we add now values of at most $2^{\ell'} - 1$ to the counters in the D vector, instead of just 0 or 1 (as for D'), we must allocate $\ell = \left\lceil \log_2(k + 2^{\ell'}) \right\rceil + 1$ bits for them. However, this does not change anything in asymptotic terms, i.e. $\ell \leq \lceil \log_2(2k) \rceil + 1 = \lceil \log_2 k \rceil + 2 = O(\log k)$ bits.

Now adding the two sets of counters can be done without causing an overflow, but the problem is how to add them in parallel. The difficulty is that the counters have different numbers of bits, and hence are unaligned. The vector D' must therefore be expanded so that we insert $\ell - \ell'$ zero bits between all counters prior to the addition, i.e. we must obtain a bit-vector x, so that

$$x_{[i]\ell} = D'_{[i]\ell'}.$$

Then we need to effectively add the counters in D and D' as $D + x$. Note that if the counters in D' consume a whole machine word, i.e., w bits, then the counters in D need $O(w \log(k)/\log\log k)$ bits, and then the simplest solution for computing x is to use look-up tables to do that conversion in $O(\log(k)/\log\log k)$ time. It may seem it requires $O(2^w)$ space and even more preprocessing time. In the RAM model of computation it is assumed that $w = O(\log_2 n)$, where n, roughly speaking, corresponds to the length of the longest addressable text, and hence we may use e.g. $\log_2(n)/2$ bit words for indexing the table, and construct the final answer from two pieces. The space is then just $2^{\log_2(n)/2} = \sqrt{n}$ words, which is negligible (in the asymptotic sense) compared to the length of the text. In practice we may use e.g. $w/2$ or $w/4$ bit indexes, depending on w. Clearly, transforming (*expanding*) the bit-vector D' cannot be done in constant time, but fortunately it is performed only every $O(\log k)$ steps, hence in the amortized sense the whole operation is constant-time per input character. For long patterns the cost becomes $O(\lceil m \log\log(k)/w \rceil)$ per character. Observe that this solution assumes that $w = O(\log n)$. In Sect. 2.3.3 we show another method not based on precomputation, and hence it removes the above assumption.

Note that we cannot shift the vector D at each step as this would cost $O(\lceil m\ell/w \rceil)$ time. Instead, we shift it only each $2^{\ell'} - 1$ steps in one shot prior to adding the two counter sets:

$$D \leftarrow D << (2^{\ell'} - 1)\ell.$$

As in plain Shift-Add, we must take care not to overflow the counters. The overflow bit is therefore cleared before the addition, and restored afterwards if it was set:

$$D \leftarrow ((D \ \& \ {\sim}om) + x) \mid (D \ \& \ om).$$

The final obstacle is the detection of the occurrences, but this is easy to do. At each step j, we just add $D'_{[m-1]\ell'}$ and $D_{[m-1-j \bmod \ell']\ell}$. This constitutes the true sum of mismatches for the whole pattern at text position j. If the sum is at most k, we report an occurrence. Note that this takes only constant time since we only add up two counters, one from each of the two vectors (the whole counter sets are added only each $2^{\ell'} - 1$ steps). Note that as the vector D is not shifted at each step, we simulate the shift by selecting the $(m - 1 - j \bmod \ell')$th field when detecting the possible occurrences. We finally note that the parameter ℓ' can be adjusted to be a power of 2, without any change in the algorithm complexity, and thus the modulo operation needs no division or multiplication.

Summing up, we have an $O(n\lceil m \log\log(k)/w \rceil)$ worst-case time algorithm for string matching under Hamming distance. Alg. 11 shows the pseudocode.

2.3.3 Expanding and contracting counters

A weakness of the shown idea is its dependence on precomputed tables, which limits the machine word to $O(\log n)$ bits, with a constant less than 1 in practice. This goes against the trend in modern hardware, where architectures with wide CPU registers appear more and more often. For example, Intel Pentium4 CPU contains a special set of 128-bit registers, which could not be fully used in the variant shown in the previous section, because of the exponential preprocessing costs.

Now we show how to compute $x_{[i]\ell} = D'_{[i]\ell'}$ parallely without precomputed tables, thus removing the exposed flaw. This requires that the counters are arranged in a different way. Let us call the new vector D^*, replacing D'. The counters in D^* are grouped in *blocks* of ℓ bits. Hence each block contains $c = \lfloor \ell/\ell' \rfloor = O(\log(k)/\log\log k)$ counters, and the total number of

Alg. 11 Shift-Add-Log-Log-k(T, n, P, m, k).

```
1   ℓ' ← ⌈log₂(log₂(k + 1) + 1)⌉
2   ℓ ← ⌈log₂(k + 1 << ℓ')⌉ + 1
3   iv ← 0
4   for i ← 0 to m − 1 do iv ← iv | (1 << (iℓ'))
5   for i ← 0 to σ − 1 do B[i] ← iv
6   for i ← 0 to m − 1 do B[p_i] ← iv ^ (1 << (iℓ'))
7   om ← 0
8   for i ← 0 to m − 1 do om ← om | 1 << ((i + 1)ℓ − 1)
9   D' ← 0; D ← om; j ← 0
10  while j < n do
11      for i ← 1 to 2^ℓ' − 1 do
12          D' ← (D' << ℓ') + B[t_j]
13          if D'_[m−1] + D_[m−1−j mod (2^ℓ'−1)] ≤ k then report match
14          j ← j + 1
15      x ← Expand(D')
16      D ← D << (2^ℓ' − 1)ℓ
17      D ← ((D & ∼om) + x) | (D & om)
18      D' ← 0
```

blocks is $b = \lceil m/c \rceil$. Note that the total number of bits is still $O(m \log\log k)$. The possible $\ell - \lfloor \ell/\ell' \rfloor \ell'$ extra bits within each block are unused. The counters have a different arrangement now. Let us denote the ith ℓ'-bit counter in the jth block of D^* as $D^*_{[i,j]\ell'}$. Then $D^*_{[i/b, i \bmod b]\ell'} = D'_{[i]\ell'}$. For example, if $c = 4$ and $b = 5$ we use an arrangement

$$(0, 5, 10, 15)(1, 6, 11, 16)(2, 7, 12, 17)(3, 8, 13, 18)(4, 9, 14, 19),$$

where the parentheses represent blocks of ℓ bits, and the numbers denote the permutation of the counters $0 \ldots 19$ within the blocks. The shift is no longer by ℓ' bits, but by ℓ bits, hence we shift a whole block at a time. The last block needs a special treatment; the last counter of the last block is dropped out, and the rest is shifted by ℓ' bits and the block is moved to the first block position (cleared by the shift operation). Hence we obtain

$$(-1, 4, 9, 14)(0, 5, 10, 15)(1, 6, 11, 16)(2, 7, 12, 17)(3, 8, 13, 18),$$

where -1 denotes a new counter introduced by the shift. This does not pose any problems to the rest of the algorithm as long as we permute the preprocessed $B[\cdot]$ vectors in the same way.

The benefit of this permutation is that now D^* is easy to expand without precomputation. Note that the first counter of each block is aligned to start at an ℓ-bit boundary, and they are already in the order we use for the D

vector. Hence we need only to mask the rest of the counters (bit-parallely) away, and $1/c$ of the work is done. The counters $2 \ldots c$ are obtained in the same way, we shift the D' vector to align each of the counters in the block in turn to the first position, and mask the rest out. The final answer is then a concatenation of the resulting c vectors. The cost of the shifting and masking is $O(1)$ per counter position (assuming that D' fits to w bits), and we have c counter positions for the blocks. Hence the total cost is $O(\log k/\log\log k)$, including the concatenation. But again, this happens only every $O(\log k)$ steps, so the amortized cost is $o(1)$, not affecting the total time.

In some applications of our technique, we will need also the inverse of the expand function, i.e. we need to compute $x'_{[i]\ell'} \leftarrow y_{[i]\ell}$ for all i efficiently. We call this function *Contract*$(\cdot)$. Note that this is not possible in general, as the value of $y_{[i]\ell}$ may not fit into ℓ' bits. However, in the context we are going to use this, we have a guarantee that ℓ' bits will suffice. In general, we may assume that we want to compute $x'_{[i]\ell'} \leftarrow y_{[i]\ell}$ & $1^{\ell'}$ for all i. It is easy to see that this can be computed in parallel just by inverting and doing in opposite order all the steps required for expanding the counters (or by using precomputed tables, which makes the task trivial). Hence the time bound remains also the same, including the amortized $o(1)$ time, as we will be doing this operation only in the companion of *Expand*.

2.3.4 Matryoshka counters

The above scheme can be improved by using more counter levels. We call these *Matryoshka counters*, to reflect their nested nature. Assume that we use $\ell_1 = 2$ bits in the first level, so this requires $O(\lceil m/w \rceil)$ time per text character. The second level uses $\ell_2 = \ell_1 + 1 = 3$ bits, and so on, in general the level i has $\ell_{i-1} + 1 = i + 1$ bits. The ith level is touched every $2^{\ell_{i-1}} - 1$ steps, and costs $O(\lceil \ell_i m/w \rceil / (2^{\ell_{i-1}} - 1))$ amortized time. The total time is then of the form

$$O\left(\sum_{i}^{\log_2 m} \frac{\lceil \ell_i m/w \rceil}{2^{\ell_{i-1}} - 1} \right) = O\left((m/w) \sum_{i}^{\infty} \frac{i+1}{2^i - 1} \right) = O(m/w).$$

Hence, the total amortized worst-case time is $O(n\lceil m/w \rceil)$.

However, the method we used for detecting the occurrences is too costly in this case (i.e. $O(L)$ per text position, where L is the number of counter levels). Our solution is to delay the occurrence reporting. The second highest level counters have $\lfloor \log_2 k \rfloor + O(1)$ bits, so the last level is touched every $I = O(k)$ steps of the algorithm, and at precisely these steps we can detect

the occurrence in $O(1)$ time, examining only one counter. But at this time $I-1$ highest counters positions have been lost. We therefore add $I-1$ new (high) counter positions for each level, which are shifted and added together as any other counter, but do not accumulate any errors through the $B[\cdot]$ vectors. In other words, these $I-1$ new counters are just to preserve the accumulated error values without shifting them out. Hence at every Ith step we detect the possible I occurrences for the last I text positions using the "overflow" counter positions. This costs only $O(1)$ time per text character, even if implemented naïvely. Finally note that the overflow counters increase the vector lengths only by a constant factor even in the worst case ($k = m-1$), and hence the time bound is preserved.

Expanding the Matryoshka counters without precomputation is possible as well, using a method similar to that of Sect. 2.3.3. This time we use a constant number of bits per counter in the first level, and double it in each next level, i.e. we set $\ell_{i+1} = 2\ell_i$. The initial counter arrangement is simply the identity permutation, and in the second level we use a prebuilt mask to separate the odd and even counter positions, and shift the counters in odd positions to the end of the row of counters (possibly several machine words from the original position). The process for the higher counter levels is similar, but the counters are wider. The invariant of the counter shifting is that the set of counters is divided in three groups only (one of them having always a single counter, the last one), and the counters of each group are shifted by the same number of bits, i.e., the overall work per input machine word is constant. The summation is analogous to the one above, with a new formula for ℓ_i, which again leads to $O(n\lceil m/w \rceil)$ time.

2.4 Average-Optimal Shift-Add for short patterns

In the previous section we showed how to improve the worst case of the Shift-Add algorithm. Now we show how to apply our technique of skipping text characters, presented in Chapter 1, basically for exact string matching, to Shift-Add. This is not difficult since Shift-Add resembles Shift-Or, for which we originally devised that idea. The pattern is again splitted to q partitions. If some of our q patterns occur with at most k mismatches, then we verify if the whole pattern occurs with at most k mismatches. Note that this is different from most of the other pattern partitioning based approaches, that partition the pattern into q pieces, and then search the pieces with $\lfloor k/q \rfloor$ errors. This latter approach leads to $O(nk \log_\sigma(m)/m)$ average time in general, and works for $k = O(m/\log_\sigma m)$. This time is not optimal,

Alg. 12 Average-Optimal-Shift-Add(T, n, P, m, q, k).

```
1   ℓ ← ⌈log_2(k + 1)⌉ + 1
2   iv ← 0
3   for i ← 0 to m − 1 do iv ← iv | (1 << (iℓ))
4   for i ← 0 to σ − 1 do B[i] ← iv
5   iv ← (1 << (ℓ − 1)) − (k + 1)
6   h ← 0; mm ← 0; hm ← 0; om ← 0
7   for j ← 0 to q − 1 do
8       for i ← 0 to ⌊m/q⌋ − 1 do
9           B[P[iq + j]] ← B[P[iq + j]] ^ (1 << h)
10          h ← h + ℓ
11      hm ← hm | (((1 << ℓ) − 1) << (h − ℓ))
12      mm ← mm | (1 << (h − 1))
13      iv ← iv | (iv << h)
14  for i ← 0 to σ − 1 do B[i] ← B[i] + iv
15  for i ← 0 to ⌊m/q⌋q − 1 do om ← om | (1 << (((i + 1)ℓ) − 1))
16  D ← 0; o ← om; i ← 0
17  do
18      D ← (D << ℓ) + B[T[i]]
19      o ← (o << ℓ) | (D & om)
20      D ← D & ~hm & ~om
21      if (o & mm) ≠ mm then Verify(T, i, n, P, m, q, k, o)
22      o ← o & ~hm
23      i ← i + q
24  while i < n
```

whereas our approach leads to $O(n(k + \log_\sigma m)/m)$ optimal average time.

Adapting the Shift-Add algorithm to multiple patterns requires some modifications on the preprocessing and searching algorithms. The problem is how to detect the matches of several patterns in parallel. This is solved by initializing the counters to $2^{\ell-1} - (k + 1)$, instead of to zero. This trick has been used before, e.g. in [CIN+05]. This ensures that the overflow bit is activated immediately when the counter reaches a value $k + 1$, and is therefore easy to detect for all patterns in parallel. This could be implemented explicitly, by setting the first field in D of each pattern to this value after the shift operation. Instead, we add $2^{\ell-1} - (k + 1)$ to all fields of the $B[c]$ vectors that correspond to the first character of each of the patterns. This ensures that the counters in D get correctly initialized, assuming the first counters of each pattern were zero before the addition. This zeroing is done explicitly with a bit mask. Alg. 12 gives the code.

The probability of a match of our $\lfloor m/q \rfloor$ length pattern piece with at most k mismatches is exponentially decreasing if $k/\lfloor m/q \rfloor < 1 - e/\sigma$ [Nav01a]. For our $q = O(m/\log_\sigma m)$, this becomes $k/\log_\sigma m < 1 - e/\sigma$.

This condition ensures that the probability of a verification is $\gamma^{\lfloor m/q \rfloor}$, where $\gamma < 1$, and hence the number of verifications is negligible, and the total average time is $O(n \log_\sigma m/m)$, which is again optimal. This is good only for reasonably large alphabets and very small k, at most $O(\log_\sigma m)$. For larger k one can choose $q = O(m/(k + \log_\sigma m))$, to get again an optimal $O(n(k + \log_\sigma m)/m)$ average time. Linear worst-case time (for short patterns) can be obtained in similar way as in the case of Shift-Or. For long patterns all the bounds must be multiplied by $O(m \log_2(k)/w)$.

2.5 Other applications of Matryoshka counters

There exist many algorithms based on techniques similar to Shift-Add. In the subsections below we present our solutions [FG09c] to the problems of (δ, γ)-matching, (δ, k)-matching, intrusion detection and episode matching. Additionally, in [FG09c] we presented an algorithm for (δ, α)-matching, reducing the worst-case complexity of our previous algorithm [FG06c] from $O(n\lceil m \log(\alpha)/w \rceil)$ to $O(n\lceil m \log\log(\alpha)/w \rceil)$. Still, the algorithm is relatively complex and requires understanding of its predecessor, which will be described in Sect. 3.10. From those reasons, we omit the details here.

2.5.1 (δ, γ)-matching and (δ, k)-matching

Let us now consider (δ, γ)-matching [CCI+99, CIN+05]. In this problem, the pattern matches a text area if $|p_i - t_{j-m+1+i}| \leq \delta$ for all $i = 0 \ldots m-1$, and additionally the sum of all those errors does not exceed γ. Note that a mismatch at a particular position can be handled by adding to the accumulated differences the value $\gamma + 1$. The bit-parallel solution for this problem is very similar to Shift-Add, and runs in $O(n\lceil m \log(\gamma)/w \rceil)$ worst-case time [CIN+05]. The algorithm reserves only $\ell = O(\log \gamma)$ bits per counter, and the B table is preprocessed using the δ-condition. The rest of the algorithm mimics Shift-Add, just taking care not to cause counter overflows in a similar manner as before.

Now we present our Matryoshka solution for this problem. In the following we assume $w = O(\log n)$, but generalization to any w and using the counter arrangement from Sect. 2.3.3 is possible too. Consider first the two-level variant. We will handle the mismatches (i.e. differences $|p_i - t_j| > \delta$) separately, and hence use two preprocessed tables and state vectors to represent the search state. The small counters again have ℓ' bits, the actual value being fixed later. Let B^δ be the table for the absolute differences, i.e. $B^\delta[c]_{[i]\ell'} = |c - p_i|$ if $|c - p_i| \leq \delta$, and 0 otherwise. Likewise, the

mismatches are recorded using table B^γ as $B^\gamma[c]_{[i]\ell'} = 0$ if $|c - p_i| \leq \delta$, and 1 otherwise. The differences can now be accumulated as in Shift-Add: $D^\delta \leftarrow (D^\delta << \ell') + B^\delta[t_j]$. In the case of mismatches, the added value is 0, so we record the mismatches separately: $D^\gamma \leftarrow (D^\gamma << \ell') \mid B^\gamma[t_j]$. Note that the shift operation automatically initializes the first field of the D^γ vector to 0. Also observe that we could just use $\ell' = 1$ (for the mismatch case), but the total complexity would not improve.

For the top level we use only one state vector, D. The mismatches and accumulated sums exceeding γ are represented with a value $\gamma + 1$ (or greater). However, as the value of the bottom level counter can be $2^{\ell'} - 1$, we use $\ell = \left\lceil \log_2(\gamma + 2^{\ell'}) \right\rceil + 1$. The values in D^δ are added to D as in the case of plain Shift-Add, again expanding the D^δ counters first. The D^γ vector is expanded as well, and then shifted to left by $\ell - 1$ positions, resulting in counter values of at least $\gamma + 1$. This is added to D as well. The overflows are handled precisely as in plain Shift-Add.

It should be clear that the above approach works correctly. Let us now estimate the complexity of the scheme. The bottom level counters may overflow at every $2^{\ell'}/\delta$ steps, and hence the overall total cost is of the form

$$O\left(n\lceil m\ell'/w\rceil + \frac{n}{2^{\ell'}/\delta}\left\lceil m\log(\gamma + 2^{\ell'})/w\right\rceil\right), \tag{2.1}$$

where the first term comes from the bottom level counters, and the second term from the top level counters. The bottom level counters should have at least $\ell' = \lceil \log(\delta + 1)\rceil$ bits, since the accumulated values can reach δ. Let us consider values of the form $\ell' = c\log_2(\delta)$, for some $c \geq 1$. Note that we require $c\log_2(\delta) \leq \log_2(\gamma)$, i.e. $c \leq \log_2(\gamma)/\log_2(\delta)$. The maximum value of a bottom level counter is now $2^{c\log_2(\delta)} = \delta^c$, and the complexity becomes

$$O\left(n\lceil mc\log_2(\delta)/w\rceil + \frac{n}{\delta^{c-1}}\lceil m\log(\gamma + \delta^c)/w\rceil\right), \tag{2.2}$$

which is optimized for

$$c = \log_\delta\left(\frac{\log_2(\gamma + \delta^c)}{\log_2(\delta^c)}\right) + 1 = \frac{\log_2\left(\frac{\log_2(\gamma+\delta^c)}{\log_2(\delta^c)}\right)}{\log_2(\delta)} + 1. \tag{2.3}$$

Taking that $\delta^c = O(\gamma)$, and that the second term of the complexity decreases when c increases, the total complexity can be (somewhat pessimistically) upper-bounded by

$$O(n\lceil m(\log_2\log(\gamma) + \log_2(\delta))/w\rceil). \tag{2.4}$$

This can be improved by using more levels, and reserving only $O(i\log(\delta))$ bits for a level i in the hierarchy of counters. The time then becomes

$$O\left(\sum_i \frac{\lceil \ell_i m/w\rceil}{2^{\ell_{i-1}}/\delta}\right) = O\left(\frac{m\log(\delta)}{w}\sum_i^{\infty}\frac{\delta(i+1)}{\delta^i}\right) = O\left(\frac{m\log(\delta)}{w}\right) \quad (2.5)$$

per text character, and $O(n\lceil m\log(\delta)/w\rceil)$ in total.

Another problem example is δ-matching under the Hamming distance ((δ,k)-matching). A trivial solution is to modify the Shift-Add algorithm so that the array B is preprocessed with respect to δ-matching of characters. In this way, $O(n\lceil m\log(k)/w\rceil)$ worst-case time is achieved. Just as trivially, we can apply our technique to improve the time complexity to $O(n\lceil m/w\rceil)$. Note that in the RAM model of computation this is $O(nm/\log n)$.

We note that if the pattern is long and the alphabet size, σ, is small, a practical alternative can be a simple adaptation of the classic FFT-based algorithm of Fischer and Paterson [FP74] for matching under Hamming distance. To this end, the bit-vectors used in the FP algorithm should respect the δ match relaxation, but only for one of the two sequences, e.g., the pattern. The overall time complexity is like in the original algorithm, $O(n\sigma\log m)$ in the worst case.

2.5.2 Intrusion detection and episode matching

A close relative to α-matching is searching allowing k insertions of symbols into the pattern. In other words, we want to find all minimal length text substrings t, such that $id(P,t) \le k$, where $id(P,t)$ is the minimum number of symbols inserted to P, to convert it to t. It follows that if P is a subsequence of t, then $id(P,t) = |t| - |P| = |t| - m$, and ∞ otherwise, and that $m \le |t| \le m+k$ if t matches P with at most k insertions. This has important applications in intrusion detection [KNM03]. The problem can be solved using dynamic programming [KNM03]. We define a vector C of counters: $C_i = id(p_{0\ldots i}, t_{h\ldots j})$, for $i = 0\ldots m$ and some h. The initial values are $C_0 = 0$, and $C_{i>0} = \infty$. (The value ∞ can be represented as any value $> k$ in practical implementation.) Therefore, the goal is to report all j such that $C_m \le k$. The computation of the new values of C, given the old values C^o, is based on the following recurrence:

$$C_i = C^o_{i-1}, \text{if } p_i = t_j, \text{and } C^o_i + 1 \text{ otherwise.} \quad (2.6)$$

The obvious implementation of the recurrence leads to $O(nm)$ worst-case time. It was then shown by Kuri et al. [KNM03] how to compute C_i using

bit-parallelism, which resulted in an $O(n\lceil m \log(k)/w \rceil)$ worst-case time algorithm. We briefly present the Kuri et al. algorithm as this is the starting point of our solution.

The C_i counters are packed into machine words. Only error counts up to $k+1$ are interesting, so any C_i value above k could be replaced by $k+1$, and a similar effect is achieved using overflow bits, a technique known from Shift-Add. In this way, the counters occupy $\ell = \lceil \log_2(k+1) \rceil$ bits each. A table B storing σ bit-vectors of length $m(\ell+1)$ is built in the preprocessing. Each $(\ell+1)$-bit field of $B[c]$ is set to 01^ℓ if $c = p_i$, and $0^{\ell+1}$ otherwise. Note that the highest bit in each field is 0. Also the state vector D has $m(\ell+1)$ bits, initialized to 0s. If the counters could use $O(\log k)$ bits, the update formula (invoked once per text character) could be simply

$$D^{\text{new}} = (B[t_j] \ \& \ (D << (\ell+1))) \mid (\sim B[t_j] \ \& \ (D + (0^\ell 1)^m)), \qquad (2.7)$$

but the real algorithm is somewhat more complicated (we omit details for lack of space). Let us only note for the formula above that each field of $B[t_j]$ selects between $(D << (\ell+1))$ operation, which corresponds to $C_i \leftarrow C^o_{i-1}$ assignment in the plain dynamic programming formula, and $(D + (0^\ell 1)^m)$, which replaces $C_i \leftarrow C^o_i + 1$ assignment.

It might seem that nested counters could be used for this algorithm just as easily as with Shift-Add, but there is actually a new problem. As seen in Eq. 2.6, the new values C_i depend on the old values of C in less "predictable" way than it was in Shift-Add. In Shift-Add the counter values are simply shifted left (i.e., to the next position) with each text character (and then their counts possibly increased by 1), while here they depend on a condition. Let us assume a two-level counter scheme. The manipulations on counters should be done both in the bottom level and the top level. The top level updates should be done infrequently, and here is where the problem lies, as it seems difficult to delay such operations and then perform a bulk update in constant time. We found a compromise solution though.

Our algorithm gives an improvement over the Kuri et al. algorithm if $\log k = \omega(\log w)$. This may seem quite restrictive but for the intrusion detection problem large values of k (exceeding m) are quite typical. The basic observation is that during the inner loop the *set of distinct values* in the top level counters is never extended, as the counter values are simply copied from one to another ($C_i \leftarrow C_{i-1}$ operations). So, we do not need to know their actual values, only we need to distinguish those m counters somehow. To this end, just before the inner loop we label the top level counters. Fortunately, we do not need to give them truly unique labels (which would imply $\log m$ bits per label) but only choose from a smallest

set of labels which prevents from losing identity of any counter during the inner loops. Since the copy operations always involve only adjacent counters, it is enough to assign label $i \bmod 2^{\ell'}$ to a counter at position i. In this way, we need ℓ' bits per label. Now, for every text character also the upper level may change, but all the copy operations on labels are performed in parallel, with $O(m\ell'/w)$ time per character.

When the inner loop is over, we need to get back the true top level counters, to increase their counts with values from the bottom level. This requires remapping with the labels mentioned above, reflecting the actual label arrangement. A brute-force rearrangement takes $O(m)$ time, after which we we can update the top level counters with the respective counts from the bottom level, in $O(m\ell/w)$ time.

The total time spent on bottom level operations is $O(n\lceil m\ell'/w\rceil)$. The total time spent for the top level is $O(n\lceil m\ell'/w\rceil + n/2^{\ell'}(m + \lceil m\ell/w\rceil))$. The sum of the above is optimized for $\ell' = \log w$, which gives $O(n\lceil m \log(w)/w\rceil)$ overall worst-case time complexity.

Finally, assuming that $w = \Theta(\log n)$, we can improve the brute-force counter rearrangement to take just $O(m/w)$ amortized time per text character, by using look-up tables. In this case we can use $\ell' = \Theta(\log\log k)$, and $\ell = \Theta(\log k)$, giving $O(n\lceil m \log\log(k)/\log n\rceil)$ total time.

We note in passing that for the opposite scenario, i.e., for small k, a theoretical $O(nk)$-time algorithm may be of lower complexity. We mean an application of the classic technique of Landau and Vishkin [LV86], where they build a suffix tree with LCA (lowest common ancestor) support over the concatenated sequence $P\#T$ (# is a unique separator), in linear time, and then, for every text character, jump between matching subsequences of P in constant time using LCA queries, hence finding a match or resolving a mismatch in $O(k)$ time per text position. Translating to our problem, after finding each pair of equal substrings, the position in the pattern is shifted by one, while in the text the position is shifted by two, i.e., a single (mismatching) character is skipped.

A similar problem to matching with k-insertions is *episode matching*, which can be stated like that: Find the shortest text substring(s) that contain P as a subsequence. Using our technique, for $k = n - m$, and keeping track of the minimum values, immediately gives $O(n\lceil m \log(w)/w\rceil)$ time complexity as above.

This is not always better than the best known algorithms for this problem [DFG+97], working in $O(nm/\log m)$ and $O(n+s+nm \log\log(s)/\log(s/m))$ time, using $O(s)$ additional space, but our algorithm dominates over the former result if m is small enough, namely if $\log m = o(w/\log w)$, and

either uses less time or less space than the latter algorithm.

2.6 Global similarity measures

Sometimes we are interested to know how similar two whole sequences are, rather than finding approximate occurrences of one of them (the shorter one) in the other. To compare sequences effectively, we need a similarity measure. Perhaps the most widely used global measure of similarity of two sequences is the *longest common subsequence* (LCS) measure, a standard tool in DNA sequence analysis. Other applications of the LCS measure are, e.g., comparisons of two different versions of a program source file (Unix *diff* tool) and plagiarism detection [GB06]. The LCS problem is surveyed in [Apo97, BHR00].

2.6.1 The LCS problem

The longest common subsequence problem is defined as follows. Given two sequences, $A = a_0 \dots a_{m-1}$ and $B = b_0 \dots b_{n-1}$, over an alphabet Σ, find the longest subsequence $\langle a_{i_1}, a_{i_2}, \dots, a_{i_\ell} \rangle$ of A such that $a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \dots, a_{i_\ell} = b_{j_\ell}$. The found sequence may not be unique. Often, only a simplified version of this problem is considered, when one is interested in telling the length of the longest common subsequence rather than the sequence itself. Note that the LCS problem is a dual of the indel distance: $2LCS(A,B) = n + m - id(A,B)$ [WF74]. It is however accepted to speak about LCS when we are interested in finding the global measure of similarity, and the indel distance (being a dissimilarity measure) when a pattern shifts over a (much) longer text.

Despite over 30 years of research, surprisingly little can be said about the worst-case complexity of LCS. In other words, the gap between the proven lower bound and the best worst-case algorithm is huge. It is known that in the very restrictive model of unconstrained alphabet and comparisons with equal/unequal answers only, the lower bound is $\Omega(nm)$ [WC76], which is reached by a trivial DP algorithm. If the input alphabet is fixed, the lower bound improves to $\Omega(\sigma n)$, but if total order between alphabet symbols exists, and $\leq$-comparisons are allowed, then the lower bound improves to $\Omega(n \log n)$ [Hir78a]. The latter assumption is very natural in real-world applications, yet, even for this "easiest" scenario, the best known algorithm for the worst case achieves $O(nm/\log n)$ [MP80], a mild improvement over a plain DP algorithm.

A simple idea proposed in 1977 by Hunt and Szymanski [HS77] has become a milestone in LCS reseach, and the departure point for the theoretically best algorithms for this problem [AG87, EGGI92]. The Hunt–Szymanski (HS) algorithm is essentially based on dynamic programming, but it visits only the matching cells of the matrix, typically a small fraction of the entire set of cells. This kind of selective scan over the DP matrix is called *sparse dynamic programming* (SDP).

Before we present the outline of the algorithm, we need a definition. We will say that a cell (i, j) of the dynamic programming matrix M stores a match of rank k iff $a_i = b_j$ and $\text{LLCS}(a_{1\ldots i}, b_{1\ldots j}) = k$.

Now we can briefly present the HS algorithm. In the preprocessing, lists of successive occurrences of all alphabet symbols in the shorter sequence, A, are gathered. This requires $O(m + \sigma)$ space and time, and enables to move from one matching cell to the next one in constant time. We assume that $\sigma = O(n)$, hence this factor is negligible in practice. For eliminating a little practical issue, the matches in rows can be traversed from right to left, instead of a more natural left-to-right order. During the main stage of processing, another array, T, is also maintained, which stores at position j the leftmost seen-so-far column with a match of rank j. Note that the values in the "occupied" part of T are strictly increasing (the non-written yet cells in the right part of T may be all initiated with a fixed value, e.g., $m+1$). For each visited cell (i, j), we look for the minimum index t such that $T[t] \geq j$. This can be performed with e.g. binary search. We need to distinguish between $T[t] = j$ and $T[t] > j$; in the former case, the current match is irrelevant and we skip to the next one, in the latter case, we update $T[t]$ with the current j. After processing all the matches from M, the maximum achieved rank of a match is the desired LCS length.

Let us denote the number of all matches in M with the symbol R. It is easy to notice that the time complexity of the algorithm depends on how fast we can find, for each of R matches, the proper t to satisfy the aforementioned inequality. The plain binary search immediately leads to $O(n + R \log m)$ time, but since the non-empty range of T never has more than $\ell = \text{LLCS}(A, B)$ elements, it is more precise to express the worst-case complexity with $O(n + R \log \ell)$. This complexity includes the preprocessing cost. Note that in the worst case, i.e., for $R = O(nm)$, this complexity is superquadratic, i.e., even worse than of the plain DP algorithm.

The Hunt–Szymanski concept was an inspiration for a number of subsequent algorithms for LCS calculation. Finding the rank of a match can be performed in a more refined way than with a binary search, in particular, using the van Emde Boas (vEB) dynamic data structure [vEBKZ77], which

Alg. 13 BP-LCS(A, m, B, n).

```
1   for i ← 0 to σ − 1 do PM[i] ← 0^m
2   for i ← 0 to m − 1 do PM[a_i] ← PM[a_i] | 0^(m−i−1) 1 0^i
3   V ← 1^m
4   for j ← 0 to n − 1 do
5       U ← V & PM[b_j]
6       V ← (V + U) | (V − U)
7   r ← 0; V ← ~V
8   while V ≠ 0^m do
9       r ← r + 1; V ← V & (V + 1)
10  return r
```

is applicable if the universe of keys is nicely bounded (roughly speaking, if σ is not much greater than $\max\{n, m\}$).

In our problem, this translates to $O(n + R \log\log m)$ worst-case complexity. The possibility of using the vEB structure was noticed already by Hunt and Szymanski in their original work. There are even better (and more complex) theoretical algorithms [AG87, EGGI92] based on the idea of Hunt–Szymanski, where for example the symbol R is replaced with D, the number of so-called dominant matches ($D \leq r$). The best of them, the algorithm of Eppstein et al. [EGGI92], achieves $O(D \log\log(\min(D, nm/D)))$ worst-case time (plus preprocessing). Note that this complexity can be upper-bounded with nm for any value of D.

A different approach is to use bit-paralellism to compute several cells of the dynamic programming matrix at a time. The key observation is that the LCS values in successive columns (or rows) differ only by 0 or 1. There are three such algorithms [AD86, CIPR00, Hyy04], all working in $O(\lceil m/w \rceil n)$ worst-case time, after $O(\sigma\lceil m/w \rceil + m)$-time and $O(\sigma m)$-space preprocessing. Alternatively, the search time could be $O(\lceil n/w \rceil m)$ (where n is the length of the longer sequence), but practical implementations are column-wise. The fastest of those three algorithms is Hyyrö's one [Hyy04] (Alg. 13), being a simplification of the scheme from [CIPR00]. Note that the main loop, lines 4–6, contains only four arithmetical or logical operations (assignments and the loop end check not counted), less by one than its predecessor.

2.6.2 LCS-related problem variants

One of the oldest problems related to LCS is the longest increasing subsequence (LIS) problem. In LIS, we are given a sequence S of integers (w.l.o.g. we can assume they are unique), and our task is to find the longest strictly increasing subsequence of S. A dynamic programming algorithm

with $O(n \log n)$ complexity is essentially due to Schensted [Sch61], and its optimality in the comparison based model was proved later, by Fredman [Fre75] (more precisely, he proved that $n \log n - n \log \log n + \Theta(n)$ comparisons are necessary and sufficient to compute the LIS of an integer sequence of length n). Actually, the length of LIS can be computed with any general algorithm finding the length of $LCS(A, B)$, where $A = S$, and as sequence B we take Q sorted in ascending order. Hunt and Szymanski [HS77] point out that their LCS algorithm straightforwardly achieves the $O(n \log n)$ complexity for LIS; the variant of their algorithm with the van Emde Boas priority queue even improves this result to $O(n \log \log n)$, which does not contradict the mentioned optimality, since the vEB structure assumes the RAM model of computation.

What is less obvious is that the problem transformation is possible also the other way around, i.e., having an algorithm for solving LIS, we can also solve the LCS problem [Gus97, BHR00]. To this end, it is enough to scan over the matching cells in the DP matrix, row by row, and list their column numbers in reverse direction. The result is a sequence of decreasing runs of integers. Now, running any LIS-finding algorithm over this sequence outputs the y coordinates of the items belonging to an LCS sequence (their corresponding x coordinates may be kept as satellite data, which completes the required algorithm's output).

If S is regarded as a circular buffer, the LIS problem turns into LICS (longest increasing circular subsequence), which, surprisingly perhaps, happens to be much harder. In other words, now the task is to find a longest subsequence of any *rotation* of a given sequence such that each integer of the subsequence is smaller than the integer that follows it. LICS has applications in bioinformatics. For this problem, there is no single algorithm outperforming the others. Tiskin [Tis08] gave a $O(n^{3/2})$-time solution. Deorowicz [Deo09] proposed a hybrid algorithm of time complexity $O(\min(n\ell, n \log n + \ell^3 \log n))$, where ℓ is the length of the output sequence. Very recently, Deorowicz and Grabowski [DG09b] slightly improved the latter algorithm to achieve $O(\min(\ell^3, n\ell) \log \lceil n/\ell^2 \rceil + n \log \ell)$ time, which is better than its predecessor if ℓ is close to its average value, $\Theta(\sqrt{n})$.

Another problem from the same family is to find the slope-constrained longest increasing subsequence (SLIS) [YC08]. The task is to find a maximum-length increasing subsequence of S, $s_{i_1} < s_{i_2} < \cdots < s_{i_k}$ for $i_1 < i_2 < \cdots < i_k$, such that the slope between two consecutive points is no less than the input ratio, i.e., $\frac{s_{i_{r+1}} - s_{i_r}}{i_{r+1} - i_r} \geq m, 1 \leq r < k$. In [YC08] Yang and Chen gave an $O(n \log \ell)$-time algorithm. Based on it and assuming the RAM ma-

chine, Deorowicz and Grabowski [DG09b] modified this algorithm to achieve $O(n \min(\sqrt{\log \ell / \log\log \ell}, \log\log n))$ worst-case time.[1]

Yet another variation is the constrained LCS problem (CLCS) [Tsa03]. In CLCS, the computed LCS must also contain, in order, all characters of a third sequence (the constraint), of length r. The problem motivation comes from bioinformatics: to compute the homology of two biological sequences it may be important to take into account a common specific structure [Tsa03]. Several algorithms of $O(nmr)$ time complexity are known (e.g., [Pen03, AE05] for this problem, and recently there were shown two output-dependent algorithms, with time complexity expressed either in terms of R, the number of matches between sequences A and B, or $\ell = LLCS(A, B)$. Deorowicz [Deo07] achieved $O(r(m\ell + R) + n)$, while the algorithm of Iliopoulos and Rahman [IR07] gets $O(rR \log\log n + n)$. Clearly, those two algorithms are also $O(nmr)$ (or worse) in the worst case. As a side-note, we point out that in the important case of $n = m$, the complexity formula in the Deorowicz algorithm can be shortened to $O(rn\ell)$. Indeed, ℓ is equal to the number of antichains (see, e.g., [Apo97] for a definition) in the DP matrix for calculating $LCS(A, B)$, and hence any antichain cannot contain more than $2n - 1$ cells, we obtain $R \leq (2n - 1)\ell$, which immediately leads to the simplified complexity formula. Of course, in this reasoning, the condition $n = m$ can be replaced by $n = \Theta(m)$. Experiments show that Deorowicz's algorithm, especially if strengthened with a heuristic to reduce the set of cells to compute, is at least a few times faster than all other existing algorithms [DO09].

2.6.3 The LCTS problem, theoretical and practical solutions

One of the LCS variations, introduced relatively recently [LU00], has applications in music information retrieval. An important trait of similar melodic sequences is that they can differ in the key, but humans perceive them as same melodies. More formally, the problem of *longest common transposition-invariant subsequence* (LCTS) that we talk about is to find the length of the longest subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_\ell} \rangle$ of A such that $a_{i_1} = b_{j_1} + t, a_{i_2} = b_{j_2} + t, \ldots a_{i_\ell} = b_{j_\ell} + t$, for some $-\sigma < t < \sigma$. In other words, we look for the length of the longest subsequence of A and B matching according to any *transposition*. This corresponds to a music phrase (melody) shifted to another key, which is perceived by humans as the same melody. The alphabet size in music (MIDI) application is usually 128. As long as this does

[1]This result can be improved to $O(n \log\log \ell)$, which will be presented in the planned journal version of the cited work.

not lead to confusion, we will denote the length of LCS (LCTS) by LLCS (LLCTS). A naïve algorithm for calculating LCTS is to run the dynamic programming algorithm independently for each transposition, which yields $O(nm\sigma)$ time. Almost all existing better solutions belong to one of the two categories: they are bit-parallel or based on sparse dynamic programming.

Adopting any of the bit-parallel LCS-solving algorithms for the LCTS problem is straightforward: it is enough to run the LCS routine for each of the $2\sigma-1$ transpositions separately, achieving $O(n\sigma\lceil m/w\rceil)$ time complexity (not counting a preprocessing stage). Experiments show [Deo06] that this approach is quite practical.

It may seem that the preprocessing time for a bit-parallel LCTS algorithm must be $O(\sigma^2\lceil m/w\rceil + m\sigma)$, i.e., the preprocessing routine for LCS is performed $2\sigma - 1$ times in total, once for each transposition. This, however, can be easily reduced to $O(\sigma\lceil m/w\rceil + m)$ (and similarly the space can be reduced), by merely using LCS preprocessing for the LCTS problem and replacing the symbol b_j in the current column with the symbol $\lambda = b_j - t$ (provided that the resulting symbol is within the alphabet range), where t is the current transposition [DG09a]. This idea was found practical. Less trivially, the preprocessing can be improved even more, to $O(m)$ worst-case time, for the price of increasing the space by a constant factor, using an old array initialization idea [Meh84, Sect. III 8.1]. This technique, however, is unlikely to be practical in this setting, as the number of array accesses in the search phase gets multiplied by 3.

Sparse dynamic programming has been successfully used for the LCTS problem as well [MNU05]. In this setting, the DP matrix is a set union of matches (cells) belonging to different transpositions, and all of them may be visited in a single scan over the matrix, switching between models. Applying the HS technique for LCTS is thus straightforward, with $O(nm \log m)$ worst-case time. Mäkinen et al. [MNU05] showed how this result can be improved to $O(nm \log\log m)$, using the vEB data structure [vEBKZ77]. Grabowski and Navarro [GN04] suggested another idea, which was to run a brute-force computation over small enough subarrays (blocks of $k \times k$ cells), such that cannot contain too many different transpositions, and the scores for the remaining transpositions are only copied into the subarray corners. This gives immediately $O(nm\sqrt{\sigma})$ complexity for $k = \sigma^{1/4}$, and partitioning the subarrays recursively improves the time complexity to $O(nm \log \sigma)$.

Finally, Navarro et al. combined the technique of working on blocks [GN04] with the SDP algorithm from [MNU05], to achieve an algorithm working in $O(nm \log\log \min(m, \sigma))$ time [NGMD05]; whether $O(nm)$ time, for alphabet size $\sigma = O(n^{\epsilon})$, $\epsilon > 0$, is possible for LCTS remains open.

The same result, but in a somewhat simpler algorithm, was obtained by Deorowicz [Deo06]. Also, in the same paper he presents a related variant, with a slightly worse time complexity, namely $O(nm \log \sigma / \log w)$,[2] which however wins in his thorough tests for MIDI and for uniformly random data if the alphabet is large enough (say, 96 or more). Note that this time complexity equals to $O(nm)$ as long as $\sigma = O(\log^{O(1)} n)$, since $w = \Omega(\log n)$.

Another approach was taken by Lemström et al. [LNP05]. Their branch-and-bound algorithm superimposes groups of transpositions, in order to eliminate, in a lucky case, many of them as if only a single transposition were checked. Additionally, a variant strengthened with bit-parallelism was presented. Despite its elegance, this method cannot keep pace in practice with the best BP or SPD algorithms, even in its BP variant [Deo06], and in the worst case the time complexity of the basic variant loses even to the naïve DP algorithm run $O(\sigma)$ times.

Recently, Grabowski and Deorowicz [GD08, DG09a] proposed a practice-oriented hybrid algorithm for LCTS, making use of a simple observation: if the alphabet is small, the BP approach is a clear winner, but for large enough alphabets SDP algorithms start to dominate. More precisely, they used Hyyrö's bit-parallel algorithm [Hyy04] (later called the BP component) for frequent transpositions, and the most practical LCS algorithm by Deorowicz [Deo06] (later called the HS component, since it can be classified as a variant of the Hunt–Szymanski algorithm) for rare ones. The latter algorithm needs $O(nm \log \sigma / \log w)$ time for a single transposition. A quick and rather robust heuristic was used to distinguish between the two groups of transpositions. Thorough experiments on music data ($\sigma = 128$) showed that the hybrid approach overcomes the better of its two components (which was usually the BP algorithm, especially in tests with $w = 32$) by a factor of 1.4 to 2.0 in 32-bit implementations, and 1.1 to 1.7 in 64-bit implementations, where the larger gaps are for longer sequences. On uniformly random data the improvements were smaller, and they rarely exceeded the factor 1.2. The gains were greater for Gaussian distribution of data, and they sometimes exceeded the factor 2.0, even in the 64-bit implementations.

The music data were a concatenation of 7543 music pieces, obtained by extracting the pitch values from MIDI files. The total length is 1 828 089 bytes. Although the pitch values are in the range $0 \ldots 127$, this data is far from random; the six most frequent pitch values occur 915 082 times, which

[2] As long as $w = O(\log^{O(1)} n)$, which is a reasonable assumption. For $w = O(n^{\varepsilon})$, the complexity grows by the factor $\log \log w$, i.e., $O(\log \log n)$, using the algorithm by Brodnik et al. [BMM97]. For a discussion, see [DG09a].

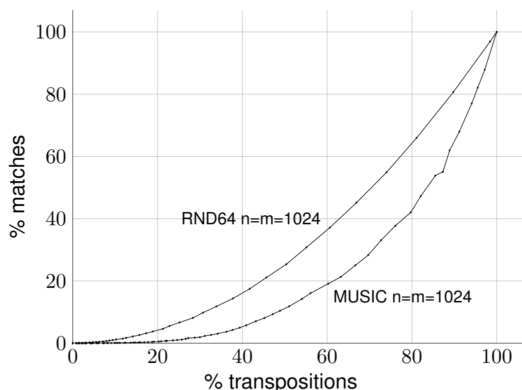

Figure 2.3: LCTS, % matches vs. % transpositions (transpositions sorted by frequency)

is approximately 50% of the whole text, and the total number of different pitch values is just 55. Consequently, the number of possible existing "transpositions", i.e., differences between any pairs of characters from two different excerpts of this file, is much lower than the theoretical maximum of 255. This dataset was previously used in the literature (e.g., [FMN06, Deo06]), for various MIR-oriented problems, including LCTS.

A set of 101 pairs of randomly extracted excerpts from the text was generated. The sequence lengths, n and m, were varied, but always $n = m$. The reported times are the medians over all 101 trials.

Fig. 2.3 demonstrates the relation between the (percentage) amount of most frequent transpositions and the amount of matches covered by them. We can see, for example, for the MIDI data, that the top 20% of the existent transpositions (sorted by frequency) already cover more than half of the matches, while 60% of the existent transpositions are enough to cover over 90% of matches. For the uniformly random data, as expected, the curve is more flat.

Fig. 2.4 shows the overall processing time of our hybrid in the function of the minimal number of matches in transpositions handled by the HS component. Basically the same phenomenon, in the function of varying alphabet size (only for the two random distributions), is also presented in Fig. 2.5. Note that extreme parameters of the thresholds trigger a single component for all transpositions; in most cases, for alphabet size up to 64 or 128 (depending on the dataset and whether the implementation is 32- or 64-bit), the "single best" component is the BP algorithm, while for larger

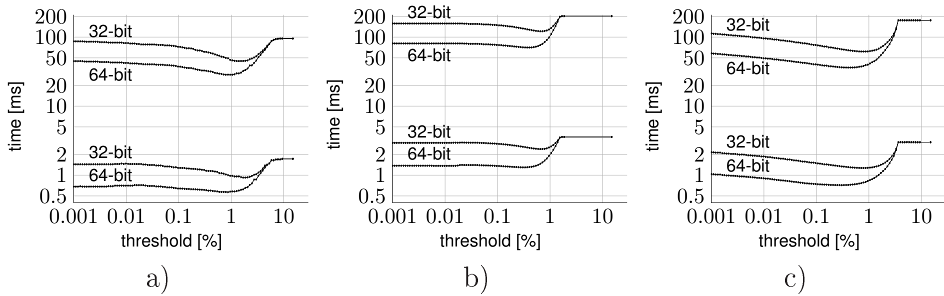

Figure 2.4: LCTS, overall processing time of the hybrid algorithm with varying threshold of the minimal number of matches in transpositions handled by the HS component. a) MUSIC, b) RANDOM-64, c) GAUSS-64. Top pairs of curves for $n = m = 4096$, bottom pairs for $n = m = 512$

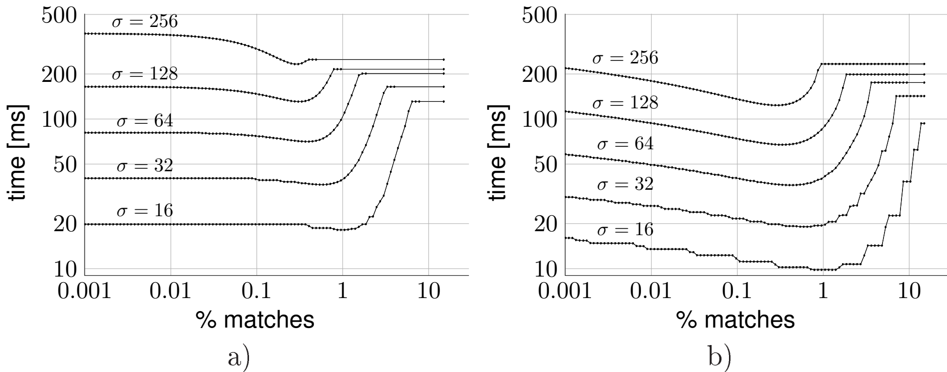

Figure 2.5: LCTS, overall processing time of the hybrid algorithm with varying threshold of the total percentage of matches handled by the HS component (transpositions ordered from the sparsest to the densest). $n = m = 4096$, $\sigma = 16, \ldots, 128$, 64-bit implementation: a) RANDOM, b) GAUSS

alphabets the HS algorithm starts to win. It can be also seen that the Gaussian data are much more sensitive to small changes of the thresholds (in their low ranges), which is not surprising.

It occurs that the best split for the music data allots about 80% matches (from the most frequent transpositions) to the BP algorithm, while the remaining 20% matches are handled by the HS variant. In other words (cf. Fig. 2.3), less than 40% of the most frequent transpositions should be processed by BP. Note also, again for the music data, that the BP component is faster by about 25% (i.e., needs about 20% less time) than the HS component, for $w = 32$, and even about twice, for $w = 64$, if applied exclusively.

Table 2.1: LCTS, MUSIC, 32-bit implementation

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.3157	0.5167	0.2686	0.2768	1.6928	1.3930	45.4	83.8
512	1.4366	1.7222	0.9210	0.9686	1.8621	1.5752	42.1	80.5
1024	5.5727	6.3598	3.0630	3.1441	1.6928	1.4759	40.2	81.5
2048	23.2568	25.5058	13.3131	13.5623	1.5389	1.4969	34.8	78.3
4096	91.7169	95.0223	44.9423	45.6610	1.3990	1.4930	34.7	80.9
8192	381.6627	381.3309	185.5993	188.4778	1.5389	1.5642	33.6	79.9

Table 2.2: LCTS, MUSIC, 64-bit implementation

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.1842	0.5183	0.1842	0.2151	0.0000	0.7371	61.9	92.4
512	0.6811	1.7288	0.5669	0.5868	0.8687	0.7022	61.2	93.1
1024	3.0936	6.3303	2.0560	2.1032	1.1562	0.8202	55.0	91.1
2048	12.2291	25.5285	8.3442	8.6279	0.8687	0.7895	52.1	90.1
4096	47.1900	95.0745	28.4840	29.0666	0.7897	0.7683	48.9	91.8
8192	193.8347	380.7648	112.5319	113.1798	0.8687	0.7952	43.7	91.2

Table 2.3: LCTS, RANDOM-128, 64-bit implementation

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.7996	1.4635	0.7677	0.7771	0.1890	0.1898	75.6	93.6
512	2.7441	4.5429	2.4480	2.4640	0.2287	0.2248	71.3	91.2
1024	11.8452	15.4458	9.6989	9.7430	0.3044	0.2938	62.3	85.3
2048	42.9331	56.6781	34.5518	34.7321	0.3044	0.2914	62.3	85.6
4096	164.4345	215.2981	130.7266	131.3415	0.3044	0.2933	62.3	85.3
8192	729.7012	877.4197	572.9071	576.5836	0.3349	0.3295	57.6	81.8

For the uniformly random data ($\sigma = 64$), those differences are even greater, but drop rapidly with growing alphabet size (on the other hand, for σ smaller than 64, the advantage of the BP algorithm is even more striking).

We also evaluated an "oracular" component selection (column *Best time* in all tables), which sets the lower bound for any selecting heuristic. It can be seen that our idea based on crossing lines is quite stable across all experiments and the loss to the lower bound is usually within 2% of overall time (the worst case is in Table 2.2, $n = m = 256$, where our time was by over 6% longer than with using the oracle).

The detailed timings of the algorithms: HS, BP and our hybrid, are given in the tables, for 32-bit and 64-bit implementations separately. In rows, the problem size changes from $n = m = 256$ to $n = m = 8192$. The right

Table 2.4: LCTS, GAUSS-128, 64-bit implementation

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.5463	1.1105	0.4193	0.4316	0.1420	0.2572	51.1	94.3
512	1.9991	3.7308	1.3030	1.3254	0.1890	0.2741	46.7	94.0
1024	9.2762	13.3282	5.0989	5.1376	0.3349	0.3455	40.3	92.3
2048	35.9229	50.6291	18.0331	18.1210	0.3349	0.3336	38.0	92.6
4096	144.2099	198.6377	67.3530	67.4932	0.3044	0.3349	36.1	92.7
8192	597.2110	778.5215	263.3859	263.5459	0.3684	0.3506	34.3	92.3

half of the columns requires a brief explanation. *Best threshold* specifies the minimum fraction of matches per transposition, for which the BP algorithm starts to work faster than the HS algorithm (if used for this transposition). *Hybrid threshold* conveys similar information; the difference is that here we see the threshold selected by our component selection heuristic. Roughly, the closer those two threshold values are, for a given problem instance, the better the heuristic is expected to work.[3]

The next column, *BP transpositions* is the fraction of transpositions handled by the BP component, using our selection heuristic. Finally, the column *BP match* holds the fractions of matches in the transpositions for which the BP algorithm is triggered.

The speedup factor of the hybrid algorithm over the better of the two components (i.e., BP) on the music data varies from 1.36 ($n = 256$) to 1.97 ($n = 8192$) in the 32-bit implementations, and from 1.11 ($n = 256$) to 1.71 ($n = 8192$) in the 64-bit implementations, so it improves with growing n. Note that we skip non-existent transpositions in the BP algorithm, which boosts its performance on the music data very significantly.

On uniformly random data, the situation is somewhat different. For small to moderate σ (up to 64) the bit-parallel algorithm is much faster than the HS one (note the scale on Fig. 2.5), even in the 32-bit version the difference is 4-fold in case of $\sigma = 16$ and $n = 4096$ (switching to 64 bits makes is almost 7-fold), but the picture changes for $\sigma = 128$ and $\sigma =$

[3]The presented thresholds are medians from individual experiments, but there is a subtle difference between "best times" and "hybrid times". The column *Best time* uses a single threshold (the one for which the median time over 101 runs is minimized), while for the column *Hybrid time* individual thresholds for each test run are used (and the median times for runs with those possibly different thresholds are presented). Although insignificant in practice, this could, in theory, lead to surprising effects, e.g., a shorter *Hybrid time* than *Best time*. The reason for which we chose this presentation methodology was to make it compatible with Figs. 2.4 and 2.5, where a single "best threshold" had to be used.

256. Interestingly, for small alphabets the HS component beats the BP component on some (few) transpositions, so the hybrid, with the threshold selected properly, again appears better than both its components (but with the speedup of about 10% only, at best). For a large enough alphabet ($\sigma = 256$) the BP algorithm usually can win on no transposition, hence the "optimal" hybrid degenerates into the HS component. The border case is $\sigma = 128$ where HS takes the lead but its advantage over BP is quite moderate; in that case the hybrid algorithm is faster than HS by 12–24% (easy to guess, the speedups close to 24% are for the 32-bit implementations).

The HS algorithm does not change its speed when switched from 32 to 64 bits, while the improvement is obvious for the BP algorithm, and the speedup factor varies from about 1.6 to 1.9. The stages of the algorithm which do not gain from longer registers are the preprocessing and the final counting of the set bits in vector V (cf. Alg. 13). The columns *BP transpositions* and *BP match* confirm that after switching from the 32- to 64-bit implementation, the bit-parallel component is selected more often.

2.7 Conclusions

Approximate string matching is an umbrella term encompassing a plethora of matching models and respective applications. In this chapter, we presented only selected approximate matching problems, with applications in, e.g., molecular biology, music information retrieval and natural language processing. A broad class of approximate matching models, those involving gaps between the pattern symbols, are the subject of the next chapter.

Our main contribution to this research area is a novel technique of nested counters in bit-parallel algorithms, which we refer to as Matryoshka counters. This idea easily enables to shave off the $\log m$ factor from the complexity of the classic Shift-Add algorithm for the k-mismatches problem. We have also successfully applied Matryoshka counters for many other problems (which was often not that simple).

Other results of ours include an average-optimal Shift-Add variant for short patterns (based on our technique presented in Chapter 1), word-level parallelization technique for FFT-based algorithms for k-mismatches and Hamming distance and a simple, yet efficient, hybrid algorithm for the longest transposition-invariant common subsequence problem. For the same problem, we collaborated in the design of the best known worst-case optimized algorithm.

Chapter 3

Matching with gaps

One of seemingly underexplored problems with applications in music information retrieval (MIR) and molecular biology (MB) is (δ, α)-matching [CIM$^+$02] and its variations. In this problem, the pattern $p_0 p_1 \ldots p_{m-1}$ is allowed to match a substring of the text $t_0 t_1 \ldots t_{n-1}$ with α-limited gaps, and the respective pairs of matching characters may be different, only if their numerical values do not differ by more than δ. Translating this model into a music (melody seeking) application, we can allow for small distortions of the original melody because the (presumably unskilled) human user may sing or whistle the melody imprecisely. The gaps, on the other hand, allow to skip over ornamenting notes (e.g., arpeggios), which appear especially in classical music. Other assumptions here, that is, monophonic melody and using pitch values only (without note durations), are reasonable in most practical cases. Sometimes, we also want to bound the total sum of distortions over individual characters (notes), hence an extra condition, γ, can also be imposed. In biology, somewhat relaxed version of the α-matching problem is important for protein matching, especially together with allowing for matching classes of characters. Fortunately, in all the new algorithms we are going to present in this chapter, δ-matching can be straightforwardly changed into matching classes of characters, without any penalty in the complexities if the size of the character class is of the order of δ.

We proposed a number of algorithms for (δ, α) and (δ, γ, α)-matching problems, motivated by music information retrieval (MIR) applications. Our algorithms are competitive either in theory, or in practice, or both. For those problems, complex interplays between the preprocessing times and search times, and also average time and worst-case time complexities, have been spotted and analyzed. Our algorithms are based on bit-parallelism, sparse

dynamic programming with cut-off, and automata.

The outline of this chapter is as follows. In the beginning, we introduce formally the (δ, α) and (δ, γ, α)-matching problems. Section 3.2 briefly surveys previous results for those problems. Section 3.3 presents dynamic programming solutions, basic variants with quadratic behavior in every case, and cut-off variant optimized for the worst case. The next two sections introduce two sparse dynamic programming algorithms, a row-wise and a column-wise one, with different interplay between the preprocessing and search complexity. We consider many variations and improvements for those algorithms, e.g., intented to speed up the preprocessing phase. One of those variations is a row-wise SDP algorithm oriented for large α cases. A simple, but surprisingly efficient in practice, offspring of the SDP algorithms is presented in Sects. 3.6 and 3.7. We even show its variant, taking inspiration from the BMH algorithm, with sublinear average time. The next section deals with bit-parallelism. Although the BP algorithms we propose for the (δ, α) and (δ, γ, α)-matching problems are intrinsically similar, there are some extra difficulties concerning the (δ, γ, α) problem. Interestingly, our BP approach appears to be a surprisingly simple solution to an old problem, motivated in computational biology, of matching when negative gaps are allowed. Section 3.10 presents a novel bit-parallel NFA simulation for the (δ, α)-matching problem, which is inspired by the previous algorithm of this kind [NR03, CCF05b], but needs fewer bits, hence is more efficient on long patterns. Section 3.11 discusses other related problems, in particular about the translation from the (δ, γ, α) model (developed for MIR applications) to matching with gaps and character classes, with up to k mismatches, which is more relevant for MB. The last theory section contains non-trivial adaptations of several presented earlier algorithms for (δ, α)-matching for the scenario with transposition invariance. The next two sections contain results of our experiments, for (δ, α) and (δ, γ, α)-matching, respectively. The last section concludes.

The arsenal of novel algorithms and techniques presented in this chapter is quite spacious, and it is not easy to explain all the nuances and particular niches of application for individual algorithms. Let us try, however, to point out the main achievements.

1. We proposed a number of sparse dynamic programming algorithms for the (δ, α) and (δ, γ, α)-matching problems, with different dependencies between the preprocessing and search times, both for the worst and the average case.
2. We presented an $O(\lceil n/w \rceil m)$ search time bit-parallel algorithm for

(δ, α)-matching and a similar variant with $O(\lceil n \log(\gamma)/w \rceil m)$ search time for (δ, γ, α)-matching. Those complexities assume that $w = \Theta(\log n)$. For an unrestricted machine word size, the times should be multiplied by $\log \alpha$. More practical implementations of those algorithms are also equipped with the so-called cut-off technique which lets stop computations for those areas of the dynamic programming matrix in which no pattern prefixes can be prolonged. We use the cut-off idea also for most other algorithms of ours.

3. We proposed several worst-case oriented preprocessing techniques for the bit-parallel algorithms. One of them works in merely $O(n)$ time and can be used for the (δ, γ, α) problem.
4. We gave an $O(n + nm \log(\alpha|\mathcal{M}|/(nm))/\alpha)$ worst-case time algorithm for (δ, α)-matching, suitable for the case of large α. $\mathcal{M}$ denotes the set of matches in the dynamic programming matrix for this problem.
5. Almost all our algorithms can be easily transformed to work in a protein search application, where gaps between any pair of pattern characters can be of different width, and the δ condition is replaced with a more general notion of a character class. Moreover, most of our sparse dynamic programming and bit-parallel techniques work even for negative gaps, a problem variant which seemed hard earlier.
6. We showed that the $O(\lceil m\alpha/w \rceil)$-bit automaton of Navarro and Raffinot [NR03] can be represented more compactly, in $O(\lceil m \log(\alpha)/w \rceil)$ bits, which preserves the linear time complexity for longer patterns than in the original solution.
7. Experiments show that some of our algorithms are the fastest in practice, for real data from music information retrieval applications, or used to be the fastest at their publication time.

Our results were presented, chronologically, in [FG05a, FG06c, FG06b, FG08b] ((δ, α) and related problems), and in [FG06a, FG08a] ((δ, γ, α) and related problems).

3.1 Preliminaries

Let the pattern $P = p_0 p_1 p_2 \ldots p_{m-1}$ and the text $T = t_0 t_1 t_2 \ldots t_{n-1}$ be numerical strings, where $p_i, t_j \in \Sigma$ for $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. The numbers of distinct symbols in the pattern and in the text are denoted by σ_{p} and σ_{t}, respectively. Moreover, we use $\sigma_{\mathrm{p} \cap \mathrm{t}}$ to denote the number of characters that occur both in P and T simultaneously. Note that $\sigma_{\mathrm{p} \cap \mathrm{t}} \leq \sigma_{\mathrm{p}}, \sigma_{\mathrm{t}}, m$.

In δ-approximate string matching the symbols $a, b \in \Sigma$ match, denoted by $a =_\delta b$, iff $|a - b| \leq \delta$. Pattern P (δ, α)-matches the text substring

$t_{j_0}t_{j_1}t_{j_2}\dots t_{j_{m-1}}$, if $p_i =_\delta t_{j_i}$ for $i \in \{0,\dots,m-1\}$, where $0 < j_{i+1} - j_i \leq \alpha + 1$. If string A (δ,α)-matches string B, we write $A =_\delta^\alpha B$.

We sometimes need the symbol $\mathcal{M}$ to denote the set of all matches in the dynamic programming matrix. More precisely, $\mathcal{M} = \{(i,j) \mid p_i =_\delta t_j\}$ is the set of indexes of the δ-matching character pairs in P and T. Obviously, $|\mathcal{M}| = O(nm)$.

Pattern P (δ,γ,α)-matches the text substring $t_{j_0}t_{j_1}t_{j_2}\dots t_{j_{m-1}}$, if $p_i =_\delta t_{j_i}$ for $i \in \{0,\dots,m-1\}$, where $0 < j_{i+1} - j_i \leq \alpha + 1$, and $\sum_{j=0}^{m-1} |p_i - t_{j_i}| \leq \gamma$. To say that string A (δ,γ,α)-matches string B, we sometimes write $A =_{\delta,\gamma}^\alpha B$.

In the (δ,γ,α)-matching problem, the pattern $p_0p_1\dots p_{m-1}$ is allowed to match a substring of the text $t_0t_1\dots t_{n-1}$ with α-limited gaps, the respective pairs of matching characters' numerical values may differ only by δ at most, and the total sum of differences is limited to γ. In music information retrieval, a standard application of this model is again to "softly" match a melody hummed or whistled by a human user, against a music database, with tolerance for false notes and possible additional (ornamenting) notes in a referred pitch sequence in the database. The extra parameter, γ, limits the sum of allowed pitch distortions.

In all our average-case analyses we assume uniformly random distribution of characters in T and P, and constant α and δ/σ, unless otherwise stated. Moreover, we often write δ/σ to be terse, but the reader should understand that we mean $(2\delta+1)/\sigma$, which is the upper bound for the probability that two randomly picked characters match.

3.2 Previous work

The first algorithm for the (δ,α)-matching problem [CIM+02] is based on dynamic programming, and runs in $O(nm)$ time. This algorithm was later reformulated [CCF05a] to allow to find all pattern occurrences, instead of only the positions where the occurrence ends. This needs more time, however. The algorithm in [CCF05b] improves the average case of the one in [CCF05a] to $O(n)$, assuming constant α. More general forms of gaps were considered in [PW05], retaining the $O(nm)$ time bounds.

In molecular biology, in particular for searching patterns in protein sequences, the problem variant of α-matching with specified classes of characters is applied. In that case, the gap limits for each pattern character may be of different length, in particular, it is assumed that for many characters it is zero. For this particular problem, there exists an efficient bit-parallel non-

deterministic automaton solution [NR03]. This algorithm can be trivially generalized to handle (δ, α)-matching [CCF05b], but the time complexity becomes $O(n\lceil \alpha m/w \rceil)$ in the worst case, where w is the length of the machine word. For small α the algorithm can be made to run in $O(n)$ time on average. Sparse dynamic programming can be used to solve the problem in $O(n+|\mathcal{M}| \min\{\log(\delta+2), \log\log m\})$ time, where $\mathcal{M} = \{(i,j) \mid |p_i - t_j| \leq \delta\}$, and $|\mathcal{M}| \leq nm$ [Mäk03b]. This can be extended for the harder problem variant where transposition invariance and character insertions, substitutions or mismatches are allowed together with (δ, α)-matching [MNU05].

Recently, after our works on (δ, α)-matching, Cantone et al. [CCF08] showed four new algorithms for this problem, one of which, (δ, α)-tuned-sequential-sampling HBP, wins in their tests over the fastest of our algorithms, Simple, achieving up to 1.5 greater speed. Their algorithm is a column-wise bit-parallel one, and works in $O(nm\lceil \alpha/w \rceil)$ worst-case time and in $O(n)$ time on average.

The (δ, γ, α) problem has been rarely addressed earlier in the literature; we are aware of only one algorithm [CIM$^+$02], based on dynamic programming and running in $O(nm)$ time. Notice that a brute-force approach to this problem has worse time complexity, namely $O(nm\alpha)$.

We note yet that many works attempted to present relevant translations from musical similarity to formal matching models, and the number of aspects to consider when dealing with music data, both monophonic and polyphonic, can be really large. Those considerations go out of the scope of this work and a curious reader may be referred to e.g. [CI04] for an introduction.

3.3 Dynamic programming

The dynamic programming solution to (δ, α)-matching is based on the following recurrence [CIM$^+$02, CCF05a]:

$$D_{i,j} = \begin{cases} j, & t_j =_\delta p_i \text{ and } (i = 0 \text{ or } (i, j \geq 1 \text{ and } D_{i-1,j-1} \geq 0)), \\ D_{i,j-1}, & t_j \neq_\delta p_i \text{ and } j > 0 \text{ and } j - D_{i,j-1} < \alpha + 1, \\ -1, & \text{otherwise.} \end{cases} \tag{3.1}$$

In other words, if $D_{i,j} = j$, then the pattern prefix $p_0 \ldots p_i$ has an occurrence ending at text character t_j, i.e. $p_i =_\delta t_j$ and the prefix $p_0 \ldots p_{i-1}$ occurs at position $D_{i-1,j-1}$, and the gap between this position and the position j is at most α. If $p_i \neq_\delta t_j$, then we try to extend the match by extending the gap, i.e. we set $D_{i,j} = D_{i,j-1}$ if the gap does not become too large. Otherwise,

Alg. 14 DA-dp$(T, n, P, m, \delta, \alpha)$.

```
1      for i ← 0 to m − 1 do D_{i,0} ← −1
2      if |p_0 − t_0| ≤ δ then D_{0,0} ← 0
3      for i ← 0 to m − 1 do
4          for j ← 1 to n − 1 do
5              if |p_i − t_j| ≤ δ and (i = 0 or D_{i−1,j−1} ≥ 0) then
6                  D_{i,j} ← j
7                  if i = m − 1 then report match
8              else if D_{i,j−1} ≥ j − α then D_{i,j} ← D_{i,j−1}
9              else D_{i,j} ← −1
```

we set $D_{i,j} = -1$. The algorithm then fills the table $D_{0...m-1,0...n-1}$, and reports an occurrence ending at position j whenever $D_{m-1,j} = j$. This is simple to implement, and the algorithm runs in $O(nm)$ time using $O(nm)$ space. Alg. 14 gives the pseudocode.

It should be stressed that both cited algorithms from the literature have quadratic time complexity even in the best case. Interestingly, it is possible to improve the average-case complexity of the DP approach to $O(n)$ while preserving the $O(nm)$ time in the worst case.

As already mentioned in [CIM$^+$02], the space requirement of Alg. 14 can be made $O(n)$ by noticing that the computation of the current row of D depends only on the previous row. However, it is also true that the computation of the current *column* of D depends only on the previous column. This can be used to make the space complexity just $O(m)$, by computing the matrix column-wise, instead of row-wise, and storing only the current and previous columns.

Yet the authors of the cited work [CIM$^+$02] noted that the average time could be made just $O(n)$, but they did not explain explicitly how. The first explicit description of this idea was given by Cantone et al. in [CCF05b]. Below we present our solution [FG05a], which is similar, but different in some details. Moreover, in our analysis the assumption of constant-size gaps is not needed, as opposed to [CCF05b].

The algorithm is based on the following observation: if $D_{i...m-1,j} = -1$, for some i, j, then $D_{i+1...m-1,j+1} = -1$. This is because there is no way the recurrence can introduce any other value for those matrix cells. In other words, if $p_0 \dots p_i$ does not (δ, α)-match $t_h \dots t_{j-k}$ for any $k = 0 \dots \alpha$, then the match at the position $j + 1$ cannot be extended to $p_0 \dots p_{i+1}$.

This can be utilized by keeping track of the highest row number *top* of the current column j such that $D_{top,j} \neq -1$, and computing the next column

Alg. 15 DA-dpco$(T, n, P, m, \delta, \alpha)$.

```
1   for i ← 0 to m − 1 do D'_i ← −1
2   if |p_0 − t_0| ≤ δ then D'_0 ← 0
3   top ← m − 1
4   for j ← 1 to n − 1 do
5       for i ← 0 to top do
6           if |p_i − t_j| ≤ δ and (i = 0 or D'_{i−1} ≥ 0) then
7               D_i ← j
8               if i = m − 1 then report match
9           else if D'_i ≥ j − α then D_i ← D'_i
10          else D_i ← −1
11      while top ≥ 0 and D_top = −1 do top ← top − 1
12      if top < m − 1 then top ← top + 1
13      Dt ← D; D ← D'; D' ← Dt
```

only up to row $top + 1$. More formally, we define

$$top_j = \text{argmax}_i\{D_{i,j-1} = -1 \text{ and } D_{i-1,j-1} \neq -1\}, \tag{3.2}$$

and at text position j compute the column j only up to row top_j. We call this a *cut-off* trick. This technique was first used (in a different context) by Ukkonen [Ukk85a].

Alg. 15 gives the pseudocode to implement these two tricks. The space complexity is clearly $O(m)$, and we show that the average-case time complexity is $O(n)$. First consider the time taken for keeping track of the last active row *top*. For each text position *top* can increase only by at most 1, giving a total of $O(n)$ increments. While *top* can decrease by $O(m)$ for a processed column, which costs $O(m)$ time, the total number of decrements cannot be larger than the number of increments plus $m-1$, so the amortized cost of decrements is at most $O(n)$ as well. In total *top* can be maintained in $O(n)$ worst-case time during the whole computation.

We now show that the average value of *top* is $O(1)$, which gives a total $O(n)$ average time. The average value of *top* is the same as the average length $i+1$ of the longest prefix $p_0 \dots p_i$ that matches at some text position. The probability that $a =_\delta b$ for some $a, b \in \Sigma$ is $O(\delta/\sigma)$ assuming uniform Bernoulli model of probability. We assume that δ/σ is constant here. Hence the probability that $p_0 \dots p_i$ (δ, α)-matches $t_h \dots t_j$ is

$$\Pr(i) = O((1 - (1 - \delta/\sigma)^{\alpha+1})^i(\delta/\sigma)), \tag{3.3}$$

and the expected value of top is

$$\sum_{i=0}^{m-1} i \ \Pr(i) \ = \ O\left(\sum_{i=0}^{\infty} i \ \Pr(i)\right) \ = \ O\left(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}}\right), \tag{3.4}$$

as the sum is a geometric series, which does not depend on m, and is $O(1)$ for constant α. A similar analysis can be found from [CCF05b] for the Sequential Sampling algorithm.

Now, we are going to discuss the respective DP algorithms for (δ, γ, α)-matching. Although the basic ideas are similar, this is a harder problem than the previous one. The recurrence for (δ, γ, α)-matching is the following:

$$D_{i,j} = \begin{cases} D_{i-1,j'} + |p_i - t_j|, & p_i =_\delta t_j \wedge 0 < j - j' \leq \alpha + 1,\ \min D_{i-1,j'} \leq \gamma \\ \gamma + 1, & \text{otherwise.} \end{cases} \tag{3.5}$$

If $D_{m-1,j} \leq \gamma$, then $P =^{\alpha}_{\delta,\gamma} t_h \ldots t_j$ for some h. The matrix D is simple to compute in $O(\alpha nm)$ time. As we are only interested in the matching text positions, the $O(nm)$ space complexity can be easily improved. Using row-wise computation only the current and the previous rows need to be in memory, and hence the space complexity is just $O(n)$. For column-wise computation the space complexity is $O(\alpha m)$ as up to $\alpha + 1$ columns have to be stored.

As shown in [CIM$^+$02] the time complexity can be improved to $O(nm)$ using *min-queue* data structures [GT86]. However, in practical MIR applications α is usually so small that the simple brute-force evaluation is faster than using sophisticated data structures that have large (constant) overhead. From a practical point, a more interesting solution is (again) the simple cut-off trick that improves the average case.

For the current problem, the cut-off idea is based on the following observation: if $D_{i \ldots m-1, j-\alpha \ldots j} > \gamma$, for some i, j, then $D_{i+1 \ldots m-1, j+1} > \gamma$. This is because there is no way the recurrence can introduce any other value for those matrix cells. In other words, if $p_0 \ldots p_i$ does not (δ, γ, α)-match $t_h \ldots t_{j-k}$ for any $k = 0 \ldots \alpha$, then the match at the position $j + 1$ cannot be extended to $p_0 \ldots p_{i+1}$. This can be utilized by keeping track of the highest row number *top* of the current column j such that $D_{top+1 \ldots m-1, j-\alpha \ldots j} > \gamma$, and computing the next column only up to row $top + 1$. For this sake we maintain an array C so that $C[i]$ gives the largest j such that $p_0 \ldots p_i =^{\alpha}_{\delta,\gamma} t_h \ldots t_j$. This is easy to do in $O(1)$ time per accessed matrix cell. Alg. 16 shows the complete pseudocode.

Now consider the average time of this algorithm. Computing a single cell $D_{i,j}$ costs $O(\alpha)$ in the worst case. However, this happens only if $p_0 \ldots p_{i-1} =^{\alpha}_{\delta,\gamma} t_h \ldots t_{j'}$ and $p_i =_\delta t_j$ for some $j' \geq j - \alpha - 1$, and otherwise the cost is just $O(1)$. Therefore on average each cell is computed in $O(\alpha\delta/\sigma)$ time. Maintaining *top* costs only $O(n)$ time in total, since it can be incremented only by one per text character, and the number of decre-

Alg. 16 DGA-dpco($T, n, P, m, \delta, \gamma, \alpha$).

```
1   for j ← 0 to α + 1 do for i ← 0 to m − 1 do D_{i,j} ← γ + 1
2   for i ← 0 to m − 1 do C[i] ← −α − 1
3   D_{0,0} ← |t_0 − p_0|
4   if D_{0,0} > δ then D_{0,0} ← γ + 1
5   if D_{0,0} ≤ γ then C[0] ← 0
6   top ← m − 1
7   for j ← 1 to n − 1 do
8       C' ← C[0]
9       k ← j % (α + 2)
10      D_{0,k} ← |t_j − p_0|
11      if D_{0,k} > δ then D_{0,k} ← γ + 1
12      if D_{0,k} ≤ γ then C[0] ← j
13      for i ← 1 to top do
14          d ← |t_j − p_i|
15          min ← γ + 1
16          if d ≤ δ and j − C' ≤ α + 1 then
17              k' ← (j − 1) % (α + 2)
18              min ← D_{i−1,k'}
19              for h ← max{0, j − α − 1} to j − 2 do
20                  k' ← h % (α + 2)
21                  if D_{i−1,k'} < min then min ← D_{i−1,k'}
22          D_{i,k} ← min + d
23          C' ← C[i]
24          if D_{i,k} ≤ γ then
25              C[i] ← j
26              if i = m − 1 then report match
27      while top ≥ 0 and j − C[top] > α + 1 do top ← top − 1
28      if top < m − 1 then top ← top + 1
```

ments cannot be larger than the number of increments. The average time of this algorithm also depends on the average value of *top*, i.e. the total time is $O(n\ \mathrm{avg}(top)\ \alpha\delta/\sigma)$. For $\gamma = \infty$ it can be shown that $\mathrm{avg}(top) = O\left(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}}\right)$ [CCF05b]. This is $O(\alpha\delta/\sigma)$ for $\delta/\sigma < 1 - \alpha^{-1/(\alpha+1)}$, so the average time is at most $O(n(\alpha\delta/\sigma)^2)$. We have neglected the effect of γ, but by forcing the γ condition the time can only improve, hence our analysis is pessimistic. In the worst case the time is $O(\alpha nm)$, but this can be improved to $O(nm)$ as in [CIM$^+$02], the only difference being that we need m queues, since we are computing column-wise (as opposed to row-wise in [CIM$^+$02]).

Note that the average-case analyses may have little practical value for music, which is far from random and most of its pitch alphabet is hardly ever used. Hence, practical evaluation of these algorithms, for both considered problems (see Sect. 3.13) will make more sense.

3.4 Row-wise sparse dynamic programming

The algorithm for (δ, α)-matching that we now present can be seen as a row-wise variant of the sparse dynamic programming algorithm of the algorithm in [MNU05, Sect. 5.4]. We show how to improve its average-case running time. Our variant can also be easily extended to handle more general gaps, see Sect. 3.11.

3.4.1 Efficient worst case

From the recurrence of D it is clear that the interesting computation happens when $t_j =_\delta p_i$, and otherwise the algorithm just copies previous entries of the matrix or fills some of the cells with a constant.

Further we refer to $\mathcal{M}$, the set of indexes for all δ-matching pairs of characters from P and T. For every $(i, j) \in \mathcal{M}$ we compute a value $d_{i,j}$. For the pair (i, j) where $d_{i,j}$ is defined, it corresponds to the value of $D_{i,j}$. If $(i, j) \notin \mathcal{M}$, then $d_{i,j}$ is not defined. Note that $d_{m-1,j}$ is always defined if P occurs at $t_{h \dots j}$ for some $h < j$. The new recurrence is

$$d_{i,j} = j \mid (i-1, j') \in \mathcal{M} \text{ and } 0 < j - j' \leq \alpha + 1 \text{ and } d_{i-1,j'} \neq -1,$$

and -1 otherwise. Computing the d values is easy once $\mathcal{M}$ is computed. As we have an integer alphabet, we can use table look-ups to compute $\mathcal{M}$ efficiently. Instead of computing $\mathcal{M}$, we compute lists $L[p_i]$, where $L[p_i] = \{j \mid p_i =_\delta t_j\}$. These are obtained by scanning the text linearly, and inserting j into each list $L[p_i]$ such that p_i δ-matches t_j. Clearly, there are at most $O(\delta)$ and in average only $O(\delta\sigma_{\mathrm{p}}/\sigma)$ symbols p_i that δ-match t_j. Therefore this can be obtained in $O(\delta n)$ worst-case time, and the average-case complexity is $O(n(\delta\sigma_{\mathrm{p}}/\sigma + 1))$. Note that $|\mathcal{M}|$ is $O(nm)$ in the worst case, but the total length of all the lists is at most $O(\min\{\sigma_{\mathrm{p}}, \delta\}n)$, hence L is a compact representation of $\mathcal{M}$. The indexes in $L[p_i]$ will be in increasing order.

Consider a row-wise computation of d. The values of the first row $d_{0,j}$ correspond one to one to the list $L[p_0]$, that is, the text positions j where $p_0 =_\delta t_j$. The subsequent rows d_i correspond to $L[p_i]$, with the additional constraint that $j - j' \leq \alpha + 1$, where $j' \in L[p_{i-1}]$ and $d_{i-1,j'} \neq -1$. Since the values in $L[p_i]$ and d_{i-1} are in increasing order, we can compute the current row i by traversing the lists $L[p_i]$ and d_{i-1} simultaneously, trying to enforce the condition that $L[p_i][h] - d_{i-1,k} \leq \alpha + 1$ for some h, k. If the condition cannot be satisfied for some h, we store -1 to $d_{i,h}$, otherwise we store the text position $L[p_i][h]$. The algorithm traverses L and $\mathcal{M}$ linearly,

and hence runs in $O(n+|\mathcal{M}|)$ worst-case time. We now consider improving the average-case time of this algorithm.

3.4.2 Efficient average case

The basic sparse algorithm still does some redundant computation. To compute the values $d_{i,j}$ for the current row i, it laboriously scans through the list $L[p_i]$, for all positions, even for the positions close to where $p_0 \ldots p_{i-1}$ did not match. In general, the number of text positions with matching pattern prefixes decreases exponentially on average when the prefix length i increases. Yet, the list length $|L[p_i]|$ will stay approximately the same. The goal is therefore to improve the algorithm so that its running time per row depends on the number of matching pattern prefixes on that row, rather than on the number of δ-matches for the current character on that row.

The modifications are simple: (1) the values $d_{i,j} = -1$ are not maintained explicitly, they are just not stored since they do not affect the computation; (2) the list $L[p_i]$ is not traversed sequentially, position by position, but binary search is used to find the next value that may satisfy the condition that $L[p_i][h] - d_{i-1,k} \leq \alpha + 1$ for some h, k.

Consider now the average search time of this algorithm. The average length of each list $L[p_i]$ is $O(n\delta/\sigma)$. Hence this is the time needed to compute the first row of the matrix, i.e. we just copy the values in $L[p_0]$ to be the first row of d. For the subsequent rows we execute one binary search over $L[p_i]$ per each stored value in row i of the matrix. Hence in general, computing the row i of the matrix takes time $O(|d_{i-1}| \log(n\delta/\sigma))$, where $|d_i|$ denotes the number of stored values in row i. For $i > 0$ this decreases exponentially as $|d_i| = O(n(\delta/\sigma) \times \rho^i)$, where $\rho = 1-(1-\delta/\sigma)^{\alpha+1} < 1$ is the probability that a pattern symbol δ-matches in a text window of length α symbols. Summing up the resulting geometric series over all rows we obtain $O(n\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}})$, which is $O(n\alpha\delta/\sigma)$ for $\delta/\sigma < 1-\alpha^{-1/(\alpha+1)}$. In particular this is $O(n)$ for $\alpha = O(\sigma/\delta)$. Hence the average search time is $O(n+n\alpha\delta/\sigma \log(n\delta/\sigma))$. However, the worst-case search time is also increased to $O(n+|\mathcal{M}|\log(|\mathcal{M}|/m))$. We note that this can be improved to $O(n+|\mathcal{M}|\log\log((nm)/|\mathcal{M}|))$ by using efficient priority queues [Joh82] instead of binary search.

3.4.3 Faster preprocessing

The $O(\delta n)$ (worst-case) preprocessing time can dominate the average-case search time in some cases. Note however, that the preprocessing time can never exceed $O(n+|\mathcal{M}|)$. We now present two methods to improve the

preprocessing time. The first one reduces the worst-case preprocessing cost to $O(\sqrt{\delta}n)$, and improves its average case as well. The second method achieves $O(n)$ preprocessing time, but the worst case search time is slightly increased.

$O(\sqrt{\delta}n)$ time preprocessing

The basic idea is to partition the alphabet into $\sigma/\sqrt{\delta}$ disjoint intervals $I_h, h = 0 \ldots \sigma/\sqrt{\delta} - 1$ of size $\sqrt{\delta}$ each (w.l.o.g. we assume that δ is a square number and $\sqrt{\delta}$ divides σ). Then, for each alphabet symbol s, its respective $[s - \delta, s + \delta]$ interval wholly covers $\Theta(\sqrt{\delta})$ intervals I_h, and also can partially cover at most two I_h intervals. Two kinds of lists are computed in the preprocessing, L_{b} (for "boundary" cases) and L_{c} (for "core"). For each character t_j from text T, at most $2(\sqrt{\delta} - 1)$ lists $L_{\mathrm{b}}[p_i]$ are extended with one entry, the text position j, and those lists correspond to the pattern alphabet symbols from the partially covered intervals I_h. For example, if $\Sigma = \{0, \ldots, 29\}$, $\sqrt{\delta} = 3$, and $t_j = 10$, then the $[t_j - \delta, t_j + \delta]$ interval is $[1, 19]$, and j is appended to the lists $L_{\mathrm{b}}[1]$, $L_{\mathrm{b}}[2]$, $L_{\mathrm{b}}[18]$, $L_{\mathrm{b}}[19]$, assuming that P contains all the symbols 1, 2, 18 and 19 (if not, the respective lists are not built at all). Similarly, each character t_j also causes to append j to $O(\sqrt{\delta})$ lists $L_{\mathrm{c}}[p_i/\sqrt{\delta}]$, those that correspond to the I_h intervals wholly covered by $[t_j - \delta, t_j + \delta]$.

Fig. 3.1 illustrates. The δ-interval for $t_j = 10$ spans over the dark-shaded and light-shaded cells. The light-shaded symbols (1, 2, 18, 19) are the "boundary" cases corresponding to the two partially covered intervals, and j is appended to the corresponding L_{b} lists. The dark-shaded intervals (1, 2, 3, 4, 5) are the fully covered "core" cases, and j is appended to the corresponding L_{c} lists.

More formally (and still assuming for simplicity that δ is a square number) text position j is appended to the lists $L_{\mathrm{b}}[p_i]$ for

$$p_i \in \{t_j - \delta \ldots \left\lceil (t_j - \delta)/\sqrt{\delta} \right\rceil \sqrt{\delta} - 1, \quad \left\lfloor (t_j + \delta + 1)/\sqrt{\delta} \right\rfloor \sqrt{\delta} \ldots t_j + \delta\}.$$

Likewise, j is appended to the lists $L_{\mathrm{c}}[p_i/\sqrt{\delta}]$ for

$$p_i/\sqrt{\delta} \in \{\left\lceil (t_j - \delta)/\sqrt{\delta} \right\rceil \ldots \left\lfloor (t_j + \delta + 1)/\sqrt{\delta} \right\rfloor - 1\}.$$

It is easy to see that the preprocessing needs $O(\sqrt{\delta}n)$ time in the worst case and $O(n\sqrt{\delta}\sigma_{\mathrm{p}}/\sigma)$ time on average.

The search is again based on a binary search routine, but in this variant we binary search two lists: $L_{\mathrm{b}}[p_i]$ and $L_{\mathrm{c}}[p_i/\sqrt{\delta}]$, as the δ-matches to p_i may

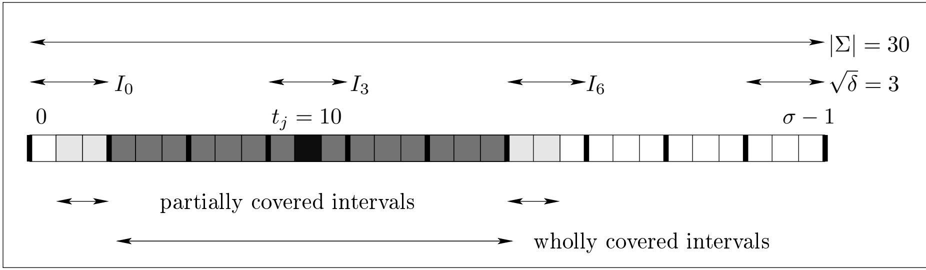

Figure 3.1: Row-wise SPD for (δ, α)-matching, $O(\sqrt{\delta}n)$ time preprocessing

be stored either at some L_b, or at some L_c list. This increases both the average and worst-case search cost only by a constant factor.

We can generalize this idea and have a preprocessing/search tradeoff. Namely, we may have k levels, turning the preprocessing cost into $O(k\delta^{1/k}n)$, for the price of a multiplicative factor k in the search. For $k = \log \delta$ the preprocessing cost becomes $O(n \log \delta)$, and both the average and worst-case search times are multiplied by $\log \delta$ as well.

$O(n)$ time preprocessing

We partition the alphabet into $\lceil \sigma/\delta \rceil$ disjoint intervals of width δ. With each interval a list of character occurrences will be associated. Namely, each list $L[i], i = 0 \ldots \lceil \sigma/\delta \rceil - 1$, corresponds to the characters $i\delta \ldots \min\{(i + 1)\delta - 1, \sigma - 1\}$. During the scan over the text in the preprocessing phase, we append each index j to up to three lists: $L[k]$ for such k that $k\delta \le t_j \le (k+1)\delta - 1$, $L[k-1]$ (if $k - 1 \ge 0$), and $L[k+1]$ (if $k + 1 \le \lceil \sigma/\delta \rceil - 1$). Note that no character from the range $[t_j - \delta \ldots t_j + \delta]$ can appear out of the union of the three corresponding intervals. Such preprocessing clearly needs $O(n)$ space and time in the worst case.

Now the search algorithm runs the binary search over the list $L[k]$ for such k that $k\delta \le p_i \le (k+1)\delta - 1$, as any j such that $t_j =_\delta p_i$ must have been stored at $L[k]$. Still, the problem is there can be other text positions stored on $L[k]$ too, as the only thing we can deduce is that for any j in the list $L[k]$, t_j is $(2\delta - 1)$-match to p_i. To overcome this problem, we have to verify if t_j is a real δ-match. If $t_j \neq_\delta p_i$, we read the next value from $L[k]$ and continue analogously. After at most $\alpha + 1$ read indexes from $L[k]$ we either have found a δ-match prolonging the matching prefix, or we have fallen off the $(\alpha + 1)$-sized window. As a result, the worst-case time complexity is $O(n + |\mathcal{M}|(\log n + \alpha))$. The average time

Alg. 17 DA-sdp-rows($T, n, P, m, \delta, \alpha$).

```
1   for j ← 0 to n − 1 do
2       for c ← max{0, ⌊t_j/δ⌋ − 1} to min{⌊(σ − 1)/δ⌋, ⌊t_j/δ⌋ + 1} do
3           L[c] ← L[c] ∪ {j}
4   for i ← 0 to |L[p_0]| − 1 do
5       j ← L[p_0][i]
6       if |t_j − p_0| ≤ δ then D'_i ← j
7   h ← |L[p_0]|
8   for i ← 1 to m − 1 do
9       c ← p_i; pl ← h; k ← 0; h ← 0; u ← 0
10      while u < |L[c]| and k < pl do
11          j ← L[c][u]
12          do  j' ← D'_k
13              if j − j' > α + 1 and k < pl then k ← k + 1
14          while j − j' > α + 1 and k < pl
15          if j' < j and k < pl and |t_j − c| ≤ δ then
16              D_h ← j; h ← h + 1
17              if i = m − 1 then report match
18          if k < pl then u ← min{v | D'_k < L[c][v], v > u}
19      swap(D, Dt)
```

in this variant becomes $O(n + n\alpha\delta/\sigma \log n)$. Alg. 17 shows the complete pseudocode.

3.4.4 Improved algorithm for large α

In this section we present a variant of the row-wise SDP algorithm, particularly suited to problem instances with large α.

In the preprocessing, we again compute lists $L[p_i] = \{j \mid t_j =_\delta p_i\}$. But now we also store $2\lfloor n/(\alpha+1)\rfloor$ pointers to each list. In each list, for each $j = k(\alpha+1)$ where $k \in 0 \ldots n/(\alpha+1)-1$, there are two pointers, showing the leftmost and the rightmost item with the value from the interval $[j, j+\alpha+1]$. These pointers are kept in two 2-dimensional arrays, named $\mathcal{L}$ and $\mathcal{R}$. More formally, the elements of $\mathcal{L}$ and $\mathcal{R}$ are defined in the following way:

$$\begin{aligned}\mathcal{L}[p_i,k] &= \min\{j \mid j \in L[p_i] \text{ and } j \in [k(\alpha+1)\ldots(k+1)(\alpha+1)]\},\\ \mathcal{R}[p_i,k] &= \max\{j \mid j \in L[p_i] \text{ and } j \in [k(\alpha+1)\ldots(k+1)(\alpha+1)]\},\end{aligned}$$

assuming the minimum and the maximum is seeked over a non-empty slice of a list $L[p_i]$. If this is not the case, the respective pointers are set to null. In total, the extra preprocessing cost is $O(\delta n + \sigma_{\rm p} n/\alpha)$ in time, and $O(\sigma_{\rm p} n/\alpha)$ space, in the worst case.

The search is basically prefix prolongation. A specific trait of the algorithm is that during the search we are not interested in finding all matching

prefixes: what is enough are (at most) two prefixes per an $(\alpha+1)$-sized chunk of each row (except for the last row, where we perform an extra scan, to be described later). The end positions of those prefixes are maintained in two auxiliary arrays, $\mathcal{C}_{\mathrm{L}}$ and $\mathcal{C}_{\mathrm{R}}$, of size $\lfloor n/(\alpha+1)\rfloor$ each. They are initialized with the exact copy of the rows $\mathcal{L}[p_0]$ and $\mathcal{R}[p_0]$, respectively.

Now we assume the matrix row i we are in is at least 1. W.l.o.g. we also assume that we are in the column at least $\alpha+1$. For each $k \in 1 \ldots n/(\alpha+1)-1$ we read $\mathcal{L}[p_i, k]$ and $\mathcal{R}[p_{i-1}, k-1]$, and if both are non-null and $\mathcal{L}[p_i, k] - \mathcal{R}[p_{i-1}, k-1]$ is at most $\alpha+1$, then we have found a relevant prefix, which we write to $\mathcal{C}_{\mathrm{L}}$. If not, we check if $\mathcal{L}[p_i, k] - \mathcal{L}[p_{i-1}, k] > 0$ (note that this difference cannot be greater than $\alpha+1$, so testing for a positive difference of non-null values is all we need). Affirmative answer again corresponds to finding a relevant prefix (and requires updating $\mathcal{C}_{\mathrm{L}}[k]$), but a negative one means that we have to look for a prefix prolongation somewhere further in the current chunk. In such case, we perform a binary search over the fragment of the list $L[p_i]$ with the boundaries kept in the pointers $\mathcal{L}[p_i, k]$ and $\mathcal{R}[p_i, k]$, to find the smallest value being greater than $\mathcal{L}[p_{i-1}, k]$. The interval has as most $\alpha+1$ items, so the binary search cost is $O(\log\alpha)$. If this results in a failure (which happens only if the considered interval is empty), it means that we do not have a prefix ended in the current chunk, and $\mathcal{C}_{\mathrm{L}}[k]$ should be updated with a null value.

Analogously we proceed at the right boundary of each chunk. The invariant for the procedure is that after processing a row i, all the end positions of $p_0 \ldots p_i$ in the text chunk $t_{k(\alpha+1)} \ldots t_{(k+1)(\alpha+1)}$ are exactly those $\mathcal{C}_{\mathrm{L}}[k] \le j \le \mathcal{C}_{\mathrm{R}}[k]$ for whose t_j δ-matches p_i, assuming non-null values of $\mathcal{C}_{\mathrm{L}}[k]$ and $\mathcal{C}_{\mathrm{R}}[k]$. If either $\mathcal{C}_{\mathrm{L}}[k]$ or $\mathcal{C}_{\mathrm{R}}[k]$ is null, there are no prefixes ending in the given text chunk.

As mentioned, the last row requires an extra scan, to find all the δ-matches between the positions in $\mathcal{C}_{\mathrm{L}}[k]$ and $\mathcal{C}_{\mathrm{R}}[k]$, for all $k \in 0 \ldots n/(\alpha+1)-1$. Note that it is possible that $\mathcal{C}_{\mathrm{L}}[k] = \mathcal{C}_{\mathrm{R}}[k]$ or $\mathcal{C}_{\mathrm{R}}[k] = \mathcal{C}_{\mathrm{L}}[k+1]$, so we must be careful not to count duplicates more than once. This stage needs $O(n)$ time, i.e. is always dominated by the preprocessing time.

The overall search complexity can be bounded by $O(n + nm\log(\alpha)/\alpha)$, but actually we can bound it better: with $O(n + nm\log(\alpha|\mathcal{M}|/(nm))/\alpha)$. Indeed, a single chunk may have up to $\alpha+1$ items over which we binary search, but in total there are only $|\mathcal{M}|$ matches in the matrix, which can be much less than nm. This means that on average there are $O(\alpha|\mathcal{M}|/(nm))$ items in a chunk, and equal number of matches in chunks leads also to the worst overall case, which is trivially implied from the convexity of the log function.

3.5 Column-wise sparse dynamic programming

In this section we present a column-wise variant for (δ, α)-matching. This algorithm runs in $O(n + n\alpha\delta/\sigma)$ and $O(n + \min(|\mathcal{M}|\alpha, nm))$ average and worst-case time, respectively.

The algorithm processes the dynamic programming matrix column-wise. Let us define *last prefix occurrence* $\mathcal{D}$ as

$$\mathcal{D}_{i,j} = \begin{cases} j', & \max j' \le j \mid p_0 \ldots p_i =_\delta^\alpha t_h \ldots t_{j'}, \\ -\alpha - 1, & \text{otherwise.} \end{cases} \tag{3.6}$$

Note that $\mathcal{D}_{0,j} = j$ if $p_0 =_\delta t_j$. Note also that $\mathcal{D}_{i,j}$ is just an alternative definition of $D_{i,j}$ (Eq. (3.5)). The pattern matching task is then to report every j such that $\mathcal{D}_{m-1,j} = j$. As seen, this is easy to compute in $O(nm)$ time. In order to do better, we maintain a list of *window prefix occurrences* $\mathcal{W}_j$ that contains for the current column j all the rows i such that $j - \mathcal{D}_{i,j} \le \alpha$ where $i \in \mathcal{W}_j$.

Assume now that we have computed $\mathcal{D}$ and $\mathcal{W}$ up to column $j-1$, and want to compute $\mathcal{D}$ and $\mathcal{W}$ for the current column j. The invariant is that $i \in \mathcal{W}_{j-1}$ iff $j - \mathcal{D}_{i,j-1} \le \alpha + 1$. In other words, if $i \in \mathcal{W}_{j-1}$ and $j' = \mathcal{D}_{i,j-1}$, then $p_0 \ldots p_i =_\delta^\alpha t_h \ldots t_{j'}$ for some h. Therefore, if $t_j =_\delta p_{i+1}$, then the (δ, α)-matching prefix from $\mathcal{D}_{i,j-1}$ can be extended to text position j and row $i+1$. In such case we update $\mathcal{D}_{i+1,j}$ to be j, and put the row number $i+1$ into the list $\mathcal{W}_j$. This is repeated for all values in $\mathcal{W}_{j-1}$. After this we check if also p_0 δ-matches the current text character t_j, and in such case set $\mathcal{D}_{0,j} = j$ and insert j into $\mathcal{W}_j$. Finally, we must put all the values $i \in \mathcal{W}_{j-1}$ to $\mathcal{W}_j$ if the row i was not already there, and still it holds that $j - \mathcal{D}_{i,j} \le \alpha$. This completes the processing for the column j.

Alg. 18 gives the code. Note that the additional space we need is just $O(m)$, since only the values for the previous column are needed for $\mathcal{D}$ and $\mathcal{W}$. In the pseudocode this is implemented by using $\mathcal{W}$ and $\mathcal{W}'$ to store the prefix occurrences for the current and previous column, respectively.

The average-case running time of the algorithm depends on how many values there are on average in the list $\mathcal{W}$. Similar analysis as in Sect. 3.4 can be applied to show that this is $O(\alpha\delta/\sigma)$. Each value is clearly processed in constant worst-case time, and hence the algorithm runs in $O(n + n\alpha\delta/\sigma)$ average time. In the worst case the total length of the lists for all columns is $O(\min(|\mathcal{M}|\alpha, nm))$, and therefore the worst-case running time is $O(n + \min(|\mathcal{M}|\alpha, nm))$, since every column must be visited. The preprocessing phase only needs to initialize $\mathcal{D}$, which takes $O(m)$ time.

Alg. 18 DA-sdp-columns$(T, n, P, m, \delta, \alpha)$.

1 **for** $i \leftarrow 0$ **to** $m-1$ **do** $\mathcal{D}_i \leftarrow -\alpha - 1$
2 $top \leftarrow 0$
3 **for** $j \leftarrow 0$ **to** $n-1$ **do**
4 $c \leftarrow t_j$; $h \leftarrow 0$
5 **for** $i \leftarrow 0$ **to** $top - 1$ **do**
6 $pr \leftarrow \mathcal{W}'_i$
7 **if** $|c - p_{pr+1}| \leq \delta$ **then**
8 **if** $pr + 1 < m - 1$ **then**
9 $\mathcal{W}_h \leftarrow pr + 1$; $h \leftarrow h + 1$
10 **else**
11 report match
12 **if** $|c - p_0| \leq \delta$ **then**
13 $\mathcal{W}_h \leftarrow 0$; $h \leftarrow h + 1$
14 **for** $i \leftarrow 0$ **to** $h - 1$ **do** $\mathcal{D}_{\mathcal{W}_i} \leftarrow j$
15 **for** $i \leftarrow 0$ **to** $top - 1$ **do**
16 **if** $\mathcal{D}_{\mathcal{W}_i} \neq j$ and $j - \mathcal{D}_{\mathcal{W}_i} \leq \alpha$ **then**
17 $\mathcal{W}_h \leftarrow \mathcal{W}'_i$; $h \leftarrow h + 1$
18 $top \leftarrow h$
19 swap$(\mathcal{W}, \mathcal{W}')$

Finally, we note that this algorithm can be seen as a simplification of the algorithm in [MNU05, Sect. 5.4]. We avoid the computation of $\mathcal{M}$ in the preprocessing phase and traversing it in the search phase. The price we pay is a deterioration in the worst-case time complexity, but we achieve simpler algorithm that is efficient on average. This also makes the algorithm alphabet independent.

3.6 Simple algorithm for (δ, α)-matching

In this section we will develop a simple algorithm that in practice performs very well on small (δ, α). The algorithm inherits the main idea from Alg. 17, and actually can be seen as its brute-force variant. The algorithm has two traits that distinguish it from Alg. 17: (i) the preprocessing phase is interweaved with the searching (lazy evaluation); (ii) binary search of the next qualifying match position is replaced with a linear scan in an $\alpha + 1$ wide text window. These two properties make the algorithm surprisingly simple and efficient on average, but impose an $O(\alpha)$ multiplicative factor in the worst-case time bound.

The algorithm begins by computing a list L of δ-matches for p_0:

$$L_0 = \{j \mid t_j =_\delta p_0\}.$$

This takes $O(n)$ time (and solves the (δ, α)-matching problem for patterns of length 1). The matching prefixes are then iteratively extended, subsequently computing lists:

$$L_i = \{j \mid t_j =_\delta p_i \text{ and } j' \in L_{i-1} \text{ and } 0 < j - j' \leq \alpha + 1\}.$$

List L_i can be easily computed by linearly scanning list L_{i-1}, and checking if any of the text characters $t_{j'+1} \ldots t_{j'+\alpha+1}$, for $j' \in L_{i-1}$ δ-matches p_i. This takes $O(\alpha|L_{i-1}|)$ time. Clearly, in the worst case the total length of all the lists is $\sum_i L_i = |\mathcal{M}|$, and hence the algorithm runs in $O(n + \alpha|\mathcal{M}|)$ worst-case time.

With one simple optimization the worst case can be improved to $O(n + \min\{\alpha|\mathcal{M}|, nm\})$ (improving also slightly the constant in front of the average time complexity). When computing the current list L_i, Simple algorithm may inspect some text characters several times, because the subsequent text positions stored in L_{i-1} can be close to each other, in particular, they can be closer than $\alpha + 1$ positions. In this case the $\alpha + 1$ wide text windows will overlap, and same text positions are inspected more than once. Adding a simple safeguard to detect this, each value in the list L_i can be computed in $O(\alpha)$ *worst-case* time, and in $O(1)$ best case time. In particular, if $|\mathcal{M}| = O(nm)$, then the overlap between the subsequent text windows is $O(\alpha)$, and each value of L_i is computed in $O(1)$ time. This results in $O(nm)$ worst-case time. The average case is improved as well. Alg. 19 shows the pseudocode, including this improvement.

Consider now the average case. List L_0 is computed in $O(n)$ time. The length of this list is $O(n\delta/\sigma)$ on average. Hence the list L_1 is computed in $O(\alpha n\delta/\sigma)$ average time, resulting in a list L_1, whose average length is $O(n\delta/\sigma \times \alpha\delta/\sigma)$. In general, computing the list L_i takes

$$O(\alpha|L_{i-1}|) = O(n\alpha^i(\delta/\sigma)^i) = O(n(\alpha\delta/\sigma)^i)$$

average time. This is exponentially decreasing if $\alpha\delta/\sigma < 1$, i.e. if $\alpha < \sigma/\delta$, and hence, summing up, the total average time is $O(n)$.

3.6.1 Sublinear average case

In this section we show how the average-case time of Simple can be improved. The basic observation is that while building the list L_0 not all δ-matches need to be inserted, but rather only those that have hope to be extended to a complete match of the whole pattern. In other words, some of the δ-matches can be skipped. This can be achieved using Boyer–Moore–Horspool (BMH)

Alg. 19 DA-sdp-simple$(T, n, P, m, \delta, \alpha)$.

```
1      h ← 0
2      for j ← 0 to n − 1 do
3          if |t_j − p_0| ≤ δ then
4              L[h] ← j; h ← h + 1
5      for i ← 1 to m − 1 do
6          pn ← h; h ← 0; L[pn] = n − 1
7          for j ← 0 to pn − 1 do
8              for j′ ← L[j] + 1 to min(L[j + 1], L[j] + α + 1) do
9                  if |t_j′ − p_i| ≤ δ then
10                     L′[h] ← j′; h ← h + 1
11                     if i = m − 1 then report match
12         swap(L, L′)
```

[Hor80] strategy. We therefore build the list L_0 using the BMH approach (*filtering*), and then continue with plain Simple to compute the lists $L_{1 \ldots m-1}$. This can be seen as a *verification* phase.

In what follows, we build L_0 scanning the text backwards. Lists $L_{1 \ldots m-1}$ are built as before, using Simple. We first need the following definition:

$$S[c] = \min\{i, m \mid p_i =_\delta c\}.$$

This implies that if $S[t_j] \neq m$, then $p_{S[t_j]} =_\delta t_j$. This gives us a *shifting* rule. Assume now that p_0 is aligned with t_j. We then execute the following algorithm:

1. If $|p_0 - t_j| \leq \delta$, then put j into the list L_0.
2. Check $t_{j-\alpha-1 \ldots j-1}$ right to left, computing $s = \text{argmin}_i\{S[t_{j-i}] \mid i \in [1 \ldots \alpha+1]\}$. If several values of i give the same minimum shift value, return the smallest i.
3. Shift the pattern with $j \leftarrow j - (S[t_{j-s}] + s)$ to align t_{j-s} with $p_{S[t_{j-s}]}$.
4. If $j \geq 0$, then go to 1.
5. Pass the computed list L_0 to Simple, and compute lists $L_{1 \ldots m-1}$.

The core of the algorithm is the step 2. We scan the text window $t_{j-\alpha-1 \ldots j-1}$. If some occurrence overlaps this window, then some pattern character must δ-match one of the characters in this window. We therefore compute the smallest shift to align some δ-matching pattern character to one of these text characters. If such character does not exist, then the pattern occurrence cannot overlap this window, and the whole pattern is shifted past the window, i.e. the shift is $m + \alpha + 1$ characters.

The text scanning is performed backwards, as we want to put the *starting* positions (instead of ending positions) of the possible occurrences into the

Alg. 20 DA-sdp-simple-compute-$L_0(T, n, P, m, \delta, \alpha)$.

```
1   for i ← 0 to σ − 1 do S[i] ← m
2   for i ← m − 1 downto 0 do
3       for j ← max(0, p_i − δ) to min(σ − 1, p_i + δ) do S[j] ← i
4   h ← 0; j ← n − m + 1
5   while j ≥ 0 do
6       if |t_j − p_0| ≤ δ then
7           L'[n − h − 1] ← j; h ← h − 1
8       k ← α; s ← m
9       for i ← 0 to α do
10          if j − i − 1 ≥ 0 and S[t_{j−i−1}] < s then
11              s ← S[t_{j−i−1}]; k ← i
12      j ← j − (s + k + 1)
13  L[0 . . . h − 1] ← L'[n − h . . . n − 1]
14  /* continue with Simple from row 1 */
```

list L_0. The only reason for this is to be compatible with Simple algorithm. Alg. 20 gives the pseudocode.

As opposed to exact BMH matching, in this variant any shift requires $O(\alpha)$ prior character accesses. The average pattern shift can be lower-bounded by $O(\min(m, 1/\rho))$, where ρ is the probability of a δ-matching symbol in $(\alpha + 1)$-window, that is, $\rho = 1 - (1 - \delta/\sigma)^{\alpha+1}$. This probability is $O(\frac{\alpha+1}{\sigma/\delta})$ if $\alpha + 1 < \sigma/\delta$. Thus the average time for large m and small α is $O(n\alpha^2\delta/\sigma)$, which also dominates the verification phase.

3.7 Simple algorithm for (δ, γ, α)-matching

Now we show how to modify the Simple algorithm for (δ, γ, α)-matching. Again, it is a very practical choice for small (δ, γ, α).

The definition of the list L_0 remains unchanged, but now, of course, the lists L_i, $i > 0$, take into account also the γ limitation. Namely:

$$L_i = \{j \mid p_i =_\delta t_j \wedge D_{i-1,j'} + |p_i - t_j| \leq \gamma \wedge j' \in L_{i-1} \wedge 0 < j - j' \leq \alpha + 1\}. \quad (3.7)$$

Building a list L_i from L_{i-1} is like for the previous problem, with the extra check for the sum of errors, and each time some j is appended into L_i, the corresponding matrix cell $D_{i,j}$ is also updated to hold the sum of errors for the matching pattern prefix $p_0 \ldots p_i$. Note that we put each j only once into L_i, but there can be up to $\alpha + 1$ different $j' \in L_{i-1}$ that may cause it. In the case that j is already in L_i, we only update $D_{i,j}$ if the new sum is smaller. This procedure (Alg. 21) takes $O(\alpha|L_{i-1}|)$ time.

Alg. 21 DGA-sdp-simple$(T, n, P, m, \delta, \gamma, \alpha)$.

```
1   h ← 0
2   for j ← 0 to n − 1 do
3       M[j] ← γ + 1
4       d ← |t_j − p_0|
5       if d ≤ δ then
6           L1[h] ← j
7           G[h] ← d
8           h ← h + 1
9   for i ← 1 to m − 1 do
10      pn ← h; h ← 0
11      for j ← 0 to pn − 1 do
12          g ← G[j]
13          for k ← L1[j] + 1 to min(L1[j] + α + 1, n − 1) do
14              d ← |t_k − p_i|
15              if d ≤ δ and g + d ≤ γ then
16                  if M[k] ≤ γ then
17                      if g + d < M[k] then M[k] ← g + d
18                  else
19                      L2[h] ← k
20                      h ← h + 1
21                      M[k] ← g + d
22                  if i = m − 1 and M[k] ≥ 0 then
23                      report match
24                      M[k] ← −1
25      if i < m − 1 then for j ← 0 to h − 1 do
26          k ← L2[j]
27          G[j] ← M[k]
28          M[k] ← γ + 1
29      Lt ← L1; L1 ← L2; L2 ← Lt
```

It is clear that the worst-case time complexity of this algorithm is $O(n + \alpha|\mathcal{M}|)$, but not better, since this time we cannot (easily) benefit from the possibly overlapping text windows, since the invariant of this algorithm is that $D_{i,j}$ stores the error sum associated with the cheapest path to the cell (i, j). The average-case analysis is exactly like before, we simply ignore the γ condition, making it pessimistic. Still, because of the preprocessing, the algorithm never works in sublinear time, hence the average time remains $O(n)$.

3.7.1 Improving the worst case

As a theoretical option, we present a way to improve the worst case of this algorithm to $O(\min\{mn, \alpha|\mathcal{M}|\})$, reaching the worst-case complexity of the Simple algorithm for (δ, α)-matching. The idea, again, is to avoid brute

force handling of overlapping windows of size $\alpha + 1$, but the used means are much more sophisticated. We make use of the min-queue data structure [GT86], similarly to the concept from [CIM+02] where the min-queue was used with plain dynamic programming.

For the current cell $D_{i+1,j}$, the keys in the queue are the values of $D_{i,j'}$, where $j' \in \{L_i \mid 0 < j - L_i < \alpha + 1\}$. For calculating $D_{i+1,j}$ it is enough to add its individual error to the minimum sum of errors from the queue. An algorithmic challenge is to update the queue quickly. For each processed cell only 0 or 1 values have to be inserted to the front of the queue and from 0 to $\alpha + 1$ deleted from the tail. Note however that only $O(1)$ cells (amortized) are inserted or deleted at each step. All the operations can then be done in $O(1)$ time with the min-queue data structure. This gives $O(\min\{mn, \alpha|\mathcal{M}|\})$ worst-case time.

Finally, the $O(\alpha)$ factor can be removed by precomputing $\mathcal{M}$. This can be done in $O(\min\{|\mathcal{M}| + n, \delta n\})$ worst-case time and $O(n(\delta\sigma_{\mathrm{p}}/\sigma + 1))$ average-case time for integer alphabets (see Sect. 3.9). Having $\mathcal{M}$ available, we can avoid the brute force scanning for δ-matches. $\mathcal{M}$ can be stored e.g. in Johnson's data structure [Joh82] which supports a homogeneous sequence of insertions and successor searches in $O(\log\log(nm/|\mathcal{M}|))$ time. This gives $O(|\mathcal{M}|\log\log(nm/|\mathcal{M}|))$ worst-case time, but destroys the good average case because of the costly precomputation. Note that $O(|\mathcal{M}| + n)$ worst-case algorithm is easy to obtain by simply scanning $\mathcal{M}$ linearly, but this then becomes also the average case. Unfortunately, it is hard to find a solution that optimizes the worst case and also keeps the average case time complexity.

3.8 Bit-parallel dynamic programming for (δ, α)-matching

In this section we show how bit-parallelism can be used to bring the worst-case complexity of dynamic programming down to $O(n\delta + \lceil n/w \rceil m)$, where w is the number of bits in a computer word.

We number the bits from the least significant bit (0) to the most significant bit ($w - 1$). C–like notation is used for the bit-wise operations of words; & is bit-wise `and`, | is `or`, ~ negates all bits, << is shift to left, and >> shift to right, both with zero padding.

Let us first define a matrix D. Let $D_{i,j} = 1$ if $p_0 p_1 \ldots p_i =^{\alpha}_{\delta} t_h t_{h+1} \ldots t_j$. Otherwise, $D_{i,j} = 0$. This can be expressed as:

$$D_{i,j} = \begin{cases} 1, & p_i =_\delta t_j \text{ and } \exists j' : 0 < j - j' \leq \alpha + 1 \text{ and } D_{i-1,j'} = 1 \\ 0, & \text{otherwise.} \end{cases} \tag{3.8}$$

At a first glance it seems that this recurrence would lead to $O(\alpha nm)$ time. However, we show how to compute $O(w)$ columns in each row of the matrix in $O(1)$ time, independent of α, leading to $O(\lceil n/w \rceil m)$ total time.

To this end, assume that in the preprocessing phase we have computed a helper bit-matrix (whose efficient computation we will consider later) V:

$$V_{i,j} = \begin{cases} 1, & p_i =_\delta t_j \\ 0, & \text{otherwise.} \end{cases} \tag{3.9}$$

The computation of D will proceed column-wise, w columns at once. Each matrix element takes only one bit of storage, so we can store w columns in a single machine word. Assume that we have computed all rows of the columns $(j-1)w \ldots jw-1$, and columns $jw \ldots (j+1)w-1$ up to row $i-1$, and we want to compute the columns $jw \ldots (j+1)w-1$ at row i. Assume also that $\alpha < w$. We adopt the notation $D^w_{i,j} = D_{i,jw \ldots (j+1)w-1}$, and analogously for V. The goal is then to produce $D^w_{i,j}$ from $V^w_{i,j}$, $D^w_{i-1,j}$ and $D^w_{i-1,j-1}$. $D^w_{i,j}$ does not depend on any other D^w element, according to the definition of D, and given our assumption that $\alpha < w$.

Now, according to Eq. (3.8), the kth bit in $D^w_{i,j}$ should be set iff (i) the kth bit in $V^w_{i,j}$ is set (i.e. $p_i =_\delta t_{jw+k}$), and (ii) any of the bits $k-\alpha-1 \ldots k-1$ in $D^w_{i-1,j}$ or any of the bits $k+w-\alpha-1 \ldots w-1$ in $D^w_{i-1,j-1}$ is set (i.e. the gap length to the previous match is at most α). To compute item (ii) efficiently we assume that we have available function $M(x)$:

$$M(x) = M(x, \alpha) = (x << 1) \mid (x << 2) \mid \ldots \mid (x << (\alpha + 1)). \tag{3.10}$$

In other words, $M(x)$ copy-propagates all bits in x to left $1 \ldots \alpha+1$ positions. This means that if the 1 bits in x correspond to the matching positions of a pattern prefix, then $M(x)$ will have those 1 bits aligned in all positions where the matching prefix could be extended. Note that the representation of $M(x)$ needs $w+\alpha+1$ bits, i.e. at most $2w$ bits (two computer words) for $\alpha < w$. We can now write the recurrence for D^w:

$$D^w_{i,j} = V^w_{i,j} \ \& \ (M(D^w_{i-1,j}) \mid (M(D^w_{i-1,j-1}) >> w)). \tag{3.11}$$

Fig. 3.2 illustrates the bits affecting the current row; note that also the case of a negative gap is exposed there.

We are not able to compute $M(x)$ in constant time, hence we use a precomputed look-up table instead. Since w can be too large to make this approach feasible, we can precompute the answers e.g. to only $w/2$ or $w/4$ bit numbers, and correspondingly compute $M(x)$ in 2 or 4 pieces without

affecting the time complexity (in our tests we used $w/2 = 16$ bit numbers for computing $M(x)$).

We also need to compute V efficiently. This is easy with table look-ups as we have an integer alphabet. We first compute a table L, such that for all $c \in \Sigma$ the list $L[c]$ contains all the distinct characters p_i that satisfy $p_i =_\delta c$. Using this table we build a table V', which we will use as a terse representation of V, namely we have that $V'[p_i] = V_i$. This can be done by scanning through the text, and setting the jth bit of the bitvector $V'[c]$ to 1 for each $c \in L[t_j]$. This process takes $O(\lceil n/w \rceil \sigma_{\mathrm{p}} + m + \sigma + \delta\sigma_{\mathrm{p}} + \delta n) = O(\lceil n/w \rceil \sigma_{\mathrm{p}} + \delta n)$ worst-case time. The probability that two characters δ-match is at most $(2\delta + 1)/\sigma$, and hence the expected number of matching pattern characters for each text character is $O(\delta\sigma_{\mathrm{p} \cap \mathrm{t}}/\sigma_{\mathrm{t}})$. Therefore, the average-case complexity of the preprocessing is $O(\lceil n/w \rceil \sigma_{\mathrm{p}} + n(\delta\sigma_{\mathrm{p} \cap \mathrm{t}}/\sigma_{\mathrm{t}} + 1))$. Searching clearly takes only $O(\lceil n/w \rceil m)$ time.

3.8.1 Fast algorithm on average

We make the following observation: if $D_{i \ldots m-1, j-\alpha \ldots j} = 0$, for some i, j, then $D_{i+1 \ldots m-1, j+1} = 0$. This is because there is no way the recurrence can introduce any other value for those matrix cells. In other words, if $p_0 \ldots p_i$ does not (δ, α)-match $t_h \ldots t_{j-k}$ for any $k = 0 \ldots \alpha$, then the match at the position $j + 1$ cannot be extended to $p_0 \ldots p_{i+1}$. This can be utilized by keeping track of the highest row number *top* of the current column j such that $D_{top,j} \neq 0$, and computing the next column only up to row $top + 1$. More formally, we define (for D^w) the maximum row top_j^w for the column j as:

$$top_j^w = \mathrm{argmax}_i \{D_{i-1,j-1}^w \ \& \ a \neq 0 \text{ or } D_{i-1,j}^w \ \& \ ({\sim}0 >> 1) \neq 0\}, \quad (3.12)$$

where the bitmask $a = {\sim}0 << (w - \alpha - 1)$. Consider first the part $D_{i-1,j-1}^w \ \& \ a \neq 0$. The rationale is as follows. When we are computing $D_{i,j}^w$, only the $\alpha + 1$ highest non-zero bits of $D_{i-1,j-1}^w$ can affect the bits in $D_{i,j}^w$. These are selected by the $\& \ a$ operation. However, since we are computing w columns in parallel, the $w - 1$ least significant set bits in $D_{i-1,j}^w$ (the second part), i.e. in the previous row of the *current* set of columns, can affect the bits in $D_{i,j}^w$ as well. Obviously, this second part cannot be computed at column $j - 1$. We solve this simply by computing the first part of top_j^w after the column $j - 1$ has been computed, and when processing the column j, we increase top_j^w if needed according to the second part $(D_{i-1,j}^w \ \& \ ({\sim}0 >> 1) \neq 0)$.

Alg. 22 DA-bpdp$(T, n, P, m, \delta, \alpha)$.

```
1   V ← DA-bpdp-preprocess(T, n, P, m, δ, α)
2   w' ← w/2; msk ← (1 << w') − 1
3   for i ← 0 to (1 << w') − 1 do
4       M[i] ← 0
5       for j ← 0 to α do M[i] ← M[i] | (i << (j + 1))
6   top ← m − 1
7   D_0 ← V[p_0][0]
8   for i ← 1 to top do
9       D_i ← V[p_i][0] & (M[D_{i−1} & msk] | (M[D_{i−1} >> w'] << w'))
10  if D_{m−1} ≠ 0 then report matches
11  for j ← 1 to ⌈n/w⌉ do
12      D'_0 ← V[p_0][j]
13      i ← 1
14      while i ≤ top do
15          x ← M[D'_{i−1} & msk] | (M[D'_{i−1} >> w'] << w')
16          y ← M[D_{i−1} >> w'] >> w'
17          D'_i ← V[p_i][j] & (x | y)
18          if i = top and top < m − 1 and D'_i & (∼0 >> 1) ≠ 0 then
19              D_i ← 0
20              top ← top + 1
21          i ← i + 1
22      if top = m − 1 and D'_{m−1} ≠ 0 then report matches
23      while top > 0 and D'_top & (∼0 << (w − α − 1)) = 0 do top ← top − 1
24      if top < m − 1 then top ← top + 1
25      Dt ← D; D ← D'; D' ← Dt
```

Alg. 22 gives the pseudocode. It uses $w' = w/2$ bits for the precomputed table for the $M(\cdot)$ function. For simplicity, the code also assumes that $\alpha < w'$ (but w columns are still processed in parallel). The average-case running time of this algorithm depends on what is the average value of top^w. For $w = 1$ it can be shown that $\text{avg}(top^1) = O(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}})$ [CCF05b]. This is $O(\alpha\delta/\sigma)$ for $\delta/\sigma < 1 - \alpha^{-1/(\alpha+1)}$, so the average time is $O(n\lceil\alpha\delta/\sigma\rceil)$. We are not able to analyze $\text{avg}(top^w)$ exactly, but we make use of the trivial observation that $\text{avg}(top^1) \le \text{avg}(top^w) \le \text{avg}(top^1) + w - 1$, which lets us conclude that the amortized average search time of Alg. 22 is at most $O(\lceil n/w\rceil\lceil\alpha\delta/\sigma\rceil + n)$.

The $O(\lceil n/w\rceil\sigma_\text{p} + \delta n)$ (worst-case) preprocessing time can be the dominating factor in some cases. We now present an alternative preprocessing variant. The idea is to partition the alphabet into $\lceil\sigma/\delta\rceil$ disjoint intervals of width δ. Let us first redefine V as V^δ:

$$V^\delta_{i,j} = \begin{cases} 1, & |\lfloor p_i/\delta\rfloor - \lfloor t_j/\delta\rfloor| \le 1 \\ 0, & \text{otherwise.} \end{cases} \tag{3.13}$$

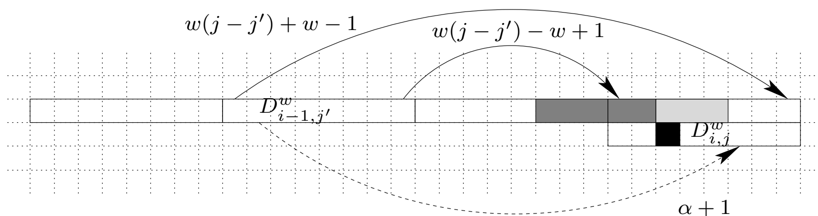

Figure 3.2: (δ,α)-matching. Tiling the dynamic programming matrix with $w \times 1$ vectors ($w = 8$). The black cell of the current tile depends on the dark gray cells of the two tiles in the previous row ($\alpha = 4$). The light gray cells depict a negative gap, together with the dark gray cells the gap is $-3 \ldots 4$. The arrows illustrate some bit distances for the case $\alpha \geq w$

Obviously, if $V_{i,j} = 1$, then also $V^{\delta}_{i,j} = 1$. On the other hand, the converse is not true, i.e. it is possible that $V^{\delta}_{i,j} = 1$ but $V_{i,j} = 0$. This means that we can use V^{δ} in place of V, but the search algorithm becomes a filter, and the candidate occurrences must be verified using some other algorithm, e.g. plain dynamic programming, which makes the total complexity $O(nm)$ in the worst case. However, the benefit is that V^{δ} is very simple to compute, taking only $O(n)$ time. The initialization time drops to $O(\lceil n/w \rceil \min(\sigma_{\mathrm{p}}, \sigma/\delta))$, since it takes $O(\lceil n/w \rceil)$ for each distinct $\lfloor p_i/\delta \rfloor$.

Note that one can use the definition $V^{\delta}_j[p_i/\delta] = 1$ iff $\lfloor p_i/\delta \rfloor = \lfloor t_j/\delta \rfloor$ instead of $V^{\delta}_{i,j}$, and then the fact that $V^{\delta}_{i,j} = V^{\delta}_j[p_i/\delta - 1] \mid V^{\delta}_j[p_i/\delta] \mid V^{\delta}_j[p_i/\delta + 1]$ in the search phase. This speeds up the preprocessing by a constant factor, but slows down the search correspondingly. We use this approach in our experiments.

This can be still improved by interweaving the preprocessing and search phases, so that we initialize and preprocess V^{δ} only for top^w_j length prefixes of the pattern for each j. At the time of processing the column j, we only know top^w_{j-1}, so we use an estimate $\varepsilon \times top^w_{j-1}$ for top^w_j, where $\varepsilon > 1$ is a small constant. If this turns out to be too small, we just increase the estimate and re-preprocess for the current column. The total preprocessing cost on average then becomes only $O(\lceil n/w \rceil \sigma_{top^w} + n)$, where σ_{top^w} is the alphabet size of top^w length prefix of the pattern. Hence the initialization time is at most $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$. We require that $\delta < \sigma/3$, as otherwise the probability of a match becomes 1. The average number of verifications decreases exponentially for $m > \mathrm{avg}(top^w)$, making their cost negligible, so the total preprocessing, filtering and verification time is $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$. For larger δ or smaller m the filter becomes useless.

3.8.2 Handling large α in $O(1)$ time

Alg. 22 assumes that $\alpha < w$. For larger α the time increases by $O(\alpha/w)$ factor, as the gap may span over several machine words. Here we present how to remove this limit while maintaining the $O(1)$ cost for processing w columns.

Let us define *Last Prefix Occurrence*:

$$LPO_{i,j} = \begin{cases} j', & \max j' \leq j \text{ such that } D^w_{i,j'} \neq 0 \\ -\alpha - 1, & \text{otherwise.} \end{cases} \tag{3.14}$$

I.e. for $LPO_{i,j} = j'$, $D^w_{i,j'}$ is the vector that corresponds to the last (δ, α)-match(es) of the prefix $p_0 \ldots p_i$ in the text area $t_0 \ldots t_{wj-1}$. If such vector does not exist (e.g. when $j = 0$) we set $LPO_{i,j} = -\alpha - 1$.

Assume that $\alpha \geq w$ and consider the computation of $D^w_{i,j}$. The recurrence becomes

$$D^w_{i,j} = V^w_{i,j} \ \&\ (M(D^w_{i-1,j}, w) \mid ov). \tag{3.15}$$

The vector ov is computed according to the last prefix occurrence information. Let $j' = LPO_{i-1,j-1}$. We have the following four cases (see also Fig. 3.2):

1. $j' < 0$: no matching prefixes have been found, hence $ov = 0$.
2. $w(j - j') - w + 1 > \alpha + 1$: no bit of $D^w_{i-1,j'}$ can affect any bit in $D^w_{i,j}$, hence we set $ov = 0$.
3. $w(j - j') + w - 1 \leq \alpha + 1$: any set bit in $D^w_{i-1,j'}$ is close enough to affect any bit in $D^w_{i,j}$, hence we set $ov = {\sim}0$.
4. Otherwise some bits of $D^w_{i-1,j'}$ can be close enough to affect some bits of $D^w_{i,j}$, and we set $ov = (M(D^w_{i-1,j'}, \alpha \bmod w) >> w)$.

Note that since $\alpha \geq w$, the function $M(\cdot, \cdot)$ is now much easier to compute. $M(D^w_{i-1,j}, w) = 2^w - 2 \times LSB(D^w_{i-1,j})$, where $LSB(x)$ extracts the least significant set bit of x. The first subtraction operation then propagates the LSB to every higher position as well, while the second subtraction then clears the least significant bit of the result. The solution for $LSB(x)$ is part of the computing folklore, and can be computed as $LSB(x) = (x \ \&\ (x - 1)) \ {}^\wedge\ x$ in $O(1)$ time. Likewise, it is easy to see that $M(D^w_{i-1,j'}, \alpha \bmod w) >> w = 2^s - 1$ for $s = \alpha \bmod w - (w - \left\lfloor \log_2(D^w_{i-1,j'}) \right\rfloor - 1) + 1$, where $\lfloor \log_2(x) \rfloor$ effectively gets the *index* of the most significant set bit of x. In other words, s tells the number of bit positions the most significant bit of $D^w_{i-1,j'}$ propagates to to fill the least significant bits of ov. If $s < 0$, we just set $ov = 0$.

Finally, $LPO_{i,j}$ can be easily maintained in constant time for each i, j. $LPO(i, -1)$ is initialized to $-\alpha - 1$ for all i, which takes $O(m)$ time. Then,

the computation of D^w proceeds column-wise. After $D^w_{i,j}$ is computed, we simply set $LPO_{i,j} = j$ iff $D^w_{i,j} \neq 0$, otherwise we set $LPO_{i,j} = LPO_{i,j-1}$. In practice we can store only the latest value of LPO for each row, so only $O(m)$ space is needed. Hence we can conclude that the value of α does not affect the running time of the algorithm.

3.8.3 Relaxing δ and α

Alg. 22 can be generalized to the case where the gap limit can be of different length for each pattern character [PW05], and where the δ-matching is replaced with character classes, i.e. each pattern character is replaced with a set of characters. More precisely, pattern $p_0p_1p_2\ldots p_{m-1}$, where $p_j \subset \Sigma$, matches $t_{i_0}t_{i_1}t_{i_2}\ldots t_{i_{m-1}}$, if $t_{i_j} \in p_j$ for $j \in \{0,\ldots,m-1\}$, where $a_j \leq i_{j+1} - i_j \leq b_j + 1$, where a_j and b_j are the minimum and maximum gap lengths permitted for a pattern position j. This problem has important applications e.g. in protein searching, see [NR03]. Yet a stronger model [Mye96] allows gaps of *negative* lengths, i.e. a_j (and b_j) can be negative. In other words, parts of the pattern occurrence can be overlapping in the text, see Fig. 3.2. First note that handling character classes is trivial, since it only requires a small change in the computation of V. As for the gaps, consider first the situation where (i) $a_i \geq 0$; or (ii) $b_i \leq 0$. In either case we have $a_i \leq b_i$. Handling the case (i) is just what our algorithm already does. The case (ii) is just the dual of the case (i), and conceptually it can be handled by just scanning the current row from right to left, and using the limits $-b_i - 2, -a_i - 2$ instead of a_i, b_i, and handling the gap -1 as a special case.

The core of Alg. 22 is the use of $M(x)$ (Eq. (3.10)) to select the positions from the previous row where a matching pattern prefix ends. To handle gaps of the form $a_i \geq 0$ we use

$$M_i^{\mathrm{L}}(x) = (x << (a_i+1)) \mid (x << (a_i+2)) \mid \ \ldots \ \mid (x << (b_i+1)). \quad (3.16)$$

For the negative gaps $b_i < 0$ we just align the bits from right, and hence define:

$$M_i^{\mathrm{R}}(x) = (x >> -b_i - 1) \mid (x >> -b_i) \mid \ \ldots \ \mid (x >> -a_i - 1). \quad (3.17)$$

The general case $a_i < 0 \leq b_i$ is handled as a combination of these:

$$M_i(x) = (x >> -a_i - 1) \mid (x >> -a_i) \mid \ \ldots \ \mid (x << (b_i + 1)). \quad (3.18)$$

The final simple modification that we need is to take $D^w_{i-1,j+1}$ into account while computing $D^w_{i,j}$, since the negative gaps may span into it. Hence we modify Eq. (3.11) to:

$$D^w_{i,j} = V^w_{i,j} \quad \& \quad ((M^{\mathrm{L}}_i(D^w_{i-1,j-1}) >> w) \mid M_i(D^w_{i-1,j}) \mid \tag{3.19}$$

$$(M^{\mathrm{R}}_i(D^w_{i-1,j+1} << w) >> w)). \tag{3.20}$$

3.9 Bit-parallel dynamic programming for (δ, γ, α)-matching

We now show how the basic dynamic programming algorithm can be bit-parallelized. The algorithm is based on the bit-parallel dynamic programming algorithm for (δ, α)-matching [FG06c] (see Sect. 3.8). All the interesting values in the matrix D are at most γ, and all other values can be represented as any value greater than γ. Hence $O(\log \gamma)$ bits per matrix cell is sufficient, and we can compute $O(w/\log \gamma)$ cells in parallel, where w is the number of bits in a machine word. Moreover, we show how to handle α up to $O(w/\log \gamma)$ efficiently. Assuming $w = \Theta(\log n)$, we obtain an $O(nm \log(\gamma)/w)$ worst-case time algorithm.

Each matrix cell is represented with

$$\ell = \lceil \log_2(2\gamma + 1) \rceil \tag{3.21}$$

bits, and number zero is represented (using ℓ bits) as $2^{\ell-1} - (\gamma + 1)$. This representation has been used before e.g. for (δ, γ)-matching [CIN+05]. We still need an additional bit per cell, and hence each machine word packs

$$C = \lfloor w/(\ell + 1) \rfloor \tag{3.22}$$

cells, or *counters*. This representation solves three problems we are going to face shortly: (*i*) counter overflows can be handled in parallel; (*ii*) it is easy to check in parallel if some of the counters have exceeded γ; (*iii*) thanks to the additional bit it is easy to compute pair-wise minima over two sets of counters in parallel.

Assume then that in the preprocessing we have computed a helper matrix V:

$$V_{i,j} = \begin{cases} |p_i - t_j|, & p_i =_\delta t_j \\ \gamma + 1, & \text{otherwise.} \end{cases} \tag{3.23}$$

The computation of D will proceed column-wise, C columns at once. We use again the notation $D^w_{i,j} = D_{i,jC\ldots(j+1)C-1}$, and analogously V^C for V, to make the parallelism explicit. Assume now that $\alpha < C$. The goal is

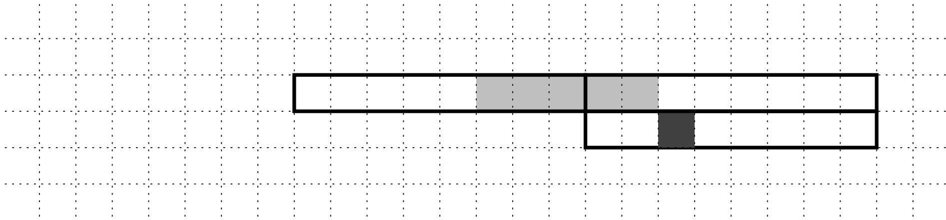

Figure 3.3: (δ, γ, α)-matching. Tiling the dynamic programming matrix with $C = \lfloor w/(\ell+1) \rfloor \times 1$ vectors ($C = 8$). The dark gray cell of the current tile depends on the light gray cells of the two tiles in the previous row ($\alpha = 4$)

then to produce $D^w_{i,j}$ from $V^C_{i,j}$, $D^w_{i-1,j}$ and $D^w_{i-1,j-1}$. $D^w_{i,j}$ does not depend on any other D^w element, according to the definition of D, and given our assumption that $\alpha < C$. Fig. 3.3 illustrates.

Now, according to the recurrence (cf. Eq. (3.5)), the kth counter in $D^w_{i,j}$ is the sum of (i) the kth counter of $V^C_{i,j}$ (i.e. $|p_i - t_{jC+k}|$) and (ii) the minimum of the counters $k - \alpha - 1 \ldots k - 1$ in $D^w_{i-1,j}$ and the counter $k + C - \alpha - 1 \ldots C - 1$ in $D^w_{i-1,j-1}$ (i.e. the gap length to the previous match is at most α), see Fig. 3.3.

To compute item (ii) efficiently we assume that we have available function $M(x)$, that replaces each counter in x with the minima of the $\alpha + 1$ previous counters in x. The recurrence for D^w then becomes:

$$D^w_{i,j} = V^C_{i,j} + (M((D^w_{i-1,j} << w) \mid (D^w_{i-1,j-1} << (w - w \ \% \ C))) >> w), \tag{3.24}$$

where for simplicity we have assumed that $M(x)$ can handle words of length $2w$. However, the above equation may cause counter overflow. To prevent this we use

$$D^w_{i,j} = (V^C_{i,j} + (M' \ \& \sim\! hmsk)) \mid (M' \ \& \ hmsk) \tag{3.25}$$

instead, where

$$M' = M((D^w_{i-1,j} << w) \mid (D^w_{i-1,j-1} << (w - w \ \% \ C))) >> w, \tag{3.26}$$

and $hmsk$ selects the ℓth bit of each counter. That is, $M' \ \& \sim\! hmsk$ clears the highest bit of each counter, so that the result can be safely added to $V^C_{i,j}$, and then $\mid (M' \ \& \ hmsk)$ restores the highest bit. This works correctly, as if the highest bit was set, then the sum is certainly greater than γ, and its exact value is not interesting anymore. The $(\ell + 1)$th bit is not affected by the summation as the maximum value added is $\gamma + 1$.

Finally, to detect the possible pattern occurrences we must add our representation of zero ($2^{\ell-1} - (\gamma+1)$) to each counter. If some of the counters have still not overflowed, the corresponding text positions match. This can be detected as

$$q = \sim(((D^w_{m-1,j} \ \& \ \sim hmsk) + zeromsk) \mid D^w_{m-1,j}) \ \& \ hmsk, \qquad (3.27)$$

where $zeromsk$ has the value $2^{\ell-1} - (\gamma+1)$ in each counter position. Each set bit in q then indicates a pattern occurrence.

Consider now the computation of $M(x)$. One possible solution is to use table look-ups to compute it in constant time. Since w can be too large to make this approach feasible, we can precompute the answers e.g. to only $w/2$ or $w/4$ bit numbers, and correspondingly compute $M(x)$ in 2 or 4 pieces without affecting the time complexity (in our tests we used at most $w/2 = 16$ bit numbers for computing $M(x)$).

Another solution is to use repeated shifting and minimization. That is, assuming that $\text{vmin}(x, y)$ computes pair-wise minima of the counter sets x and y, we compute $x \leftarrow \text{vmin}(x, (x << (\ell+1)) \mid (\gamma+1))$ and repeat that α times, and then perform the final shift $x \leftarrow x << (\ell+1)$, which gives the desired result. The minimization can be done in $O(1)$ time [PS80], see Alg. 23. The total time for computing $M(x)$ is then $O(\alpha)$. This can be easily improved to $O(\log(\alpha))$. Without loss of generality assume that α is a power of two. Instead of shifting one counter position at a time we first shift by $\alpha/2$ counter positions, then $\alpha/4$ counter positions, and so on $\log_2(\alpha)$ times, performing the minimization at each step. (Note that if α is not a power of two, the number of shifts in the procedure will grow at most (almost) twice, which can be accomplished by caching temporary results.) Alg. 24 shows the code, handling the general case as well. This algorithm takes the counter sets $D^w_{i-1,j}$ and $D^w_{i-1,j-1}$, that can affect the current counters $D^w_{i,j}$, as parameter. For simplicity these are handled as a concatenated single word of $2w$ bits. Eq. (3.26) then becomes

$$M' = M(D^w_{i-1,j}, D^w_{i-1,j-1}, \alpha, msk), \qquad (3.28)$$

where msk has every $(\ell+1)$th bit set, needed at the counter minimization.

We also need to compute V efficiently. Again, we make use of table look-ups for that, as we deal with an integer alphabet. First we build a table L, such that for all $c \in \Sigma$ the list $L[c]$ contains all the distinct characters p_i that satisfy $p_i =_\delta c$. Next we build a a table V', which will be used as a terse representation of V, namely we have that $V'[p_i] = V_i$. This can be

Alg. 23 vmin(x, y, msk)

```
1    F ← ((x | msk) − y) & msk
2    F ← F − (F >> ℓ)
3    return (x & ∼F) | (y & F)
```

Alg. 24 $M(x, y, \alpha, msk)$.

```
1    x ← (x << w) | (y << (w − w % C))
2    while α ≠ 0 do
3        r ← α % 2
4        α ← ⌊α/2⌋
5        x ← vmin(x, x << ((ℓ + 1)α), msk)
6        if r = 0 then continue
7        x ← vmin(x, x << (ℓ + 1), msk)
8    return (x << (ℓ + 1)) >> w
```

done by scanning through the text, and setting the jth counter of $V'[c]$ to $|c - t_j|$ for each $c \in L[t_j]$. This process takes $O(\lceil n/C \rceil \sigma_{\rm p} + m + \sigma + \delta\sigma_{\rm p} + \delta n) = O(\lceil n/C \rceil \sigma_{\rm p} + \delta n)$ worst-case time. Using a known argument (cf. Sect. 3.8) we have that the expected number of matching pattern characters for each text character is $O(\delta\sigma_{\rm p}/\sigma)$. Therefore, the average-case complexity of the preprocessing is $O(\lceil n/C \rceil \sigma_{\rm p} + n(\delta\sigma_{\rm p}/\sigma + 1))$. Searching takes only $O(\lceil n/C \rceil m) = O(\lceil n \log(\gamma)/w \rceil m)$ time if table look-ups are used for computing $M(x)$, and $O(\lceil n \log(\alpha) \log(\gamma)/w \rceil m)$ if Alg. 24 is used. For α larger than $O(w/\log(\gamma))$ the search time must be multiplied by $O(\lceil \alpha \log(\gamma)/w \rceil)$.

3.9.1 Cut-off

The cut-off trick used in Sect. 3.3 obviously works for the bit-parallel algorithm as well. More formally, we define (for D^w) the maximum row top_j^C for the column j as:

$$\begin{aligned} top_j^C = \text{argmax}_i\{ & (\text{MatchMsk}(D^w_{i-1,j-1}) >> ((C - \alpha - 1)(\ell + 1))) \neq 0 \text{ or} \\ & (\text{MatchMsk}(D^w_{i-1,j}) << (\ell + 1)) \neq 0\}, \end{aligned} \quad (3.29)$$

where

$$\text{MatchMsk}(x) = \sim(((x \;\&\; \sim hmsk) + zeromsk) \mid x) \;\&\; hmsk. \quad (3.30)$$

Consider first the part (3.29). The rationale is as follows. When we are computing $D^w_{i,j}$, only the last $\alpha + 1$ counters of $D^w_{i-1,j-1}$ that are at most γ can affect the counters in $D^w_{i,j}$. We therefore select the corresponding counter bits that indicate whether or not the sum has exceeded γ. However,

since we are computing C columns in parallel, the $C-1$ first counters that have a value of at most γ in $D^w_{i-1,j}$ (3.29), i.e. in the previous row of the *current* set of columns, can affect the counters in $D^w_{i,j}$ as well. Obviously, this second part cannot be computed at column $j-1$. We solve this simply by computing the first part of top^C_j after the column $j-1$ has been computed, and when processing the column j, we increase top^C_j if needed according to the second part (3.29).

Alg. 25 gives the pseudocode. It uses the $O(\log(\alpha))$ time algorithm for the $M(\cdot)$ function. The average-case running time of this algorithm depends on what is the average value of top^C. For $C=1$ and $\gamma=\infty$ $\text{avg}(top^1) = O\left(\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}}\right)$, see Sect. 3.3. We are not able to analyze $\text{avg}(top^C)$ exactly, but we have trivially that $\text{avg}(top^1) \le \text{avg}(top^C) \le \text{avg}(top^1)+C-1$, and hence the amortized average search time of Alg. 25 is at most $O((\lceil n/C\rceil\lceil\alpha\delta/\sigma\rceil+n)\log(\alpha))$. The $\log(\alpha)$ factor can be easily removed with precomputation.

Constant time initialization of V

The initialization of V (to set all values to $\gamma+1$) can be done in $O(1)$ time using the trick described in [Meh84, Sect. III 8.1]. This requires using two auxiliary arrays, but the asymptotic space complexity does not change, and V can be still accessed in $O(1)$ time. This removes the $O(\lceil n/C\rceil\sigma_\text{p})$ term from the preprocessing time, which could otherwise dominate the search time of the cut–off algorithm.

3.9.2 Lazy preprocessing

This can be still improved by interweaving the preprocessing and search phases, so that we initialize and preprocess V^C only for top^C_j length prefixes of the pattern for each j. At the time of processing the column j, we only know top^C_{j-1}, so we use an estimate $\varepsilon\times top^C_{j-1}$ for top^C_j, where $\varepsilon>1$ is a small constant. If this turns out to be too small, we just increase the estimate and re-preprocess for the current column. The total preprocessing cost on average then becomes only $O(\lceil n/C\rceil\sigma_{top^C}\delta/\sigma+n)$, where σ_{top^C} is the alphabet size of top^C length prefix of the pattern. Hence the initialization time is at most $O(\lceil n/C\rceil\lceil\alpha\delta/\sigma\rceil+n)$ on average. This matches the search time, and together with the preprocessing the total is $O(\lceil n/C\rceil\lceil\alpha\delta/\sigma\rceil+n\lceil\alpha\delta/\sigma\rceil\delta/\sigma+n)$ on average.

Alg. 25 DGA-bpdp$(T, n, P, m, \delta, \gamma, \alpha)$.

1 $\ell \leftarrow \lceil \log_2(2\gamma + 1) \rceil$
2 $f \leftarrow (w/(\ell + 1))$
3 $zmsk \leftarrow (1 << (\ell + 1)) - 1$
4 **for** $i \leftarrow 0$ **to** $\sigma - 1$ **do** $A[i] \leftarrow 0$
5 **for** $i \leftarrow 0$ **to** $m - 1$ **do**
6   **if** $A[p_i]$ **then continue**
7     $A[p_i] \leftarrow 1$
8   **for** $j \leftarrow \max\{0, p_i - \delta\}$ **to** $\min\{p_i + \delta, \sigma - 1\}$ **do**
9         $Lt[j] \leftarrow Lt[j] \cup \{p_i\}$
10 $zero \leftarrow (1 << (l - 1)) - (\gamma + 1)$
11 $hhmsk \leftarrow 0$
12 **for** $i \leftarrow 0$ **to** $f - 1$ **do** $hhmsk \leftarrow hhmsk \mid (1 << ((i + 1)(\ell + 1) - 1))$
13 $hmsk \leftarrow hhmsk >> 1$
14 $b \leftarrow (n + f - 1)/f$
15 **for** $i \leftarrow 0$ **to** $\sigma - 1$ **do**
16     $V[i] \leftarrow 0$
17   **if** $A[i] \neq 0$ **then**
18   **for** $j \leftarrow 0$ **to** $b - 1$ **do** $V[i][j] \leftarrow hmsk$
19 **for** $j \leftarrow 0$ **to** $n - 1$ **do**
20   **for** $i \leftarrow 0$ **to** $|Lt[t_j]| - 1$ **do**
21         $c \leftarrow Lt[t_j][i]$
22         $V[c][j/f] \leftarrow V[c][j/f]\ \&\ {\sim}(zmsk << ((j \% f)(\ell + 1)))$
23         $V[c][j/f] \leftarrow V[c][j/f] \mid (|c - t_j| << ((j \% f)(\ell + 1)))$
24 $top \leftarrow m - 1$
25 $D[0] \leftarrow V[p_0][0]$
26 **for** $i \leftarrow 1$ **to** top **do**
27     $x \leftarrow M(D[i - 1], hmsk, \alpha, hhmsk)$
28     $D[i] \leftarrow (V[p_i][0] + (x\ \&\ {\sim}hmsk)) \mid (x\ \&\ hmsk)$
29 $zeromsk \leftarrow 0$
30 **for** $i \leftarrow 0$ **to** $f - 1$ $zeromsk \leftarrow zeromsk \mid (zero << (i(\ell + 1)))$
31 $x \leftarrow {\sim}(((D[m - 1]\ \&\ {\sim}hmsk) + zeromsk) \mid D[m - 1])\ \&\ hmsk$
32 **if** $x \neq 0$ **then** report matches
33 $k \leftarrow ((f - \alpha - 1)(\ell + 1))$
34 **for** $j \leftarrow 1$ **to** $b - 1$ **do**
35     $D'[0] \leftarrow V[p_0][j]$
36   **if** $top = 0$ **then**
37   **if** $({\sim}(((D'[0]\ \&\ {\sim}hmsk) + zeromsk) \mid D'[0])\ \&\ hmsk) \neq 0$ **then**
38             $D[0] \leftarrow hmsk$; $top \leftarrow top + 1$
39   **for** $i \leftarrow 1$ **to** top **do**
40         $x \leftarrow M(D'[i - 1], D[i - 1], \alpha, hhmsk)$
41         $D'[i] \leftarrow (V[p_i][j] + (x\ \&\ {\sim}hmsk)) \mid (x\ \&\ hmsk)$
42         $x \leftarrow {\sim}(((D'[i]\ \&\ {\sim}hmsk) + zeromsk) \mid D'[i])\ \&\ hmsk$
43   **if** $i = top$ and $top < m - 1$ and $(x << (\ell + 1)) \neq 0$ **then**
44             $D[i] \leftarrow hmsk$; $top \leftarrow top + 1$
45   **if** $top = m - 1$ and $x \neq 0$ **then** report matches
46   **do** $x \leftarrow ({\sim}(((D'[top]\ \&\ {\sim}hmsk) + zeromsk) \mid D'[top])\ \&\ hmsk) >> k$
47   **if** $x = 0$ **then** $top \leftarrow top - 1$
48   **while** $top \geq 0$ and $x = 0$
49   **if** $top < m - 1$ **then** $top \leftarrow top + 1$
50     $Dt \leftarrow D; D \leftarrow D'; D' \leftarrow Dt$

3.9.3 Improving the worst case for large α

The bit-parallel algorithm can handle C cells in $O(1)$ time only if $\alpha = O(w/\log(\gamma))$. Otherwise the time becomes $O(\lceil \alpha \log(\gamma)/w \rceil)$. Our aim is to reduce this in the case of $w/\log(\gamma) < \alpha < \gamma^2$, which is our precondition for this subsection. However, we still process just C cells in parallel. For simplicity we also assume that $\alpha = O(w/\log(\alpha/\gamma))$, as this will allow parallel processing of C cells in $O(1)$ time.

To improve the worst case we first note that we do not need to know the sum of errors for all the past $\alpha + 1$ cells, but only those that are "non-dominated". In other words, if we are computing $D_{i,j}$, then the value $D_{i-1,j'}$, is interesting only if $D_{i-1,j'} \leq \gamma$ and there is no cell $(i-1, j'')$ such that $D_{i-1,j''} \leq D_{i-1,j'}$ and $0 < j - j'' < j - j' \leq \alpha + 1$. In principle this allows us to pack the error sums more succinctly. There are, however, several problems to be solved before we can put this idea into good use. The first is that this idea cannot be applied straightforwardly for our bit-parallel algorithm. I.e. assume that we want to compute $D^w_{i,j}$. This depends on $D^w_{i-1,j}$, but the cells in this chunk can be both dominated and non-dominated, depending on which cell of $D^w_{i,j}$ we are looking at. Hence we are going to handle $D^w_{i-1,j}$ as a special case. However, this is not a problem for $D^w_{i-1,j'}$, where $j' < j$, as then there is no ambiguity with respect to the cells in $D^w_{i,j}$.

This means that we can keep the difference sum positions ordered according to the distance from the left-most cell of $D^w_{i,j}$, and only store the positions according to increasing sums, in a differential representation. There are only at most $\gamma + 1$ such interesting cells, i.e. different sums of errors. The differences between the positions of them can be encoded e.g. with Elias coding [Eli75] and we obtain amortized $O(\log(\alpha/\gamma))$ bits for each, in the worst case. Therefore we can encode $O(w/\log(\alpha/\gamma))$ non-dominated error sums into a single computer word, which is an improvement over the previous representation if $\alpha < \gamma^2$. Note that not all the $\gamma + 1$ distinct error values may occur in the window. We encode a dummy position offset of 0 for these sums to indicate that they are not present.

We now show how this encoding is used. Assume that we are processing the chunk $D^w_{i,j}$. The chunk $D^w_{i-1,j}$ is processed exactly as before, i.e. it is given as an input to the function $M(\cdot)$ to obtain (in $O(1)$ time) a bit-vector which we call u from now on. Assume that we have another matrix Γ that packs the error sums as described above. More precisely, $\Gamma_{i,j}$ packs the non-dominated error sums corresponding to $D^w_{i-1,j-1}, D^w_{i-1,j-2}, \ldots, D^w_{i-1,j-\lceil \alpha/C \rceil}$. If we make the assumption that $\Gamma_{i,j}$ has only w bits, we can use precomputed tables (see below) to implement a function $M^\Gamma(\cdot)$, that is similar to $M(\cdot)$

used before. That is, $M^\Gamma(\Gamma_{i,j})$ gives a bit-vector v, that is, minimized and copy-propagated error sums (to $\alpha+1$ positions), exactly as before. Then to compute the value of $D^w_{i,j}$ we use Alg. 23 to compute the pair-wise minima of the fields of u and v, i.e. we execute $g = \text{vmin}(u, v, msk)$. This takes $O(1)$ time. Then we can just use g in place of M' (see Eq. (3.26)) in Eq. (3.25).

This leaves only one loose end to be resolved. That is, we must somehow obtain $\Gamma_{i,j+1}$ before we continue. This depends on $\Gamma_{i,j}$ we just used, and on $D^w_{i,j}$ we just obtained using it. Our solution is to simply use precomputation again. I.e. we use $\Gamma_{i,j+1} = U[\Gamma_{i,j}, D^w_{i,j}]$. In practice w is too large, so as in the case of precomputed $M(\cdot)$ function, we use e.g. $w/2$ or $w/4$ and compute the answers in several (constant number of) pieces. Theoretically, in unit cost RAM model of computation $\log(n) = O(w)$, and we can use e.g. bit-vectors of length $\frac{1}{2}\log(n)$ to compute M^Γ in two pieces, and similarly $\frac{1}{4}\log(n)$ to implement U. This results in tables of size $O(\sqrt{n})$ only.

In total the worst-case time of the algorithm is still $O(nm\log(\gamma)/w)$, but this time the bound holds for much larger values of α. The technique obviously works for the Cut-off variant equally well.

3.9.4 Multiple patterns

The algorithm has relatively high preprocessing cost $O(\delta n)$ (or $O(\delta n + \sigma_{\text{p}}\lceil n/C\rceil)$ without the fast initialization technique) in the worst case. However, if we want to search for a set of r patterns, instead of only one pattern, the preprocessing remains essentially the same, since it depends only on the text and the pattern alphabet. The total (worst-case) preprocessing time increases only to $O(\delta n + rm)$, where we have pessimistically considered that m is the length of the longest pattern in the set. The search times have to be multiplied by r, but the amortized preprocessing cost per pattern is considerably reduced. If r is small as compared to σ/δ, the search cost can be reduced by "superimposing" the patterns, that is we define

$$V_{i,j} = \begin{cases} \min|p - t_j|, & p =_\delta t_j \text{ and } p \in p^h_i, h \in 0 \ldots r-1 \\ \gamma + 1, & \text{otherwise,} \end{cases} \tag{3.31}$$

where we use the notation p^h_i to denote the ith symbol of the hth pattern. We then need only one search, but the potential matches must be verified. Superimposing works for the other algorithms as well.

3.9.5 Filtering

Alg. 25 is substantially more complex than its ancestor, the (δ, α)-matching algorithm [FG06c]. In addition to being simpler, the previous algorithm achieves greater parallelism, as it search phase takes $O(\lceil n/w \rceil m)$ time in the worst case. However, we note that this algorithm (as any (δ, α)-matching algorithm) can be used as a filter, since it implicitly assumes that $\gamma = \infty$. The potential occurrences have to verified, which can be done using any of the algorithms given in this chapter. The worst-case time then becomes that of the verification algorithm.

3.10 Non-deterministic finite automata for (δ, α)-matching

In this section we present an algorithm for (δ, α)-matching based on non-deterministic finite automaton. We review our worst-case optimized algorithm [FG06c] only, but in [FG08b] we also reduced its average-case running time. Our algorithm improves the solution from [NR03], where the proposed automaton required $m + (m-1)\alpha$ bits to represent the search state. We managed to reduce this to $O(m \log(\alpha))$ bits, and hence the worst-case time to $O(n\lceil (m \log(\alpha))/w \rceil)$.

At a high level, the algorithm can be seen as a novel combination of Shift-And and Shift-Add algorithms [BYG92]. The 'automaton' has two kinds of states: Shift-And states and Shift-Add states. The Shift-And states keep track of the pattern characters, while the Shift-Add states keep track of the gap length between the characters. The result is a systolic array rather than automaton; a high level description of a building block for character p_i is shown in Fig. 3.4. The final array is obtained by concatenating one building block for each pattern character. We call the building blocks *counters*.

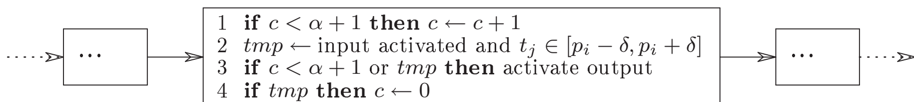

Figure 3.4: A building block for a systolic array detecting δ-matches with α-bounded gaps.

To efficiently implement the systolic array in sequential computer, we need to represent each counter with as few bits as possible while still being able to update all the counters bit-parallelly.

We reserve $\ell = \lceil \log_2(\alpha+1) \rceil + 1$ bits for each counter, and hence we can store $\lfloor w/\ell \rfloor$ counters into a single machine word. We use the value $2^{\ell-1} - (\alpha+1)$ to initialize the counters, i.e. to represent the value 0. (This representation has been used before, e.g. in [CIN+05].) This means that the highest bit (ℓth bit) of the counter becomes 1 when the counter has reached a value $\alpha+1$, i.e. the gap cannot be extended anymore. Hence the lines 3–4 of the algorithm in Fig. 3.4 can be computed bit-parallelly as

$$C \leftarrow C + ((\sim C >> (\ell - 1)) \ \& \ msk),$$

where msk selects the lowest bit of each counter. That is, we negate and select the highest bit of each counter (shifted to the low bit positions), and add the result to the original counters. If a counter value is less than $\alpha+1$, then the highest bit position is not activated, and hence the counter gets incremented by one. If the bit was activated, we effectively add 0 to the counter.

To detect the δ-matching characters we need to preprocess a table B, so that $B[c]$ has $i\ell$th bit set to 1, iff $|p_i - c| \le \delta$. We can then use the plain Shift-And step:

$$D' \leftarrow ((D << \ell) \mid 1) \ \& \ B[t_j],$$

where we have reserved ℓ bits per character in D as well. Only the lowest bit of each field has any significance, the rest are only for aligning D and C appropriately. The reason is that a state in D may be activated also if the corresponding gap counter has not exceeded $\alpha+1$. In other words, if the highest bit of a counter in C is not activated (the gap condition is not violated), then the corresponding bit in D should be activated:

$$D \leftarrow D' \mid ((\sim C >> (\ell - 1)) \ \& \ msk).$$

The only remaining difficulty to solve is how to reinitialize (bit-parallelly) some subset of the counters to zero, i.e. how to implement the lines 1–2 of the algorithm in Fig. 3.4. The bit vector D' has value 1 in every field position that survived the Shift-And step, i.e. in every field position that needs to be initialized in C. Then

$$C \leftarrow C \ \& \ \sim(D' \times ((1 << \ell) - 1))$$

$$C \leftarrow C \mid (D' \times ((1 << (\ell - 1)) - (\alpha + 1)))$$

first clears the corresponding counter fields, and then copies the initial value $2^{\ell-1} - (\alpha+1)$ to all the cleared fields.

Alg. 26 DA-NFA$(T, n, P, m, \delta, \alpha)$.

```
1   ℓ ← ⌈log₂(α + 1)⌉ + 1
2   for i ← 0 to σ − 1 do B[i] ← 0; B′[i] ← 0
3   for i ← 0 to m − 1 do B′[p_i] ← B′[p_i] | (1 << (i × ℓ))
4   for i ← 0 to σ − 1 do if B′[i] ≠ 0 then
5       for j ← max(0, i − δ) to min(i + δ, σ − 1) do B[j] ← B[j] | B′[i]
6   msk ← 0
7   for i ← 0 to m − 1 do msk ← msk | (1 << (i × ℓ))
8   am ← (1 << (ℓ − 1)) − (α + 1)
9   D ← 0; C ← (am + α + 1) × msk
10  msk ← msk >> ℓ
11  mm ← 1 << ((m − 1) × ℓ)
12  for i ← 0 to n − 1 do
13      C ← C + ((~C >> (ℓ − 1)) & msk)
14      D′ ← ((D << ℓ) | 1) & B[t_i]
15      D ← D′ | ((~C >> (ℓ − 1)) & msk)
16      C ← C & ~((D′ << ℓ) − D′)
17      C ← C | (D′ × am)
18      if (D & mm) = mm then report match
```

This completes the algorithm. Alg. 26 gives the pseudocode. Alg. 26 runs in $O(n)$ worst-case time, if $m(\lceil \log_2(\alpha+1) \rceil + 1) \leq w$. Otherwise, several machine words are needed to represent the search state, and the time grows accordingly. However, by using the well-known folklore idea, it is possible to obtain $O(n)$ average time for long patterns not fitting into a single word by updating only the "active" (i.e. non-zero) computer words. This works in $O(n)$ time on average as long as $\delta/(\sigma(1-\delta/\sigma)^{\alpha+1}) = O(w/\log(\alpha))$. The preprocessing takes $O(m + (\sigma + \delta\sigma_{\mathrm{p}})\lceil m\log(\alpha)/w \rceil)$ time, which is $O(m + (\sigma + \delta\min\{m, \sigma\})\lceil m\log(\alpha)/w \rceil)$ in the worst case.

Improving the average-case complexity of our algorithm is based on the idea [NR03] of combining the forward matching automaton with BNDM [NR00]. The achieved average time is $O(n\alpha\log_{1/\rho}(m)/m)$ if $m\log(\alpha) = O(w)$, where ρ is the probability of a character match and $\rho = 1 - (1 - \delta/\sigma)^{\alpha+1}$. For general (large) m, we obtain $O(n\alpha\log(\alpha)\log_{1/\rho}(m)/w)$ time. For $m > w/\log(\alpha)$ we could also use only pattern prefixes of length $w/\log(\alpha)$ in the backward search phase, resulting in $O(n\alpha\log(\alpha)\log_{1/\rho}(w/\log(\alpha))/w)$ average time, which is a slight improvement. The worst-case time of this algorithm becomes quadratic. However, there are some "standard tricks" that can be applied to combine the backward and forward (verification) scans so that either scans no text character twice [CCG$^+$94, NR03]. These work with our algorithm as well, letting it preserve the $O(n\lceil(m\log(\alpha))/w\rceil)$ worst-case complexity.

3.11 Other models

As mentioned in the beginning of this chapter, there exist close relations between some relevant approximate matching models in music information retrieval and molecular biology. Namely, our algorithms for the (δ, α)-matching in the MIR setting can be straightforwardly translated (generalized) to a protein search application, where the equivalent of δ-tolerance are classes of characters, and gaps can be of different length range between any pair of pattern characters. What is more, most of our sparse dynamic programming and bit-parallel techniques presented there can work even for negative gaps, a problem variant which seemed hard earlier.

We have also considered a seemingly MIR specific problem variant, with the extra parameter γ, being the maximum allowed sum of individual symbol errors. Still, perhaps unexpectedly, we show how our techniques for handling γ can be simply translated to the problem of matching with (arbitrary) gaps, character classes and up to k mismatches. We believe that our techniques are especially valuable for the case of negative gaps.

We now describe all of these extensions in turn. Most of the extensions apply to both Simple and the bit-parallel variants.

3.11.1 Handling character classes

In the case of character classes $p_i \subset \Sigma$, and t_j matches p_i if $t_j \in p_i$. For the Simple algorithms we can preprocess a table $C[0 \ldots m-1][0 \ldots \sigma-1]$, where $C[i][c] := c \in p_i$. This requires $O(\sigma m)$ space and $O(\sigma \sum_i |p_i|)$ time, which is attractive for small σ, such as protein alphabet. The search algorithm can then use C to check if $t_j \in p_i$ in $O(1)$ time. For large alphabets we can use e.g. hashing or binary search, to do the comparisons in $O(1)$ or in $O(\log |p_i|)$ time, respectively.

The variant of Simple where we preprocess $\mathcal{M}$ is a bit more complicated. We first compute lists $L'[c] = \{i \mid c \in p_i\}$ in the preprocessing phase. This can be done in one linear scan over the pattern. Then list $L[i]$ is defined as $L[i] = \{j \mid t_j \in p_i\}$. This can be computed in one linear scan over the text appending j into each list $L[i]$ where $i \in L'[t_j]$. The total time is then $O(n\delta)$, where we can consider δ as the average size of the character classes. Lists $L[i]$ for $i \in \{0 \ldots m-1\}$ then correspond to the set $\mathcal{M}$, which is used in the same way as previously.

The change for the bit-parallel algorithm is even simpler, and we can easily allow character classes for both in the pattern and the text. I.e. both the pattern and text symbols can be subsets of the alphabet, that

is, $p_i, t_j \subseteq \Sigma$. The search algorithm does not change, we just change the definition (and preprocessing) of V:

$$V_{i,j} = \begin{cases} \min |p - t|, & p =_\delta t \text{ and } p \in p_i, t \in t_j, \\ \gamma + 1, & \text{otherwise.} \end{cases} \qquad (3.32)$$

3.11.2 Matching with general gaps

We now discuss gaps of the form $g(a_i, b_i)$, where a_i denotes the minimum and b_i the maximum ($a_i \leq b_i$) gap length for the pattern position i. This problem variant has important applications e.g. in protein searching, see [MM91, Mye96, NR03]. General gaps were considered in [NR03, PW05]. This extension is easy to handle in all our algorithms, i.e. it is equally easy to check if the formed gap length satisfies $g(a_i, b_i)$ as it is to check if it satisfies $g(0, \alpha)$. The column-wise sparse dynamic programming is a bit trickier, but still adaptable. Yet a stronger model [MM91, Mye96] allows gaps of *negative* lengths, i.e. the gap may have a form $g(a_i, b_i)$ where $a_i < 0$ (it is also possible that $b_i < 0$). In other words, parts of the pattern occurrence can be overlapping in the text.

Consider first the situation where for each $g(a_i, b_i)$: (*i*) $a_i \geq 0$; or (*ii*) $b_i \leq 0$. In either case we have $a_i \leq b_i$. Handling the case (*i*) is just what our algorithms already do. The case (*ii*) is just the dual of the case (*i*), and conceptually it can be handled in any of our dynamic programming algorithms by just scanning the current row from right to left, and using $g(-b_i - 2, -a_i - 2)$ instead of $g(a_i, b_i)$.

The general case where we also allow $a_i < 0 < b_i$ is slightly trickier. Basically, the only modification for Alg. 17 is that we change all the conditions of the form $0 \leq g \leq \alpha$, where g is the formed gap length for the current position, to form $a_i \leq g \leq b_i$. Note that this does not require any backtracking, even if $a_i < 0$.

The basic Simple algorithm can be adapted as follows (we present it for the (δ, γ, α) variant). For computing the list L_i, the basic algorithm checks if any of the text characters $t_{j'+1} \ldots t_{j'+\alpha+1}$, for $j' \in L_{i-1}$ matches p_i. We modify this to check the text characters $t_{j'+a_i+1} \ldots t_{j'+b_i+1}$. This clearly handles correctly both the situations $b_i \leq 0$ and $a_i < 0 < b_i$. The scanning time for row i becomes now $O((b_i - a_i + 1)|L_{i-1}|)$. The average time is preserved as $O(n)$ if we now require that $(b_i - a_i + 1)\delta/\sigma < 1$. The optimization to detect and avoid overlapping text windows clearly works in this setting as well, and hence the worst-case time remains $O(n + \min\{(b - a + 1)|\mathcal{M}|, mn\})$, where for simplicity we have considered that the gaps are of the same size for all rows.

As for the Simple variant that precomputes $\mathcal{M}$, basically the only modification is that we change all the conditions of the form $0 \leq g \leq \alpha$, where g is the formed gap length for the current position, to form $a_i \leq g \leq b_i$. Note that this does not require any backtracking, even if $a_i < 0$. In other words, as we use either binary search or priority queues, we can just change the restriction of the text area where the δ-matches should occur.

The bit-parallel algorithm (again, we talk about the (δ, γ, α) variant, since the (δ, α) variant is easier and implies immediately) can be adapted as follows. The basic method uses the function M to select (and minimize) the dynamic programming matrix cells that affect the sum of differences. Thus, the only modification we need is to select the cells according to the new gap definition. In fact, this implies that the gap do not have to be even a continuous range, but any cells can be selected if M is implemented using preprocessed tables. For negative gaps we must compute the matrix row-wise (as opposed to the current column-wise method), since now $D^w_{i,j}$ depends on $D^w_{i-1,j+1}$, which is not available if the matrix is computed column-wise. This also means that the cut-off version cannot be used.

Note that this extension works with character classes just as well in all algorithms.

3.11.3 Matching with k mismatches

The same technique we have used for handling the γ condition can be used to solve the case where we count the number of mismatching character positions (and still allowing gaps and δ-matching or character classes). This is more interesting variant in the case of protein matching, for instance.

Basically, we can just assume that the difference counters are incremented by 1 if the two symbols do not δ-match (similarly for character classes), and by 0 otherwise, and we require that the counter must not exceed a parameter k. All the previous time bounds are preserved if we just replace γ with k. For example, in the bit-parallel algorithm the counters do not take $O(\log \gamma)$ bits anymore, but $O(\log k)$ bits instead. See also Sect. 3.11.4, where we make this idea more explicit.

3.11.4 $(\delta, k_\Delta, \alpha)$-matching

The presented algorithms can be used to solve several other problem variants as well. One example is what we call $(\delta, k_\Delta, \alpha)$-matching. In this model we assume that $\gamma = \infty$, i.e. the δ-errors can accumulate to any value (actually

the obvious upper bound is δm), but we also allow up to k_Δ "outliers" (mismatches) with the additional restriction that $|p_i - t_j| \leq \Delta$. We also assume that $\delta < \Delta$. In other words, if $\delta < |p_i - t_j| \leq \Delta$, then we increment the outlier counter by 1. If $|p_i - t_j| > \Delta$, the outlier counter is incremented by $k_\Delta + 1$ (i.e. the pattern does not match).

Consider the bit-parallel algorithm. For the parameter k_Δ we need to preprocess V as follows:

$$V_{i,j}^{k_\Delta} = \begin{cases} 0, & p_i =_\delta t_j, \\ 1, & p_i \neq_\delta t_j \text{ and } p_i =_\Delta t_j, \\ k_\Delta + 1, & \text{otherwise.} \end{cases} \tag{3.33}$$

The computation then proceeds as before, we update the matrix D^{k_Δ} exactly as we did with the basic algorithm. To detect a match we just require that the k_Δ condition holds. The search complexity of the algorithm is clearly $O(nm \log(k_\Delta)/w)$ in the worst case. Analogously, the Simple and DP Cut-off algorithms can be adapted to this setting.

There are some interesting special cases of this model, such as $(0, k_\Delta, \alpha)$-matching, and the matching with k mismatches (Sect. 3.11.3) which corresponds to $(\delta, k_\sigma, \alpha)$-matching.

3.12 Transposition invariance

In this section we consider transposition invariance together with (δ, α)-matching. In this case pattern P (δ, α)-matches the text substring $t_{i_0} t_{i_1} t_{i_2} \ldots t_{i_{m-1}}$, if $p_j + \tau =_\delta t_{i_j}$ for $j \in \{0, \ldots, m-1\}$, where $i_j < i_{j+1}$, $i_{j+1} - i_j \leq \alpha + 1$ and $\tau \in \{-\sigma + 1 \ldots \sigma - 1\}$. I.e. the condition is the same as before, but we now allow that the symbols can be "transposed" by some constant value. Now we also assume that the (integer) alphabet Σ is not arbitrary, but its symbols form a continuous range $0 \ldots \sigma - 1$. In MIR context, transposition invariance means that the pattern and its occurrence in text can be in different keys, which makes the model much more practical for query-by-humming applications.

3.12.1 Transposition invariant Simple

It appears that our Simple algorithm can be modified to this setting relatively straightforwardly. We again maintain a list L_i of text positions where the pattern prefix $p_0 \ldots p_i$ matches the text substring, but this time we must also maintain the set of possible transpositions for each such text

position. First notice that for any symbols p and t the transposition $\tau = t - p$ makes the symbols match exactly. Taking the δ condition into account, the set of possible transpositions becomes $\{\tau - \delta \ldots \tau + \delta\}$, i.e. for any single pair of symbols there are exactly $2\delta + 1$ allowed transpositions.

In the following we make the assumption that $2\delta + 1 \leq w$, where w is the number of bits in a machine word. In MIR applications this is practically never a restriction. We represent the set of possible transpositions as a pair $(\tau, \mathcal{T})$, where $\tau = t - p$ (the *base*) and $\mathcal{T}$ is the set of the $2\delta + 1$ possible *offsets* to the value τ. More precisely, $\mathcal{T}$ is a bitvector of $2\delta + 1$ bits. If the kth bit of $\mathcal{T}$ is set, then the transposition $\tau + k - \delta$ is valid.

Assume now that we have transpositions $(\tau_1, \mathcal{T}_1)$ and $(\tau_2, \mathcal{T}_2)$, and we want to compute the transposition $(\tau, \mathcal{T})$ that agrees with both, i.e.

$$(\tau, \mathcal{T}) = (\tau_1, \mathcal{T}_1) \ \cap \ (\tau_2, \mathcal{T}_2).$$

If $\tau_1 = \tau_2$ then

$$(\tau, \mathcal{T}) = (\tau_2, \mathcal{T}_1 \ \& \ \mathcal{T}_2),$$

where the bit-wise & operation effectively intersects the two sets. If $|\tau_1 - \tau_2| > 2\delta$, then the intersection is an empty set, and we just set $\mathcal{T}$ to zero. Otherwise, if $|\tau_1 - \tau_2| \leq 2\delta$ the intersection can be non-empty. To compute the intersection we must first bring $\mathcal{T}_1$ and $\mathcal{T}_2$ into the same base. This is easily achieved by shifting the bitvectors. Assume that $\tau_1 < \tau_2$. Then

$$(\tau, \mathcal{T}) = (\tau_2, (\mathcal{T}_1 >> (\tau_2 - \tau_1)) \ \& \ \mathcal{T}_2).$$

Symmetrically, if $\tau_1 > \tau_2$ we obtain

$$(\tau, \mathcal{T}) = (\tau_2, (\mathcal{T}_1 << (\tau_1 - \tau_2)) \ \& \ \mathcal{T}_2).$$

Let us now consider extending a (possible) prefix match. Let the current pattern position be i, and text position j. The set of candidate transpositions for this location is $(t_j - p_i, 1^{2\delta+1})$ (we use exponentiation to denote bit-repetition). This location is a prefix match, if in the previous row there are matching prefixes within α-window, and their corresponding transpositions agree with the pair $(t_j - p_i, 1^{2\delta+1})$. Let these transpositions be $(\tau_1, \mathcal{T}_1), \ldots, (\tau_k, \mathcal{T}_k)$, $k \leq \alpha + 1$. Then the set of transpositions extending the prefix match to position (i, j) is

$$((t_j - p_i, 1^{2\delta+1}) \cap (\tau_1, \mathcal{T}_1)) \cup \ldots \cup ((t_j - p_i, 1^{2\delta+1}) \cap \ldots (\tau_k, \mathcal{T}_k)),$$

Alg. 27 TI-DA-sdp-simple$(T, n, P, m, \delta, \alpha)$.

1 $tpm \leftarrow \sim 0 >> (w - (2\delta + 1))$
2 **for** $j \leftarrow 0$ **to** $n - 1$ **do**
3 $\quad \tau[j] \leftarrow t_j - p_0$; $\mathcal{T}[j] \leftarrow tpm$; $\mathcal{T}'[j] \leftarrow 0$; $L[j] \leftarrow j$
4 $h \leftarrow n$
5 **for** $i \leftarrow 1$ **to** $m - 1$ **do**
6 $\quad pn \leftarrow h$; $h \leftarrow 0$; $L[pn] = n - 1$
7 $\quad$ **for** $j \leftarrow 0$ **to** $pn - 1$ **do**
8 $\quad\quad$ **for** $j' \leftarrow L[j] + 1$ **to** $\min(n - 1, L[j] + \alpha + 1)$ **do**
9 $\quad\quad\quad ctpo \leftarrow t_{j'} - p_i$; $ptp \leftarrow \mathcal{T}[L[j]]$
10 $\quad\quad\quad$ **if** $|ctpo - \tau[L[j]]| \leq 2\delta$ **then**
11 $\quad\quad\quad\quad$ **if** $\tau[L[j]] < ctpo$ **then** $ptp \leftarrow ptp >> (ctpo - \tau[L[j]])$ **else**
12 $\quad\quad\quad\quad$ **if** $\tau[L[j]] > ctpo$ **then** $ptp \leftarrow ptp << (\tau[L[j]] - ctpo)$
13 $\quad\quad\quad\quad \mathcal{T}'[j'] \leftarrow \mathcal{T}'[j'] \mid (ptp \ \& \ tpm)$
14 $\quad\quad\quad\quad \tau'[j'] \leftarrow ctpo$
15 $\quad$ **for** $j \leftarrow 0$ **to** $pn - 1$ **do**
16 $\quad\quad \mathcal{T}[L[j]] \leftarrow 0$
17 $\quad\quad$ **for** $j' \leftarrow L[j] + 1$ **to** $\min(L[j + 1], L[j] + \alpha + 1)$ **do**
18 $\quad\quad\quad \mathcal{T}[L[j']] \leftarrow 0$
19 $\quad\quad\quad$ **if** $\mathcal{T}'[j'] \neq 0$ **then**
20 $\quad\quad\quad\quad L'[h] \leftarrow j'$; $h \leftarrow h + 1$
21 $\quad\quad\quad\quad$ **if** $i = m - 1$ **then** report match
22 $\quad$ swap(L, L'); swap$(\mathcal{T}, \mathcal{T}')$; swap$(\tau, \tau')$

where the union $\cup$ is simply computed as bit-wise `or` of the bitvectors $\mathcal{T}$, as they are all brought to the same base by the intersection operation. Hence, assuming that $2\delta + 1 \leq w$, this computation takes $O(\alpha)$ time. If the resulting set is non-empty, we put the position j into the list L_i, just as in the Simple algorithm without transposition invariance. Alg. 27 gives the complete pseudocode.

The worst-case time of this algorithm is $O(nm\alpha\lceil\delta/w\rceil)$. As in plain Simple, computing the list L_i takes $O(\alpha|L_{i-1}|)$ time (assuming that $\lceil\delta/w\rceil = O(1)$). However, this time the lists are longer on average. Clearly $|L_0| = n$, since pattern prefix of length 1 matches every text position. Hence computing L_1 costs $O(\alpha n)$ time, and the resulting list is of length $|L_1| = O(n\alpha\delta/\sigma)$, since the probability that two intervals intersect is upper-bounded by $(4\delta + 1)/\sigma$. In general, assuming that $\alpha\delta/\sigma < 1$, the ith list is of length

$$|L_i| \ = \ O(n(\alpha\delta/\sigma)^i).$$

This is exponentially decreasing with the above assumption. Thus the average time becomes $O(\alpha n)$.

3.12.2 Transposition invariant DP

We now present a basic dynamic programming solution that has better worst-case complexity than the Simple algorithm. The algorithm (conceptually) maintains a matrix $D_{0\ldots m-1,0\ldots n-1}$ (but only $\alpha+2$ columns are active at any time), where each $D_{i,j}$ is a binary vector of size $2\sigma+1$. If the kth item of this vector is set, that is, iff $D_{i,j,k}=1$, then $p_0\ldots p_i$ matches $t_h\ldots t_j$, for some h, with transposition $k-\sigma$. Let us define a helper matrix $\mathcal{T}$ as

$$\mathcal{T}_{i,j,k} \;=\; 1 \mid k \in [t_j - p_i + \sigma - \delta \ldots t_j - p_i + \sigma + \delta].$$

Now $D_{0,j}$ is easy to compute: $D_{0,j}=\mathcal{T}_{0,j}$. In general, $D_{i,j,k}$ depends on the values of the $\alpha+1$ sized window of the previous row:

$$D_{i,j,k} = 1 \mid \mathcal{T}_{i,j,k} = 1 \text{ and } \exists j' : 0 < j - j' \leq \alpha + 1 \text{ and } D_{i-1,j',k} = 1.$$

The (almost) naïve implementation of the above recurrence would result in algorithm with $O(nm\alpha\delta)$ running time. We first remove the $O(\alpha)$ factor of the trivial algorithm, then improve the average case, and finally reduce the $O(\delta)$ factor using bit-parallelism.

A trivial algorithm implementing our recurrence for $D_{i,j,k}$ would need to scan $\alpha+1$ vectors from the previous row. This can be avoided by using *counters* maintaining the total number of "voted" transpositions for each $(\alpha+1)$-window:

$$C_{i,j,k} = \sum_{j'=j-\alpha}^{j} D_{i,j',k}.$$

Thus we can rewrite our main recurrence as

$$D_{i,j,k} = 1 \mid \mathcal{T}_{i,j,k} = 1 \text{ and } C_{i-1,j-1,k} > 0.$$

The counters can be easily updated in $O(1)$ time per value by incremental computation:

$$C_{i,j,k} = C_{i-1,j,k} - D_{i-\alpha-1,j,k} + D_{i,j-1,k}.$$

This gives us $O(nm\delta)$ worst-case time. Note that only $O(m\alpha\sigma)$ space is needed for D since only the past $O(\alpha)$ columns are needed at any time. This could be reduced to $O(m\alpha\delta)$ by using the technique we used in Sect. 3.12.1. Similarly C takes only $O(m\sigma)$ space, since only one column of counters is needed to be active at any time. Finally, $\mathcal{T}$ is not needed explicitly at all, we used it only as a tool for the presentation. Alg. 28 gives the pseudocode, omitting initialization of the arrays, which are assumed to be all zero before the main loop. It also implements a cut-off trick discussed next.

Alg. 28 TI-DA-dp($T, n, P, m, \delta, \alpha$).

```
1   for k ← t_0 − p_0 + σ − δ to k ← t_0 − p_0 + σ + δ do D[0][0][k] ← 1
2   top ← m − 1
3   for i ← 1 to n − 1 do
4       Dco[0] ← 1
5       for j ← 1 to top do Dco[j] ← 0
6       for k ← t_i − p_0 + σ − δ to k ← t_i − p_0 + σ + δ do D[0][i % (α + 3)][k] ← 1
7       for j ← 1 to top + 1 do
8           for k ← t_{i−α−2} − p_{j−1} + σ − δ to t_{i−α−2} − p_{j−1} + σ + δ do
9               c ← D[j − 1][(i − α − 2) % (α + 3)][k]
10              D[j − 1][(i − α − 2) % (α + 3)][k] ← 0
11              C[j − 1][k] ← C[j − 1][k] − c
12              Cco[j − 1] ← Cco[j − 1] − c
13          for k ← t_{i−1} − p_{j−1} + σ − δ to t_{i−1} − p_{j−1} + σ + δ do
14              c ← D[j − 1][(i − 1) % (α + 3)][k]
15              C[j − 1][k] ← C[j − 1][k] + c
16              Cco[j − 1] ← Cco[j − 1] + c
17          if j ≤ top then
18              for k ← t_i − p_j + σ − δ to t_i − p_j + σ − δ do
19                  c ← min(1, C[j − 1][k])
20                  D[j][(i − 1) % (α + 3)][k] ← c
21                  Dco[j] ← Dco[j] + c
22              if j = m − 1 and Dco[j] > 0 then report match
23      while top ≥ 1 and Cco[top] = 0 and Dco[top] = 0 do top ← top − 1
24      if top < m − 1 then top ← top + 1
```

Cut-off

We make the following observation: if $D_{i\ldots m-1,j-\alpha\ldots j,k} = 0$, for some i, j and all k, then $D_{i+1\ldots m-1,j+1,k} = 0$. This is because there is no way the recurrence can introduce any other value for those matrix cells. In other words, if $p_0 \ldots p_i$ does not (δ, α)-match $t_h \ldots t_{j-j'}$ for any $j' = 0 \ldots \alpha$, then the match at the position $j + 1$ cannot be extended to $p_0 \ldots p_{i+1}$. This can be utilized by keeping track of the highest row number *top* of the current column j such that $D_{top+1\ldots m-1,j-\alpha\ldots j} = 0$, and computing the next column only up to row *top*+1. For this sake we maintain an array *Cco* so that $Cco_{i,j}$ gives the total number of "voted" transpositions for the last $(\alpha+1)$-window:

$$Cco_{i,j} = \sum_k C_{i,j,k}.$$

This is again trivial to incrementally maintain in $O(1)$ time per computed D value. Hence after the row *top* for the column j is processed, the new value of top is computed as

$$top = \text{argmin}_i\{Cco_{i\ldots top,j} = 0\}.$$

Now consider the average time of this algorithm. Computing a single cell $D_{i,j}$ costs $O(\delta)$ time. Maintaining *top* costs only $O(n)$ time in total, since it can be incremented only by one per text symbol, and the number of decrements cannot be larger than the number of increments. The average time of this algorithm also depends on the average value of *top*, i.e. the total time is $O(n \text{ avg}(top)\ \delta)$. For p_0 the probability of a match for any text position is obviously 1 (and *top* is at least 1). For rows $i > 0$ the probability that $\mathcal{T}_{i,j}$ intersects with $D_{i-1,j-1\ldots j-\alpha-1}$ is upper bounded by

$$\rho = 1 - (1 - ((4\delta + 1)/\sigma))^{\alpha+1}.$$

Hence the expected length of a prefix match is at most

$$\sum_{i=0}^{\infty} \rho^i = \frac{1}{(1 - \frac{4\delta+1}{\sigma})^{\alpha+1}},$$

i.e. $\text{avg}(top) = O\left(\frac{1}{(1-\delta/\sigma)^{\alpha+1}}\right)$, and the average time gets $O\left(n\frac{\delta}{(1-\delta/\sigma)^{\alpha+1}}\right)$. It is easy to show that this is $O\left(n\frac{\delta}{1-\delta(\alpha+1)/\sigma}\right)$ if $\alpha + 1 < \sigma/\delta$.

Bit-parallel algorithm

We note that the $O(\delta)$ factor can be easily reduced to $O(\lceil\delta\log(\alpha)/w\rceil)$, which is practically $O(1)$ in MIR applications. To see this, note that the counter values cannot exceed $\alpha+1$, so we can pack $O(w/\log\alpha)$ counters into a single computer word. All the inner loops (involving $2\delta+1$ iterations) can then be computed parallelly, updating $O(w/\log\alpha)$ counters in $O(1)$ time. The only non-trivial detail is the computation of minima of two sets of counters (parallelization of the line 19 of Alg. 28), but the solution exists [PS80], and is reasonably well-known. Note that for realistic assumptions (for MIR data) of $(4\delta+1)\alpha < c\sigma$, for some constant $c < 1$, and for $\delta\log\alpha = O(w)$, this variant achieves $O(n)$ time on average. However, in practice δ is often so small that the benefit of this parallelization is negligible, if any.

3.13 Experimental results for (δ, α)-matching and related problems

In this section we present the results of experiments intended to evaluate the performance of our algorithms for (δ, α)-matching, (δ, α)-matching with transposition invariance, and matching with character classes and general bounded gaps. The experiments were run on Pentium4 2 GHz with 512 MB

of RAM, running GNU/Linux 2.4.18 operating system. All the algorithms were implemented in C and compiled with `icc 7.0`.

We first experimented with (δ, α)-matching, which is an important application in music information retrieval. For the text we used a concatenation of 7543 music pieces, obtained by extracting the pitch values from MIDI files. The total length is 1 828 089 bytes. The pitch values are in the range $[0 \ldots 127]$. This data is far from random; the six most frequent pitch values occur 915 082 times, i.e. they cover about 50% of the whole text, and the total number of different pitch values is just 55. A set of 100 patterns was randomly extracted from the text. Each pattern was then searched for separately, and we report the average user times. Fig. 3.5 shows the timings for different pattern lengths. The timings are for the following algorithms:

- DP: Plain Dynamic Programming algorithm [CIM+02],
- DP Cut-off: "Cut-off" version of DP (as in [CCF05b]),
- SDP RW: Basic Row-Wise Sparse Dynamic Programming,
- SDP RW fast: Binary search version of SDP,
- SDP RW fast PP: linear preprocessing time variant of SDP RW fast (Alg. 17),
- SDP CW: Column-Wise Sparse Dynamic Programming (Alg. 18),
- Simple: Simple algorithm (Alg. 19),
- BMH+Simple: BMH followed by Simple algorithm (Alg. 20),
- BP Cut-off: Bit-Parallel Dynamic Programming [FG06c],
- NFA $\boldsymbol{\alpha}$: Non-deterministic finite automaton, forward matching variant [NR03], using $\boldsymbol{O(\alpha)}$ bits per symbol,
- NFA $\mathbf{log}(\boldsymbol{\alpha})$: Non-deterministic finite automaton, forward matching variant (Alg. 26), using $\boldsymbol{O(\log(\alpha))}$ bits per symbol.

We also implemented the SDP RW variant with $O(\sqrt{\delta} n)$ worst case preprocessing time, but this was not competitive in practice, so we omit the plots.

SDP is clearly better than DP, but both show the dependence on m. The "cut-off" variants remove this dependence. The linear time preprocessing variant of the SDP "cut-off" is always slower than the plain version. This is due to the small effective alphabet size of the MIDI file. For large alphabets with flat distribution the linear time preprocessing variant quickly becomes faster as m (and hence the pattern alphabet) increases. The column-wise SDP algorithm and especially Simple algorithm are very efficient, beating everything else if δ and α are reasonably small. For very small δ and α and moderate m the BMH variant of Simple is even faster. For large (δ, α) the differences between the algorithms become smaller. The reason is that a large fraction of the text begins to match the pattern. However, this means that these large parameter values are less interesting for this application.

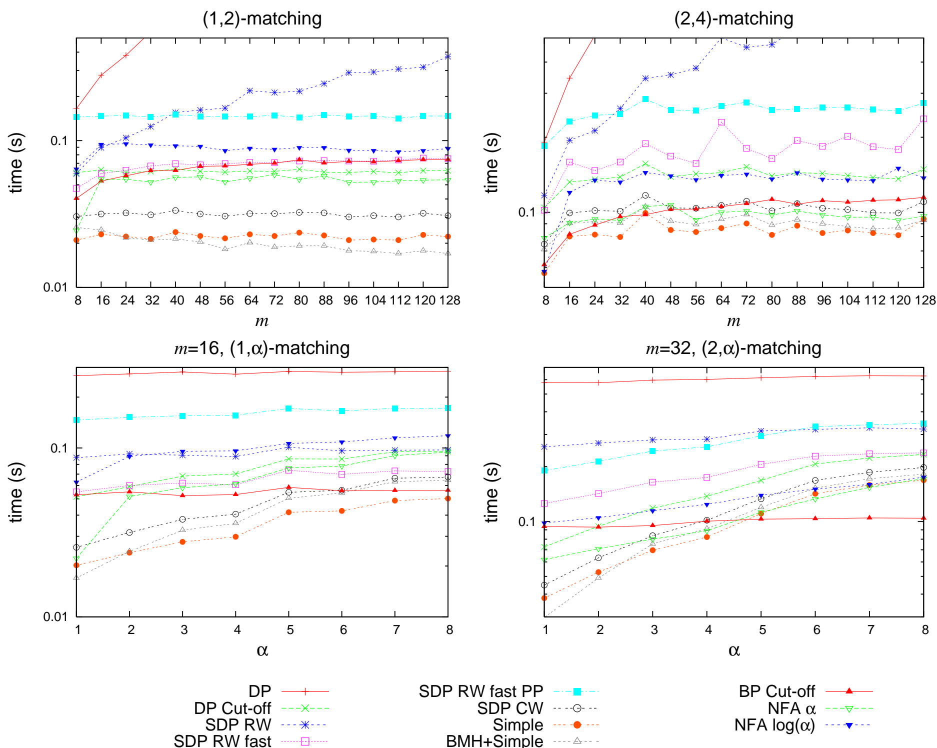

Figure 3.5: Running times for (δ, α)-matching, in seconds, for $m = 8 \ldots 128$ (top) and for $\alpha = 1 \ldots 8$ (bottom). Note the logarithmic scale

The bit-parallel algorithm [NR03] is competitive but suffers from requiring more bits than fit into a single machine word, yet Alg. 26 is even slower, besides having more efficient packing. This is attributed to the additional (constant time per text character) overhead due to the more complex packing.

3.13.1 Transposition invariance

Next, we experimented with the transposition invariant algorithms. The following algorithms were tested:

- BF-Simple: Plain Simple executed $O(\sigma)$ times,
- Simple: Transposition invariant Simple (Alg. 27),
- DP: (Transposition invariant) Dynamic Programming algorithm,
- DP Cut-off: "Cut-off" version of DP (Alg. 28).

The results are shown in Fig. 3.6. In this case Simple is again clear winner, despite of the theoretical superiority of DP Cut-off. For large α DP (Cut-

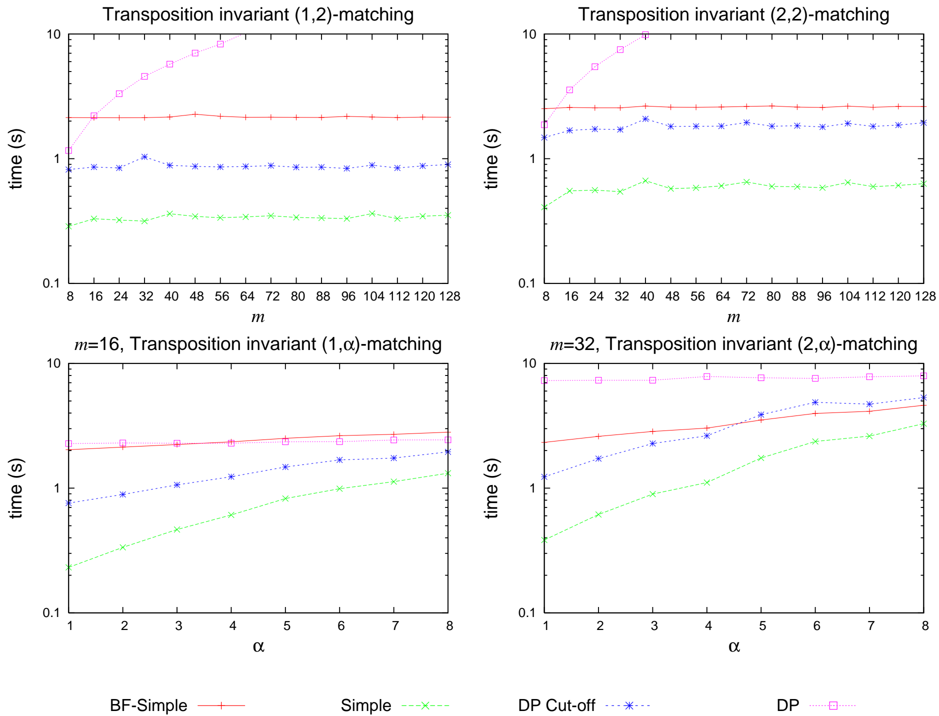

Figure 3.6: Running times for transposition invariant (δ, α)-matching, in seconds, for $m = 8 \ldots 128$ (top) and for $\alpha = 1 \ldots 8$ (bottom). Note the logarithmic scale

off) would eventually beat Simple, but in practical applications such large parameters are not interesting.

3.13.2 PROSITE patterns

We also ran preliminary experiments on searching PROSITE patterns from a 5 MB file of concatenated proteins. The PROSITE patterns include character classes and general bounded gaps. Searching 1323 patterns took about 0.038 seconds per pattern with Simple, and about 0.035 seconds with NFA. Searching only the short enough patterns that can fit into a single computer word (and hence using specialized implementation), the NFA times drops to about 0.025 seconds. However, we did not implement the backward search version, which is reported to be substantially faster in most cases [NR03]. Finally, note that the time for Simple would be unaffected even if the gaps were negative, since only the magnitude of the gap length affect the running time.

3.14 Experimental results for (δ, γ, α)-matching

We have also run experiments to evaluate the performance of our algorithms for the (δ, γ, α)-matching problem. This time, the test machine was a Pentium4 2.4 GHz with 512 MB of RAM, running GNU/Linux 2.4.20 operating system. All the algorithms were implemented in C, and compiled with `icc 9.0`.

As the text, we again used the 1.8 MB MIDI file (pitches only). Like previously, 100 patterns were randomly extracted from the text. Each pattern was then searched for separately, and we report the average user times.

We experimented with the following algorithms:

- BP Cut-off: Bit-parallel dynamic programming with cut-off, Alg. 25 (without the lazy preprocessing),
- BP Filter: The (δ, α)-matching version of BP Cut-off [FG06c] used as a filter, and Alg. 16 used for the verifications,
- DP Cut-off: Dynamic programming with cut-off, Alg. 16,
- Simple: Simple sparse dynamic programming, Alg. 21.

We omitted the results for basic dynamic programming based algorithms, since these are orders of magnitude slower. Fig. 3.7 shows the timings. Simple is the clear winner in most of the cases. BP Cut-off suffers from the large preprocessing cost, especially if the pattern alphabet is large. The same is true for the BP Filter, but this is more competitive in MIDI data, where the pattern alphabet is effectively very small.

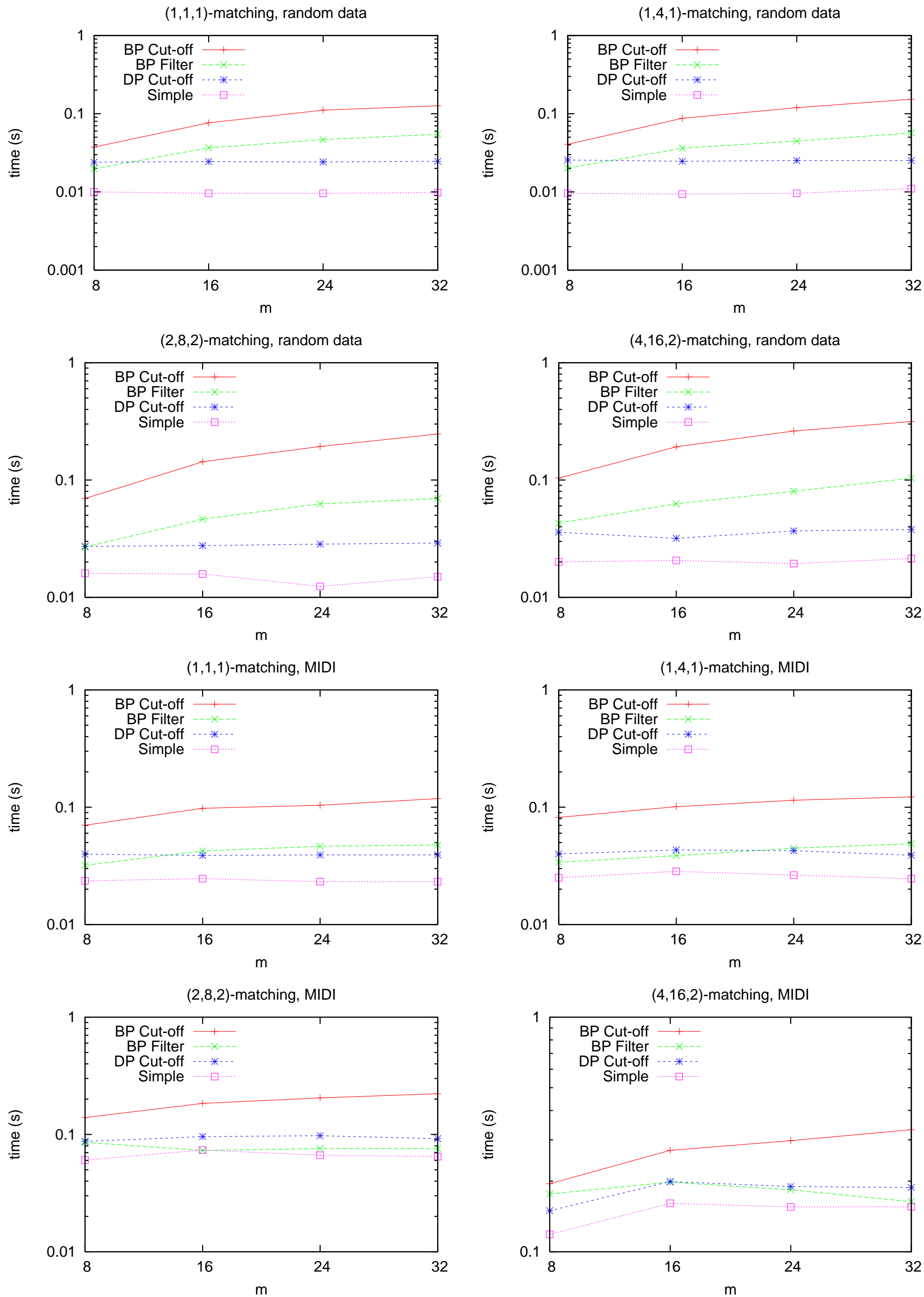

Figure 3.7: Running times for (δ, γ, α)-matching, in seconds, for $m = 8 \ldots 32$. Note the logarithmic scale

Chapter 4

Searching in compressed domain

The task of *compressed pattern matching* [AB92] is to report all the occurrences of a given pattern P in a text T available in compressed form. Certain compression algorithms allow for searching without prior decoding, which is practical if the search is fast enough.

Some compression algorithms can be adapted easier to the search scenario, while dealing with others is mainly a theoretical challenge. In this chapter we briefly present the main achievements in searching over compressed text, the inherent problems and ideas to overcome them, but more attention we pay for practical methods, discussing compression possibilities in inverted indexes, and numerous byte codes designed for high compression ratios, random access and search with pattern shift capabilities at the same time.

The price we usually pay for those spectacular achievements is some assumption on the underlying text. Our contribution [FG06d] into the field of searching in compressed domain is demonstrating how simply and still efficiently byte codes can be used in a scheme supporting searches for an arbitrary pattern in an arbitrary text. We also show how to improve (slightly) existing byte codes, which again can work with arbitrary text, if only the text is static [Gra08]. On the other side of the spectrum, for word-based schemes, we showed how to achieve competitive text compression ratios, for plain text [SGD05], or for XML [SGS08, SSG08], via a careful design of a preprocessor. Although in those cases compression was of primary concern, both schemes can be made searchable with relatively little effort, what was demonstrated practically in the XML-oriented case [SS07].

4.1 Motivation and brief overview of the area

Takeda et al. [TSM+01] distinguish between two goals of compressed pattern search. The first, easier, goal is to perform faster search over a compressed file in comparison to a plain decompression followed by an ordinary search (using some of the well-known techniques). This goal is important when storage saving has priority over search speed, which can be relevant e.g. for mobile devices with a relatively small Flash or disk memory. The second goal is to be able to run the search over a compressed file in less time than it would take to search over the uncompressed file. Achieving this goal is very attractive since it means that compression is then used not only to reduce the file size (as expected), but also to make searches faster. Fortunately, there are compression-and-search algorithms that achieve also the second goal, at least for some text characteristics and some query types. There are two reasons for which searching in the compressed domain can be faster than searching in an uncompressed text. The first is that compression reduces the I/O time. This effect can be crucial on very redundant texts (e.g. some XML databases) since disk access takes at least five orders of magnitude more time than main memory access time, on a typical computer architecture. Even if the whole text resides in main memory, a similar phenomenon can be noticed between various levels of the memory hierarchy (e.g., large but slow RAM, much smaller but much faster L2 CPU cache, and even smaller and even faster L1 CPU cache), also beneficial to compressed data, but this time the speed difference is much less drastic. The second reason for (potentially) faster search in compressed data is that, surprisingly perhaps, sometimes it requires less CPU work, e.g. fewer character comparisons.

There are basically two scenarios for compressed pattern matching. In one of them it is assumed that the text is segmented into words, i.e. short strings with separators (usually blanks) between them. The pattern is also assumed to be a phrase of one or more words, and the matches need to be at word boundaries only. For example, if the pattern is "shaggy dog", then a subsequence "shaggy dogs" from T will not be reported as a match. This is called word-based searching.

Another scenario is more flexible: it does not assume anything about the text or the pattern, except for both being arbitrary sequences of characters over a known (and usually finite and indexed) alphabet. We are going to take a closer look at the specifics of both approaches.

The word-based orientation is widely used in *inverted indexes* [Knu73, WMB99], classic data structures to index natural language texts, and nowadays belonging to fundamental mechanisms of any web search engine. An

inverted index is composed of two elements: the *vocabulary* (the set of distinct words) and the *occurrence lists* (also called *posting lists*). For large texts, the vocabulary takes only a small fraction of the whole index, which is no surprise in the light of the well-known empirical Heaps' law [Hea78], which claims that the vocabulary of a text of n words is of $O(n^\beta)$ size, where β is usually between 0.4 and 0.6 in practice. Navarro ([Nav98, Sect. 2.10, p. 24]) reports that the vocabulary for 1 GB of the TREC collection [Har95] occupied only 5 MB of memory. The occurrence lists, however, are much more demanding. They store, usually in the increasing order, the positions of a given word's occurrences in the text. A substantial part of any real NL text is used by *stopwords*, that is, words with mainly syntactical purpose (English examples are "a", "the", "of" etc.). Stopwords are also called function words [MNF58], and in the cited work they comprise articles, prepositions, pronouns, numbers, conjunctions and auxiliary verbs. Sometimes, the meaning of a stopword is understood broader, e.g. in Onix Text Retrieval Toolkit (http://www.lextek.com/manuals/onix/stopwords1.html) some popular (but non-auxiliary) verbs (e.g. "know") and adjectives (e.g. "young"), and their related grammar forms (e.g. "knows", "known", "younger") are also considered stopwords (in total, their list contains 429 English words). From a pragmatic point, a stopword can be any word that does not convey enough content to appear alone in a query. Depending on the definition and the text, stopwords use from about 40% to 50% of all words in a text file [ANZ97, NMN+00].

Still, even if the text is pruned from stopwords, the occurrence lists use about 35% of the original text size, as shown on several large text collections, using Igrep software package [ANZ97]. This is quite much, so a compromise idea is to use *block addressing* instead of exact references to word occurrences. This concept was used for the first time in Glimpse [MW94]. More precisely, the underlying idea of Glimpse is to partition a large text collection into equal blocks (of size, e.g., several hundred kilobytes) and for each word from the vocabulary keep only the list of blocks in which the given word occurs (no matter whether once or many times). The exact word positions are not stored in the index. If the search pattern is a phrase containing several words, the search means to retrieve the appropriate lists, intersect them, and finally perform a linear scan over the resulting blocks (if any left); how to perform the intersections efficiently is still a research problem [ST07]. The number of blocks can be limited to e.g. 256, to store block numbers in single bytes. The block sizes depend then on the overall text size. If the "text" to index is actually a file system or a web graph, it is more convenient to replace "blocks" with individual files, e.g. single HTML documents. Glimpse is much slower than Igrep but consumes typically only 2–4% of the text

[MW94], unfortunately its effectiveness deteriorates on huge files. Baeza-Yates and Navarro proved [BYN00] that a block addressing index may yield both sublinear space overhead and sublinear query time, on average. Still, of course, the original text cannot be discarded.

A number of techniques to compress inverted indexes (in particular, block addressing based ones) have been presented in the literature [MB95, NMN+00, TS08]. The main problem is how to represent the occurrence lists both succinctly and with support for fast access to the block numbers stored on them. The mechanisms to reach those contradictory goals are compact gap (i.e., difference) encoding, direct access to a list in regular intervals [WMB99], and complemented lists [NMN+00], i.e. storing a list of blocks not containing a given word, if this word occurs in more than a half of the blocks. Also, the word vocabulary can be compressed, making use of e.g. frequent digrams or common prefixes of words.

An important advantage of word-based inverted indexes is that they support approximate queries (and some other non-trivial queries) relatively easily [NMN+00]. The basic idea for resolving approximate queries is search for all the possible distortions, within the specified allowed error level, of a given phrase, which still result in a sequence of words from the vocabulary. Then, all those potential phrases are sought for in their respective blocks, but still some optimizations are possible to make this search faster.

Naturally, word-based compression techniques are also used to compress the text without an index [MNZBY00]. In most such schemes, a codebook over the vocabulary is built, with codewords usually taking an integer number of bytes (not bits), and the text words are replaced by their corresponding codewords [BFNP07]. Using a byte- rather than bit-oriented approach makes a search over the compressed text both simpler and faster. In the rest of this chapter we focus on searching a compressed text without an index.

To sum up, word-based compression is very practical, as long as it can be applied: its mechanism is simple, the search is fast, the compressed text together with its word dictionary takes only about 30% of the original representation [BFNP07], and more advanced queries can also be handled with relatively little difficulty. The problem is, however, that the assumption of "text" made up of "words" separated with spaces, so natural and convenient for Western languages (e.g., English, French), is inappropriate for oriental languages (e.g., Chinese, Korean), DNA and protein sequences, or structured music files (MIDI). Moreover, word-based approach is not perfect also for some European languages: agglutinative ones, like Finnish or Hungarian, or inflecting ones, like Polish or Russian. For the first group of languages, the vocabulary is potentially infinite, and for the second group, users are often

interested in finding any of several grammar forms of a given word (usually having a common prefix). It should be thus clear that there exist important applications for compression algorithms that allow searching directly in the compressed stream, without assuming practically anything about the data. The algorithm we present in Section 4.3 belongs to this category.

Let us report briefly what has been done is full-text search-supporting compression. One of the first algorithms of this kind was given by Manber in 1994 [Man94]. It replaces the most frequent pairs of consecutive characters (bytes) in the text with a byte that does not occur in that data (for e.g. plain ASCII English text it poses no technical problems as more than 128 byte values are never used and can thus be given special meaning). Thanks to a careful selection of the replacement set, which does not allow for overlapping pairs, the search mechanism is very simple but the compression ratio is weak, about 70% for English texts. A similar idea, called byte-pair encoding (BPE), was presented in the same year by Gage [Gag94], and it was devised with compression in mind only. BPE is Manber's idea taken to the extreme: the most frequent pairs are encoded one-by-one, and a given "new" symbol may be compound of symbols which are not taken from the original alphabet but are sequences already. Gage's implementation can be extremely slow in compressing the text, but Takeda et al. [TSM$^+$01] modified the algorithm in order to make compression fast, for the price of some compression loss. Still, the compression ratio for English is slightly less than 60% in their experiments, which is quite acceptable for full-text schemes. Takeda et al. show several ways to perform exact pattern search directly in BPE-compressed text, for example by modifying the transition table in the KMP algorithm. Interestingly, also a word-based version of BPE was presented [Wan03], with 25–30% compression ratio, but so far no search scheme for it has been proposed.

Another possibility is searching in run-length encoded texts [AB92], where also approximate matching algorithms exist [ALS99, MNU03, ALM02]. This line of research has limited applications since the run-length encoding (RLE), applied as a single compression algorithm, is hardly ever useful (e.g., doesn't help on NL texts, DNA, music databases etc.). A natural application of RLE are black-and-white images, e.g. faxed documents, which tend to contain long runs of white pixels (background). Also searching in two-dimensional run-length encoded text has been considered, with the optimal time complexity, $O(u)$, where u is the length of the compressed text, achieved in [ABF94].

The Huffman compression [Huf52], and variants of, have been considered many times (e.g., [KS05, TSM$^+$01, TMK$^+$02, FT03]) as a text representa-

tion for which direct searching is possible. In this case, searching is fairly straightforward, but requires the whole compressed text being scanned to keep track of the codeword boundaries. In other words, it seems hard to devise a search algorithm for Huffman stream which would enable skipping over parts of the stream in the manner of the Boyer–Moore family of algorithms.

Another line of research is searching over Lempel–Ziv [ZL77, ZL78] compressed data (e.g., [KTSA99, NT00]). Those algorithms provide better compression ratios than Huffman but the search is difficult and its speedup over the naïve "first decompress, then search" approach is at best moderate (about two-fold, i.e. it satisfies the first goal according to the cited work by Takeda et al. [TSM+01], but not the second [NT00]). Interestingly, also approximate matching in Lempel–Ziv compressed texts has been considered, again reaching the first goal in practice [KNU03]. On the theoretical front, Bille et al. recently showed that any matching algorithm for the Levenshtein distance can be plugged into their scheme for searching in the LZ78-compressed text, and also a simple time-space tradeoff is possible [BFG07].

There are also some more exotic compression schemes, like using antidictionaries [CMRS98] or SEQUITUR [NMW97] which infers grammar rules from the given text, for which search schemes have been devised [STSA99, MHM+01], but we don't find them practical for some reasons. A decade ago Rytter presented a survey [Ryt99] of theoretical one- and two-dimensional algorithms for compressed matching and related problems, including, among others, RLE, antidictionaries and LZ approaches.

4.2 Search-supporting codes for large alphabets

In this section we present several compression codes which proved their usefulness, both in theory and in practice, for word-based compression supporting fast search [BFNP07]. What is less obvious is that most of those codes can also be adapted to full-text search requirements [FG06d], which is the topic of the algorithm we present in the next section. Still, indisputably, those codes require large alphabets to deal with. Examples of large alphabets are: Unicode (a natural choice if e.g. text in one of many Asian languages is to be handled), the vocabulary prebuilt for a given natural language (typically, tens of thousands of words, i.e. symbols in the alphabet), the set of distinct words in a large NL text (again, tens of thousands of symbols, or more), the set of distinct words and markup tags in a large

and diverse XML collection, the set of distinct q-grams in a large text, and so on.

Throughout this section we however assume that those codes will be used on words. It is worth to realize at the start that natural language texts are composed of words and separators between them, so a basic question is how to efficiently handle both words and separators. It is possible to use disjoint alphabets and unrelentingly switch between them. A better practical choice is however the method called spaceless words [MNZBY00], which assumes that most words are followed by a single space. If this is the case, just the word is encoded. If not (for example, the separator is a comma followed by a space), then the separator is encoded just after the word. At the decoding time, the omitted spaces are inserted after reconstructed words, unless the next codeword corresponds to a separator.

Word-based compression is attractive since it emulates, in a way, higher-order literal compression, i.e., can attain significant compression ratios, while at the same time being simpler to implement and less memory consuming, as the compression model is not polluted with hardly useful contexts. But perhaps the most important feature of word-based compression is that even its simplest, static order-0 implementation wins in compression ratio with popular algorithms from Lempel–Ziv family. The property of being static implies a possibility of fast and simple search over a text compressed in that way, and also makes direct access to the text possible (e.g., for decoding and displaying only an excerpt of text from a middle).

The most natural choice for static order-0 compression is Huffman coding [Mof89]. It achieves less than 30% of compression ratio, but the search or decompression are not that very fast because of the need of bit-wise manipulations. A more practical idea was used by de Moura et al. [MNZ97]. They replaced the classic, binary Huffman with 256-ary Huffman. In other words, all Huffman codewords had either 1 bytes, or 2 bytes, etc. This way, decompression got much simpler but searching in the compressed data was still unable to perform any skips over text characters (bytes). The next modification, from the same work, was therefore a *tagged Huffman code*, in which 7 out of 8 bits in each 256-ary Huffman codeword byte carried the actual information about the symbol, and 1 bit was used as a flag to signal the first byte of a codeword. Thanks to it, the tagged Huffman code can be accessed in any position, even if the access point is in the middle of some codeword (examining at most a few following bytes is enough to detect a codeword boundary). Moreover, this scheme enables searching with BM-like algorithms, without any risk of finding false matches, as opposed to

plain 256-ary Huffman. The price for all those desirable properties is some deterioration in the compression ratio, to about 35%.

Interestingly, in the family of byte codes with one bit per byte spent for a flag, the tagged Huffman is not optimal. To notice it, it is enough to make a trivial change to the scheme: use a flag to denote the last, not the first, byte of each codeword. In this way, the flag bits in the stream are enough to make the code a prefix one. Consequently, all the combinations on the "message" bits are valid and should be used, which stands in contrast to the tagged Huffman. This idea was presented and analyzed in 2003 [BINP03], under the name of *end-tagged dense code* (ETDC). Indeed, this code is "denser" than tagged Huffman, but also amazingly simple and easy to implement. The compression ratio improves to 32–33%. Note that what is needed to generate the code is only the list of words ordered by frequency; the frequencies themselves are not needed. As an interesting historical note, we cite Culpepper and Moffat (2005) [CM05], on ETDC: *The exact origins of the basic method are unclear, but it has been in use in applications for more than a decade, including both research and commercial text retrieval systems to represent the document identifiers in inverted indexes.*

Traditionally, the flag is the highest bit in a byte (although this is purely conventional) in the described schemes. The solution in ETDC thus means that bytes in the range from 128 to 255 end a codeword, while bytes lower than 128 do not end the codeword. The former values can be called *stoppers* and the latter *continuers*. The ranges for stoppers and continuers must be disjoint, of course. What is important, however, is that the value 128 does not have to be the threshold. In general, we can say about having s stoppers and c continuers, with the only condition that $s + c = 256$. This generalization was proposed in [BFNE03] under the name of a (s, c)-dense code, or shortly (s, c)-DC. Obviously, ETDC is $(128, 128)$-DC. The authors also use the name (s, c) stop-cont code for a code with each codeword being a sequence of zero or more values from 0 to $c - 1$ terminated with exactly one value from c to $s + c - 1$. In fact, it can be of some value to generalize the (s, c)-DC definition given above with replacing the condition $s+c = 256$ with $s+c = 2^b$, where b is any positive integer (perhaps 4 might be useful for small alphabets, or 16 for huge alphabets), but in the following we assume byte-oriented codes, as most practical, which corresponds to setting $b = 8$.

The (s, c)-DC idea was discovered a year earlier (in 2002) by Rautio et al. [RTT02], but their search techniques were different than in [BFNE03] and were based on splitting each codeword into two parts sent into separate streams. For typical distributions, s should be greater than c. For example in the experiments in [BFNP07], the optimal s for several large collections

of English text varies from 188 to 198, leading to at least 0.5% compression improvement with respect to ETDC. In fact, (s, c)-DC loses very little to plain (non-tagged) 256-ary Huffman, about 0.2% of the size of the original text. Still, the loss to order-0 entropy in the word-based model, which can be achieved by arithmetic coding, is greater, about 4–5%. Unfortunately, it seems impossible to efficiently search in an arithmetically encoded stream. Searching in (s, c)-DC is as easy as in its predecessors, tagged Huffman and ETDC, since (s, c)-DC is also a prefix byte-oriented code, supporting fast Boyer–Moore like searching strategies, increasing their speed with growing pattern length (measured in bytes of the compressed pattern representation).

An interesting question concerns how to find the optimal s value (while $c = 256 - s$ is then obtained automatically). The code allows for s 1-byte codewords, sc 2-byte codewords, sc^2 3-byte codewords, and so on. Setting a small s implies that the amount of the alphabet symbols (i.e. words, typically) encoded with only 1 byte, will be strongly limited. On the other hand, in such a case the number of symbols represented with 2 bytes (and not 3 or more bytes) will be much larger than for a case of s being significantly larger. Clearly then, the best choice depends on the symbol probability distribution. The problem can be expressed as finding the argument s minimizing the resulting compression ratio (or compressed text size). Basically, two simple strategies can be used to find the optimal s. One assumes that the compressed text size as a function of s is a convex function and then a single local minimum exists, which can be found with binary searching. The other strategy is brute-force checking all possible 255 values of s. Interestingly, for real distributions the function is practically always convex ([BFNP07, Sect 4.3] contains intuitive arguments why it happen so, assuming that Heaps' and Zipf's laws hold), but it is possible to construct an artificial distribution with more than one minimum. An example of such a function, with two local minima, is given in [BFNP07]. From a practical point, the binary search strategy seems safe, especially that even if a function has more then one minimum, they probably yield almost identical compression.

The search in (s, c)-DC (including ETDC) is slightly different than in tagged Huffman. The latter algorithm uses tag bits to signal the codeword beginnings, i.e., it is impossible to have the pattern misaligned. In this way, false matches are impossible with this algorithm. Now we are going to present on an example what can happen with (s, c)-DC. Let $s = 156$, i.e. the values $[0 \ldots 99]$ correspond to continuers, and the values $[100 \ldots 255]$ represent stoppers. Imagine that the pattern is a single word and is encoded as a pair of bytes $(50, 200)$. Now, let us have a piece of encoded text:

$\ldots$ 35 50 200 $\ldots$ 187 50 200 $\ldots$

Seemingly, there are two matches in the shown piece of text, and both will be found by e.g. BMH algorithm, but the first of them should be rejected as being a false one. Indeed, this can be found with peeking just the previous byte. In the first case, the examined value is 35, i.e. belongs to continuers, which means that the found "match" was actually a suffix of some 3-byte or longer codeword. In the second case, the previous byte is 187, i.e. a stopper, which means that found pattern sequence starts at a codeword boundary, i.e. must be a genuine match. This nuisance is very little in fact, since the extra checks are needed only at potential match positions, which are rare.

Variable-length byte coding on words is also often used for sole text compression. One of the most successful examples of this approach is the word replacing transform (WRT) by Skibiński et al. [SGD05]. The scheme assumes a static dictionary shared by the compressor and the decompressor (or, in other words, by the sender and the receiver), and words not found in the dictionary are written verbatim. The byte-orientedness of the encoding is needed not for searching (which is not supported in WRT, although this option seems possible) but rather for more efficient further compression with general-purpose algorithms. It appears that stronger compressors, e.g., from the PPM family [CW84, Shk02], work better with less dense WRT coding, and weaker compressors, e.g. Deflate [Deu96] from the LZ77 family, prefer denser coding. To address this phenomenon, the authors of the cited work devised and implemented two variants of WRT, reaching up to 14% improvement in the latter case, with gzip, when compared to the previous leader, StarNT [SZM03].

Byte encoding on the word level is also one of the main principles in the XML compressor XWRT (XML word replacing transform) [SGS06, SGS07, SGS08, SSG08]. Together with separating different elements and content types into multiple streams, and dedicated encoding of numbers and dates, this idea helps XWRT achieve a little advantage over the previous state-of-the-art compressor, SCMPPM [AdlFN05, ANdlF07], while being also by 30% faster in the compression, and as much as nine times faster in the decompression [SGS08]. Tuning the transform for a stronger back-end compressor (from PPM family) lets the advantage in compression over the competition grow even more, but for the price of making the algorithm symmetric in speed [SSG08].

It seems very hard to improve the compression ratio of (s, c)-DC while keeping its advantages like byte-orientedness and fast search support. Fortunately, it is possible to weaken the input assumptions without losing most of the advantages of the code but improving the compression significantly. The idea, used in the scheme called *pair-based end-tagged dense code*

(PETDC) [BFNP06], was to treat frequent pairs of adjacent words as individual symbols. PETDC can be classified as a variable-to-variable length code, since the input symbols vary in their length (i.e., the number of component words – one or two) and also vary in the output codeword length. Searching for a pattern in this scheme may bring some problems at the encoded sequence boundaries (since a word can occur alone or be a prefix or a suffix of possibly many word pairs), but this can be solved by referring to a multiple search algorithm. The compression ratio, on standard large English text collections, gets around 28%, in contrast to 32–33% achieved by the original ETDC. The 28% ratio is (accidentally) very similar to order-0 entropy in the word model. The reported compression speed [BFNP06] is however about 2.5 times worse than for ETDC, but in the decompression the loss is only about 20%. We are not aware of a similar modification for the (s, c)-DC code, but it is likely that the compression improvement would be very slight.

Another possibility to generate dense codes was pointed out by Culpepper and Moffat [CM05]. In their solution, the first byte in a codeword keeps information about the length of the whole codeword. Unfortunately, this idea poses trouble with efficient search [CM06].

All the described so far byte codes are static, i.e. the correspondence between a symbol and its codeword is unchanged during the compression process. This is very convenient and makes search over the compressed text both simple and fast. On the other hand, adaptive compression methods have other advantages: the text can be compressed in one pass (as opposed to static compressors which need a preliminary pass over the whole text to gather the symbol statistics, and the actual encoding is perform in the second pass), and may achieve better compression if the text characteristic varies over time. One-pass codes are especially welcome in real-time transmissions (it is thus convenient to call the coder's side the *sender*, and the decoder's side the *receiver*). In [BFNP04] two dynamic word-based algorithms have been proposed. One is 256-ary version of the well-known FGK algorithm [Knu85], which is in turn a variant of the dynamic Huffman coding [Gal78]. The other algorithm is a dynamic version of ETDC. It is much simpler and also somewhat faster than dynamic word-based Huffman. An interesting modification, still from the same team, was published in [BFNP05]. This time they made the dynamic ETDC slightly slower in compression and slightly worse in the compression ratio but at the same time achieved about 2.5-fold speedup in the decompression. Such asymmetry of the algorithm, dubbed *dynamic lightweight end-tagged dense code* (DLETDC), is unique among adaptive algorithms. The asymmetry is possible since the decoder

has less work to do, it does not update the frequency list of words, like the coder does, but is simply explicitly informed by the coder when some codeword changes are necessary. Indeed, it does not affect the compression ratio if updates (symbol exchanges) in the vocabulary are performed only when they force a symbol get a shorter codeword than it has so far. For example, if a word W had so far 20 occurrences and its current codeword has two bytes, the 21st occurrence of W should not change its codeword if W still belongs to the words that should be encoded on two bytes (although, naturally, the increased frequency of W is likely to move it higher on the sorted frequency list). It appears that the extra information passed to the receiver is little enough to deteriorate the compression ratio by about 0.1% only.

Recently, Fredriksson and Nikitin [FN07] proposed a code which can be used in the word model, but can also work on characters (although this option seems somewhat less practical). Their code provides constant-time random access to any symbol of the original text T, and allows efficient (average case optimal) search over it. The algorithm produces two bit streams. One stream is a concatenation of codewords from the "densest possible" code: the most frequent symbol s_0 gets codeword 0, the second most frequent symbol s_1 gets 1, then symbols $s_2 \ldots s_5$ get codewords 00, 01, 10, 11, respectively, and so on. Of course, such a stream of data is non-decodable, therefore an accompanying stream must hold the information about the boundaries. In the basic variant, the codeword lengths are kept in the accompanying stream in unary ($k - 1$ zeros followed by a single bit one, to represent the length k), which is extremely simple to implement but the overall size of encoded T is (pessimistically) bounded by $|T'| = 2n(H_0(T) + 1)$ bits, where $H_0(T)$ is the zeroth-order empirical entropy of T (definition of the empirical entropy is given in Sect. 5.4.1). Constant-time random access to an arbitrary ith symbol of T is achieved thanks to the *select* structure [Mun96], which returns the position of ith set bit in a binary sequence, and is known to be sublinear in the length of the sequence it works for, i.e., the length of T' in our case. This basic scheme is enhanced in a couple of ways, for example, a generalization is given which can lead to a code similar to ETDC, as if its tag bits were order-0 entropy-compressed, still with constant-time access. A drawback of this scheme is that appending a piece of text at the end, even without caring for the change in symbol statistics, requires modifying several streams (most costly of which is updating the *select* structure).

4.3 q-gram based full-text coding with efficient search capabilities

In the previous sections we outlined the main algorithmic approaches to combine compression with search. The overall conclusion is that most of the known schemes either assume a text formed into words, or provide mediocre compression (BPE variants), or, finally, are complex and rather theoretical. We also pointed out that there exist important applications for compression algorithms that allow searching directly in the compressed stream, without assuming that the input text is segmented into words. The algorithm [FG06d] we present in this section belongs to this category.

Albeit devised for non-structured texts, the algorithm takes some inspiration from word based solutions. We show that combining ETDC or (s, c)-DC with q-gram based dictionary gives a simple compression algorithm that does not assume that the text is formed of words and separators. The compression rate is only moderate on word based texts, but much more competitive on e.g. DNA and protein data. We developed novel algorithms that perform efficient pattern matching in the compressed q-gram stream. The performance is comparable to that of direct searching on uncompressed texts (not counting the decompression times).

Note that there is no clean way to use the word based methods for arbitrary data. There seems to be at least three (problematic) ways to do this: (1) simply define some arbitrary text characters as "separators" (but it is not clear which characters should be chosen, and the optimal choice depends on the data); (2) use every qth character as a separator (this effectively leads to our approach); (3) assume that the text consists of an extremely long single word, which means that the compressor degenerates into the algorithm used to compress the vocabulary, e.g. Lempel–Ziv compression (but this means that the search algorithms do not work anymore, or at least the search degenerates into searching in LZ compressed text).

The search problem variants that the word and q-gram based compression naturally support are different. The word based approach allows flexible searching of *phrases*, consisting of words, while searching e.g. partial words requires resorting to multiple matching [MNZBY00]. Our search algorithms solve the classical string matching problem, more suitable to data that are not natural language.

Our proposal is based on q-grams and its immediate consequence is that the scheme works straightforwardly with patterns not shorter than $2q - 1$ characters. Nevertheless, we present also an algorithm for handling shorter patterns. Both cases will be discussed in the two successive subsections.

Albeit the algorithm supports full-text searching, it is based on tools devised for the word model. We could use, e.g., tagged Huffman but we experiment with more efficient techniques, ETDC and (s, c)-DC. The dictionary of symbols in those schemes contains all the space-delimited words in the given text $T = t_0 t_1 \ldots t_{n-1}$. In our case, T is a sequence of n/q non-overlapping q-grams (w.l.o.g. we can assume q divides n). The ith q-gram corresponds to $T[(i-1)q \ldots iq-1]$. All the unique q-grams in T must be represented in a dictionary *Dict*, hence it is crucial to find a balance between the more efficient compression of the sole text T and a rapidly growing dictionary *Dict* with increasing q. For natural texts, q should be 4 or 5.

4.3.1 Searching for long patterns

At the start we assume that $P = p_0 p_1 \ldots p_{m-1}$ has at least $2q-1$ characters. Searching P in the encoded T consists of two stages. In the first stage, we generate q possible alignments of P. This means that, e.g., the last, mth, character of P may be either the 1st symbol, or the 2nd, etc., or the qth symbol of some q-gram. Ignoring at least one of those alignments could result in missed matches. Each of those alignments corresponds to (at most) one encoded byte sequence, e.g., according to the (s, c)-DC algorithm and its codebook for text T. Note that at most $q-1$ characters at both P's beginning and its ending may be truncated. For example, if $q = 4$ and $m = 10$, one of the alignment variants, which could be denoted as $3+4+3$, covers only a single q-gram. It is important to note also that some alignments may contain one or more q-grams that do not appear anywhere in T (at least, according to its partition into non-overlapping q-grams) and hence cannot have a valid representation in the codebook. This is a fortunate phenomenon, since we immediately discard such a pattern alignment from further search, as this cannot be matched to. Alg. 29 shows the pseudocode.

The second stage performs multiple matching of the valid pattern alignments in their compressed form over the compressed T. We can use virtually any "off-the-shelf" algorithm devised for multiple searching, e.g., Wu–Manber [WM94] or Aho–Corasick [AC75]; more references can be found in [NR02]. Certain algorithms from this category, e.g., Wu–Manber, make use of Boyer–Moore skips, which are desirable also in our case, thanks to the properties of the ETDC and (s, c)-DC codes. Matching pattern variants has to be verified by comparing their discarded characters (at the beginning or the end of P) against the respective neighborhood of the compressed T.

We used BNDM algorithm [NR98, NR02], which is of $O(n \log_\sigma m/m)$

Alg. 29 DCPreProcLong($P, q, Dict$).

Input: pattern P, q, dictionary $Dict$
Output: Set of patterns P', number of patterns r, their minimum length $minl$

```
1   r ← 0; minl ← ∞
2   for i ← 0 to q − 1 do
3       l ← 0; k ← NIL; j ← i
4       while j < m − q + 1
5           k ← arg_k{Dict[k] = P[j ... j + q − 1]}
6           if k = NIL then break
7           Q ← codeword(Dict[k])
8           for h ← 0 to |Q| − 1
9               P′[r][l + h] ← Q[h]
10          l ← l + |Q|; j ← j + q
11      if k ≠ NIL then
12          r ← r + 1
13          if l < minl then minl = l
14  return P′, r, minl
```

average time complexity, where n and m are in our case the compressed text and pattern lengths, and σ is the alphabet size. This algorithm can easily be adapted to multi-pattern filtering by using the well-known pattern superimposition technique and classes of characters [BYN97]. This works very well in our scheme, since the number of patterns is small (at most q), and the size of the alphabet relatively large (256), that is, the effective alphabet size is $256/q$, reasonable in practice. As the algorithm is bit-parallel, the maximum pattern length (in *compressed* form) is limited by the number of bits in a computer word. The case of longer patterns may be handled by using several computer words.

Finally, we would like to illustrate the algorithm with an example. Let the pattern be `nasty_bananas`, $q = 3$ and ETDC chosen for encoding the q-grams. First we split the pattern into 3-grams considering three alignments:

```
nas ty_ ban ana
ast y_b ana nas
sty _ba nan
```

(Note that we have removed up to $q-1$ characters at each pattern boundary.)

Now, we encode the 3-grams, so the pattern alignments may turn into something like:

(110)(92)(193) (184) (24)(202) (103)(220),
(45)(236) (84)(155) (103)(220) (110)(92)(193),
(81)(211) (23)(140) (36)(17)(199),

where parentheses and spaces are added only for clarity. The shortest of those encodings has 7 bytes (the third one), therefore we truncate the other

two sequences to 7 bytes. Finally, all the 7-byte sequences are input for the BNDM algorithm adapted for multiple matching. Obviously, this approach requires verification of potential matches.

4.3.2 Searching for short patterns

For patterns shorter than $2q-1$ characters the problem is that there are pattern alignments that do not totally cover any q-gram in the text. Even worse, the pattern can be shorter than q characters, and hence no whole q-gram can occur in any pattern alignment.

There are several ways to overcome this problem. The simplest but the most inelegant solution would be to decompress the text and then search. A more sophisticated alternative resorts to bit-parallelism. We use the method proposed in [Fre03] to build a Shift-Or automaton [BYG92, WM92b] to process a whole q-gram in time $O(\lceil (m+q-1)/w \rceil)$, where w is the number of bits in computer word. This is $O(1)$ in practice, given our limit for m, and the practical values of q. The idea is to have an implicit decoding of the text, encoded to the automaton, i.e. the automaton makes implicit transitions using the original text symbols, while the input is the q-gram symbols of the compressed text. The benefits of this solution are that it is extremely simple to implement, does not need to generate different alignments of the pattern, and works in linear time w.r.t. the compressed text given our limit ($m \le 2q-1$) for the pattern length.

The standard Shift-Or automaton was presented in Sect. 1.3. Here we remind that the method updates the bit-vector d with each character of the text, and the table B (built in the preprocessing), having one bit-mask entry per alphabet symbol, is involved in the updates.

To apply Shift-Or straightforwardly to the considered problem would require that the text is decompressed to access the text symbols. However, it is possible to preprocess the table B to process q characters simultaneously. That is, the simulation step becomes $d \leftarrow (d << q) \mid B^q[C]$, where C is the codeword of a q-gram, and $B^q[C]$ is the transition mask for the corresponding (Cth) q-gram in the dictionary. The rationale behind this is that we have precomputed the q shift and `or` operations for the q characters of the q-gram $Dict[C]$ and stored the result into $B^q[C]$:

$$\begin{aligned} B^q[C] \quad \leftarrow \quad & ((B[Dict[C][0]] \ \& \ msk) << (q-1)) \mid \\ & ((B[Dict[C][1]] \ \& \ msk) << (q-2)) \mid \\ & \ldots \\ & (B[Dict[C][q-1]] \ \& \ msk), \end{aligned}$$

Alg. 30 DCPreProcShort$(P, q, Dict)$.

Input: pattern P, q, dictionary $Dict$
Output: match vectors $B^q[]$

```
1   for i ← 0 to σ − 1 do B ← ~0 >> (w − m)
2   for i ← 0 to m − 1 do B[P[i]] ← B[P[i]] & ~(1 << i)
3   for i ← 0 to |Dict| − 1 do
4       B^q[i] ← 0
5       for j ← 0 to q − 1 do B^q[i] ← (B^q[i] << 1) | B[Dict[i][j]]
6   return B^q
```

where msk has m lowest bits set and other bits zero. $B^q[C]$ therefore pre-shifts and bit-wise `or`s the state transition information for q consecutive original symbols, and the state vector d is then updated with this precomputation. Alg. 30 shows a pseudocode for computing B^q.

The search algorithm is now simple. The compressed text is scanned byte by byte, and each time we have parsed a whole codeword C, we execute the simulation step. In consequence of processing q characters in a single step, the implicit automaton has q accepting states, instead of only one. In other words, after the simulation step any of the bits numbered $m \dots m + q - 1$ may be zero in d, each indicating an occurrence of the pattern. Alg. 31 shows the pseudocode.

The size of the table B^q is $O(|Dict|)$ which is $O(\Sigma^q)$ in the worst case, but in practice much less (assuming the text is compressible). If this becomes an issue, we can preprocess B^q only for the $f \ll |Dict|$ most frequent codewords, and fall back to standard Shift-Or for the $|Dict| - f$ most rare

Alg. 31 DCSearchShort(T, P, q, B^q).

Input: Compressed text T, pattern P, q-gram match vectors $B^q[]$
Output: the number of occurrences

```
1   i ← 0; d ← 0; occ ← 0
2   qm ← ((1 << q) − 1) << (m − 1)
3   while i < n do
4       j ← i; C ← 0
5       while T[i] < c do
6           C ← C + T[i] * c^(i−j)
7           i ← i + 1
8       C ← C + (T[i] − c) * c^(i−j)
9       d ← (d << q) | B^q[C]
10      if (d & qm) ≠ qm) then
11          occ ← occ + popcount[~(d & qm) >> (m − 1)]
12      i ← i + 1
13  return occ
```

codewords. This requires that the text is locally decompressed for those codewords.

Bit-parallelism induces a new limit for the pattern length, that is, the algorithm works only for $m \leq w - q + 1$, where w is the number of bits in a computer word (typically 32 or 64). Combining this limit with the requirement $m \geq 2q - 1$, we have support for any pattern length assuming $q \leq \lfloor (w+2)/3 \rfloor$. This is a very reasonable limit, but if the text is *very* redundant and hence we would like to use longer q-grams, we can turn the Shift-Or method into filter and search only pattern prefixes of length $w - q$. This would require verification.

4.3.3 Experimental results

We have implemented the algorithms in C, and compiled with `gcc 3.4.1`. The test machine was a 2 GHz Pentium4 with 512 MB RAM running GNU Linux 2.4.20. For the experiments we used the following files: Dickens (the collected works of Charles Dickens, 10 192 446 bytes); XML (collection of XML files, 5 345 280 bytes); English, Spanish and Finnish versions of the Bible (4 486 219, 4 276 390, and 4 376 781 bytes, respectively); DNA of E.coli (4 638 690 bytes); proteins (5 050 292 bytes)[1].

Table 4.1 shows the compression ratios of different algorithms. We compared our approach to gzip and word based (spaceless) models using the same compressors as in the q-gram based algorithms. The prefix "W" in ETDC and (s,c)-DC means that word based model is used, while the prefix "q" indicates q-gram based model. The numbers include the sizes of the compressed dictionaries of words or q-grams. Our approach is clearly worse for natural languages, but the difference is smaller in the agglutinative languages, i.e., Spanish and Finnish. The number of different words is much larger than in English, and hence word based compression is less competitive. For DNA and proteins our approach gives good compression ratios (the results are very close to zero-order entropy of these texts). For the two proposed schemes we also show the used values of q and s, the range of *stoppers*, i.e., the byte values ending a codeword in (s,c)-DC. It is easy to notice that the used (optimal) values of q force s as high as possible for DNA (alphabet of 4 symbols) and proteins (alphabet of 23 symbols, including three special ones).

We compressed the q-gram dictionaries with `zlib` library (a variant of LZ77 compression)[2]. Table 4.2 shows the compressed dictionary sizes

[1] All test files available at http://szgrabowski.kis.p.lodz.pl/research/data.zip

[2] www.zlib.org

Table 4.1: Comparison of compression ratios

	Dickens	Bible English	Bible Spanish	Bible Finnish	XML	Ecoli	Protein
gzip –9	37.8%	29.5%	31.6%	32.9%	12.4%	28.0%	56.6%
W-ETDC	32.9%	34.4%	41.2%	41.5%	38.4%	N/A	N/A
W-(s,c)-DC	32.0% $s=205$	32.6% $s=226$	39.6% $s=214$	40.5% $s=202$	36.5% $s=231$	N/A	N/A
q-ETDC	47.3% $q=5$	44.9% $q=5$	46.9% $q=5$	48.0% $q=5$	32.9% $q=10$	33.3% $q=3$	64.0% $q=3$
q-(s,c)-DC	47.2% $q=5$ $s=145$	44.8% $q=5$ $s=155$	46.8% $q=5$ $s=150$	47.9% $q=5$ $s=145$	32.7% $q=10$ $s=165$	25.0% $q=4$ $s=254$	55.7% $q=2$ $s=254$

Table 4.2: Dictionary sizes and the numbers of unique q-grams for various files

	Dickens	Bible English	Bible Spanish	Bible Finnish	XML	Ecoli	Protein
Dict. size	390 672	202 114	217 345	228 618	255 462	353	538
q-grams	166 794	86 822	93 650	99 448	144 896	257	445

and the number of q-grams corresponding to Table 4.1 and the (s,c)-DC method. Table 4.3 presents how varying the parameter q affects the compression ratio on the example of Dickens file and ETDC coding. As it can be seen, $q = 4$ gives slightly worse results than $q = 5$. In practice we would like to use as small q as possible since it implies a smaller dictionary, making this scheme reasonable also for moderately sized texts, and triggers the faster of the two algorithms (Sect. 4.3.1) for shorter patterns.

Table 4.4 shows the decompression times, excluding I/O time. For ETDC and (s,c)-DC the parameters are as in Table 4.1. For natural languages the

Table 4.3: The effect of varying q on the dictionary size and the overall compression (Dickens/ETDC)

q	2	3	4	5	6
q-grams	2 343	16 809	64 950	166 794	321 950
Dict. size	3 747	35 110	142 664	390 650	792 351
Compression	63.3%	55.9%	48.7%	47.3%	48.0%

Table 4.4: Comparison of decompression times

	Dickens	Bible English	Bible Spanish	Bible Finnish	XML	Ecoli	Proteins
gunzip	0.26	0.11	0.09	0.11	0.04	0.12	0.17
W-ETDC	0.30	0.12	0.15	0.16	0.16	N/A	N/A
W-(s,c)-DC	0.30	0.13	0.14	0.15	0.16	N/A	N/A
q-ETDC	0.33	0.13	0.12	0.13	0.09	0.08	0.09
q-(s,c)-DC	0.34	0.11	0.11	0.12	0.08	0.04	0.11

different algorithms have similar performance. For XML gzip outperforms the others, but clearly loses on DNA and proteins. Note that on the XML file, where the word based methods can be used, the q-gram based algorithms are almost twice faster, partly because of the better compression they provide for this case.

Table 4.5 shows search times in seconds for Dickens, DNA, and proteins. *Direct search* means searching directly in the compressed text; *Decompress and search* means first decompressing the text, and then searching with the same algorithm used for the direct search; *Search decompressed* is the running time of BNDM [NR98] algorithm applied to decompressed text (no decompression time; used also for short patterns). The timings include all the preprocessing for each algorithm, but not the I/O times. The patterns were randomly picked from the texts.

For the long patterns we used minimum pattern lengths that produced compressed pattern lengths of at least 2. For short patterns we used maximum pattern lengths not yet supported by the algorithm for the long patterns. That means, for example, that for Dickens file the short patterns had $2q - 2 = 6$ characters. The long patterns in this particular experiment were those 7-character excerpts from the text which had passed the "encoding test". For example, `the_cat` would be rejected as according to one of the alignments the encoding would be reduced to the encoding of the 4-gram `the_`, whose length is one byte only. On the other hand, the pattern `trouble` produces at least two-byte encoding for each alignment, so is accepted as a "long" one. Deviating from these choices would make our method faster in comparison. For the timings for Dickens we used $q = 4$ and ETDC, as it gives much better performance than $q = 5$, while the compression ratios are almost equal. For example, "Decompress and search" is faster with $q = 4$ than plain decompression with $q = 5$. For DNA and proteins we used (s, c)-DC with $q = 4$ and $q = 2$, correspondingly.

Table 4.5: Search times in seconds for short and long patterns

	Direct search		Decompress and search		Search decompressed	
	short	long	short	long	short	long
Dickens	0.0481	0.0082	0.1852	0.164	0.0086	0.0115
E.coli	0.0043	0.0096	0.0631	0.062	0.0161	0.0135
Proteins	0.0195	0.0057	0.1362	0.127	0.0167	0.0113

4.4 A simple technique for denser encoding of static texts

In this section we present and experimentally verify a very simple idea increasing the effectiveness of search-supporting byte codes, under the assumption that the text we search in is static. This idea is applicable to both word and q-gram model. In further considerations we assume that the longest codewords have 3 bytes. This is a reasonable assumption for many data types, including natural language texts. Also, to fix attention, we talk about "words" as symbols of the text (albeit those could be q-grams instead). Our proposal is based on a very simple observation: some pairs of 1-byte encoded words, although globally frequent, may never go adjacently. For example, in English it is unlikely to come across phrases like "a the", "the a", "of of" or "from of", although all the words in the listed phrases belong to the most frequent ones. Consequently, the pairs of bytes representing those 2-word phrases may never occur in the encoded text. If this is a case, the most frequent words among those encoded on triples of bytes may obtain 2-byte codewords instead, namely those corresponding to the non-occurring pairs of most frequent words. Let us assume (s, c)-dense coding. There are s^2 pairs of words encoded with one byte each. The text must be scanned and the non-occurring pairs found (this can be implemented efficiently with e.g. a hash table). Note that, unfortunately, this idea requires the text to be static, i.e., its application is somewhat limited. Obviously, when codewords longer than 3 bytes occur, this idea remains applicable, but the amount of potential optimizations grows; for example, non-occurring pairs of 1-byte and 2-byte codewords may be used to shift some words encoded on 4 bytes to the group of words encoded of 3 bytes. We have not considered how to solve those issues optimally, however we believe those are mostly of theoretical interest. The set of non-occurring word pairs must be stored with the encoded text. As it is not very small typically, we found that using s^2 bits to indicate those pairs is more succinct than listing them directly on 2 bytes

per pair. For example, with ETDC ($s = 128$) the resulting overhead is 2 KB. Finally, we note that this idea makes the coding no longer pure order-0, and as such resembles PETDC a bit [BFNP06]. As it will be seen in Sect. 4.4.1, our idea gives much more humble improvements than PETDC, but is also much easier in implementation and faster in coding and decoding.

4.4.1 Experimental results

We have implemented the described idea tested its effectiveness in the word based and q-gram based models. For the word model we used the following files (also used in Sect. 4.3.3): Dickens (the collected works of Charles Dickens, 10 192 446 bytes); XML (collection of XML files, 5 345 280 bytes); English and Spanish versions of the Bible (4 486 219 and 4 276 390 bytes, respectively). In the XML file, the end-of-line symbols were inconsistent, so before further processing we converted 2-byte EOLs to #10. Additionally, we took a couple of larger files from the Pizza & Chili Corpus [3]: those are 50 MB excerpts (prefixes) of English texts, source codes, and XML, and are referred to in tables as English50, Sources50 and XML50. For the second experiment, with q-gram models, we additionally took 50 MB datasets of MIDI pitch values and proteins from Pizza & Chili, which are denoted here as Pitches50 and Proteins50. The compression results are presented in Table 4.6 and Table 4.7, respectively. For compression on words, the spaceless model was used. The vocabulary itself was zlib-compressed (with the maximum setting, -9), following [FG06d]. Clearly, better dictionary encoding schemes are possible but this requires a separate study. The words, given as input for zlib compression, were ordered according to their rank and separated with #1 symbols, which did not occur in any of the given files. Identical vocabulary compression was used for q-grams but this time the separators were not used for obvious reasons. The improvements on words are rather disappointing, especially for ETDC. In one case (the English Bible) the modification resulted even in a tiny loss (about 20 bytes). This happened because the sub-vocabulary of 3-byte encoded words had only around 2000 entries for this file, and thus not the whole gained code space could be utilized.

With (s, c)-DC, the gains are greater which is due to two reasons. One is that the parameter s is greater than 128, so among the s^2 word pairs there are more such that never occur adjacently in the text. The second reason is that with a greater s, there are more items in the group of 3-byte encoded

[3] http://pizzachili.dcc.uchile.cl/

Table 4.6: Denser encoding. Comparison of compression ratios in word based schemes.

	gzip	bzip2 -9	W-ETDC	denser W-ETDC	W-(s,c)-DC	denser W-(s,c)-DC
Dickens	37.79%	27.47%	34.80%	34.65%	33.79% ($s = 206$)	33.24% ($s = 206$)
Bible, English	29.54%	21.15%	32.43%	32.44%	31.12% ($s = 219$)	30.92% ($s = 219$)
Bible, Spanish	31.59%	23.28%	39.43%	39.19%	38.33% ($s = 205$)	37.76% ($s = 205$)
XML (5 MB)	12.39%	8.25%	37.30%	37.11%	35.52% ($s = 228$)	35.03% ($s = 228$)
English50	37.52%	28.40%	35.99%	35.74%	35.26% ($s = 197$)	34.53% ($s = 197$)
Sources50	23.29%	19.78%	46.24%	45.78%	45.22% ($s = 201$)	44.05% ($s = 201$)
XML50	17.23%	11.17%	35.66%	35.35%	35.19% ($s = 202$)	34.47% ($s = 202$)

words, hence the most frequent of them have more occurrences than it used to be with ETDC. The q value most often used in the experiment was 4. In [FG06d] we found it the best compromise for natural language texts; $q = 5$ may provide still a bit better compression (in spite of the bloated dictionary), but the minimum pattern length for which the faster of the two possible algorithms is selected, is then greater. For some data types other values of q are more suitable though. For very redundant XML data we present results for $q = 4$ and $q = 10$. Proteins and MIDI pitches are much less compressible, hence $q = 3$ was also tested for them. In both tables, gzip and bzip2 compression ratios are also given, for a reference. The implementations were written in Python and tested on an Athlon64 3000+ with 2 GB of main memory. In order to perform pattern search speed measurements, needed for a full evaluation of the idea, the programming language of the implementation must be switched to e.g. C++ in future works.

4.5 Conclusions and future work

We have presented a compression algorithm for arbitrary data which enables pattern search with Boyer–Moore skips directly in the compressed represen-

Table 4.7: Denser encoding. Comparison of compression ratios in q-gram based schemes.

	gzip	bzip2 -9	q-ETDC	denser q-ETDC	q-(s,c)-DC	denser $q-(s,c)$-DC
Dickens, $q = 4$	37.79%	27.47%	49.01%	47.90%	48.69% ($s = 166$)	47.08% ($s = 166$)
Bible, English $q = 4$	29.54%	21.15%	47.03%	46.33%	46.47% ($s = 179$)	45.33% ($s = 179$)
Bible, Spanish $q = 4$	31.59%	23.28%	48.73%	47.91%	48.21% ($s = 177$)	46.88% ($s = 177$)
XML (5 MB) $q = 4$	12.39%	8.25%	48.59%	47.81%	47.50% ($s = 199$)	45.83% ($s = 199$)
XML (5 MB) $q = 10$	12.39%	8.25%	35.91%	35.27%	35.81% ($s = 166$)	34.90% ($s = 166$)
Proteins (5 MB) $q = 3$	56.63%	53.27%	64.02%	64.06%	62.50% ($s = 221$)	62.61% ($s = 221$)
Proteins (5 MB) $q = 4$	56.63%	53.27%	70.33%	65.62%	70.33% ($s = 131$)	65.43% ($s = 131$)
English50 $q = 4$	37.52%	28.40%	49.54%	48.17%	49.25% ($s = 162$)	47.30% ($s = 162$)
Sources50 $q = 4$	23.29%	19.78%	56.03%	54.03%	55.87% ($s = 154$)	53.21% ($s = 154$)
XML50 $q = 4$	17.23%	11.17%	43.02%	41.92%	42.17% ($s = 182$)	40.28% ($s = 182$)
XML50 $q = 10$	17.23%	11.17%	34.40%	34.01%	34.17% ($s = 180$)	33.50% ($s = 180$)
Pitches50 $q = 3$	30.59%	36.12%	71.48%	69.06%	71.34% ($s = 147$)	68.28% ($s = 147$)
Pitches50 $q = 4$	30.59%	36.12%	71.92%	70.11%	71.89% ($s = 140$)	69.77% ($s = 140$)
Proteins50 $q = 3$	47.39%	45.56%	63.99%	64.00%	62.73% ($s = 217$)	62.69% ($s = 217$)
Proteins50 $q = 4$	47.39%	45.56%	65.40%	61.02%	65.39% ($s = 133$)	60.72% ($s = 133$)

tation. The algorithm is simple and the experiments validate the claim for its practicality. For natural texts this scheme, however, cannot match, e.g., the original (s, c)-dense code in compression ratio, but this is the price we pay for removing the limitation to word based textual data.

Several issues require further investigation. Experiments with succinct dictionary representation are definitely needed, as the number of entries in the dictionary grows exponentially with increasing q (at least as long as the typical condition $q \ll n$ is satisfied). Of some importance is devising a quick entropy estimation method across many tested values of q, as this speeds up preparing the compressed files. Currently we calculate the entropy for each q separately. Higher compression could also be reached if, aside from q-grams, we allow text tokens shorter than q characters. Together with some parsing rule(s), such a relaxed variant should allow to extract more occurrences of popular short sequences of characters. Another benefit of more flexible parsing should be the possibility for fast and simple updates to T (insertions or deletions in text). Finally, the presented technique seems to have potential for approximate pattern matching, e.g., in combination with the technique from [FN04], which may suffer from preprocessing costs in non-compressed version.

Additionally, we introduced a simple idea for improving the effectiveness of byte codes, if the text is static. Experiments showed that this concept is more beneficial in case of q-gram based rather than word based compression. In no case the gains are spectacular, but the idea can be defended as being very simple both conceptually and programmatically, and not affecting search or decompression speed.

Chapter 5

Compressed full-text indexes

Online searching for a pattern of length m in a text of length n, both over an integer alphabet Σ of size σ, requires accessing at least n/m characters of the text. Compressing the text may reduce (on average) the number of character comparisons by a factor, but does not change the overall picture. Much more drastic improvements are possible only if extra information for a given text is built in the preprocessing.

This extra information, called an *index*, is a subject of scientific investigations since the 1970s, but only in the last decade the research has been greatly boosted, with the advent of so-called compressed indexes, which often replace the text with its compressed representation adding search capabilities, and still handle simple queries very effectively.

Our contributions in this area include a proposal of a very simple FM-index variant, called FM-Huffman [GMN04b] and its more efficient variations offering various tradeoffs [GMNS05, PGNS06, GNP$^+$06]. As a byproduct, we present a simple constant-time implementation of the *selectNext* query (useful in our index) [GMN04a] and also a simple succinct constant-time *select* implementation (unpublished). We have also participated in designing practical implementations of *rank* and *select* structures [GGMN05].

5.1 Motivation and problem aspects

Compressing the text for online searching, discussed in the previous chapter, may speed up the search process in practice, and is especially recommended when the text is large enough to be stored on disk rather in the main memory, since in such a case compression significantly reduces the number of I/O operations, but in theory the number of character comparisons is not

reduced, or reduced only a little. This implies from the fact that the ratio n'/m', where n', m' are the lengths of the compressed text and the compressed pattern, respectively, is approximately equal to the "original" ratio n/m, at least on average. We can also say that online pattern search (no matter if the text is compressed or not) is linear, in the sense that doubling the text (approximately) doubles the pattern search time.

Obviously, there are important scenarios in which the text does not change over time, or requires updates only infrequently, but searching operations are performed often. The need for fast search is especially burning in multi-user environment, e.g. through the Web. It is conceivable that sometimes several queries are sent to the server in a second, and then referring to linear scan over the text would simply be much too slow, especially if the text collection is on the order of hundreds of megabytes, or gigabytes.

To speed up search operations significantly, one has to build an *index* over the given text. The index is carefully extracted additional information which helps to answer quickly some or all of the following questions:

- does a given pattern P occur in text T (at least once)?
- how many times does P occur in T?
- in what locations of T does P occur?
- what are the text snippets around each occurrence of P in T?

The first of the listed questions is called an *existential query* and is least informative. We will not care for this kind of query in the following considerations. The second one, a *counting query* is more useful. Trivially, knowing how many times P occurs in T answers also the existential query. The third question is known as a *reporting* or *locating query.* As we are going to see, with some indexes knowing the pattern count in the text implies that we can straightforwardly report the locations, but with other indexes the locate operation is harder. The last requirement in the list above is called a *display query.* Note that if we can tell the locations of the pattern, and we have free access to the original text (that is, we can access $T[j]$ for any j in constant time), then handling display queries poses no extra difficulty than handling locating queries.

Apart from those basic features, there can be other functionalities supported with text indexes. For the first thing, note that the assumption of static text is rarely fulfilled in practice (especially for huge collections, and those need indexing most!). For example, a newspaper database grows with each new issue, an online book library grows with every newly added book, and badly scanned and OCR'ed books are replaced with more correct versions (this is an example of both insertions and deletions in the text collection), a medicine database tends to grow quickly, but also some medicines

are removed from market (as e.g. potentially harmful) and their descriptions should also be removed from the database, and so on. Of course, some of the mentioned applications could be better addressed using structured text (e.g., hierarchical, in XML format) but this goes out of the scope of our thesis. In the following we assume unstructured, "flat" text only. Fortunately, as we will see later, some of the techniques developed for indexing flat text can also be adapted to indexing tree structures (e.g., XML).

An index is called *dynamic* if it can handle updates (insertions and deletions) to the text easily. Some indexes have only partial dynamic capabilities (e.g., appending text at its end, but not in arbitrary locations), but for many data structures all that we can do if the text changes is to build the index again from scratch. Consequently, the server storing the text and its index must be temporarily unavailable, or, at best, part of its computing power and other resources are very often used for rebuilding the index.

Another precious feature of an index is approximate matching support [MN05a]. Unfortunately, this is a very hard problem, and existing solutions are still immature and often heuristic [CGL04, NST05, MN05b, NC06, CO06]. We do not discuss indexed approximate matching in this chapter.

Most classic text indexes are several times larger than the text itself, which is a severe problem if the text is very large. To address this issue, many compact indexing data structures have been proposed in the last few years (most of this chapter is dedicated to those structures). Still, for huge enough texts, even the most succinct indexes, based on compression techniques, cannot make it possible to keep everything in the main memory. Therefore an important question appears, how to limit the number of I/O operations during the search. The indexes that solve this problem better than in a naïve way, are called *external* indexes. A prominent application where text sequences are often huge is bioinformatics. For example, the human genome takes nearly 3 GB of space, assuming each base pair occupies one byte [SS01].

In this chapter we consider *full-text indexes*, that is, structures which are able to find any subsequence of T. An alternative to full-text indexes are word-based indexes, very useful in their domain, i.e., natural languages, but not easy to be adapted to other kinds of text. Also, there are several important human languages without a clear segmentation into words (e.g., Chinese, Korean, Japanese), and even with some European languages this approach to indexing causes problems. These issues were discussed in more detail in Chap. 4. A canonical example of word-based indexing is the concept of the inverted index, also presented in Chap. 4.

We also note that the interest in *succinct data structures* is not limited

to texts, but also several results for e.g. unlabeled graphs [MR97], labeled graphs [BAHM07], and trees [MR97, GRRR04] exist. However, presenting those structures would go out of the scope of this work.

5.2 Classic indexing data structures

The first important text indexing structure was the *suffix tree* (ST) [Wei73]. The suffix tree is a trie (also known as a digital search tree) whose string collection is the set of all the suffixes of a given text, with an additional requirement that all non-branching paths of edges are converted into single edges. Typically, each ST path is truncated as soon as it points to a unique suffix. A leaf in the suffix tree holds a pointer to the text location where the corresponding suffix starts. As there are n leaves, no non-branching nodes and the edge labels represented with pointers to the text, the suffix tree takes $O(n)$ words of space, which is in turn $O(n \log n)$ bits. The structure can be built in linear time, which was earlier achieved for constant alphabets [Wei73, McC76] (even in an online manner [Ukk95]), and then also for integer alphabets [Far97]. Moreover, the linear-time construction algorithms can be fast not only in theory, but also in practice [Gri07]. Searching for all occ pattern occurrences in the suffix tree may take only $O(m + occ)$ time in the worst case, no matter how large the alphabet is, if perfect hashing is used to traverse over a node's children in $O(1)$ time per each, but in a more practical implementation the starting characters of the edges outgoing from a given node are simply kept in an unsorted array. The worst-case time grows to $O(m\sigma + occ)$ but in practice the size of most nodes is quite small and linear scan over them is fastest. Something between those extremes is to keep the children of a node lexicographically sorted, and then a binary search in each node leads to $O(m \log \sigma + occ)$ worst-case time.

The main problem with the suffix tree are its large space requirements, even in the most economical version [KB00] reaching almost $9n$ bytes on average (and $16n$ in the worst case), plus the text, for $\sigma \leq 256$, and even more for large alphabets. Most implementations need $20n$ or more space.

To address this important weakness of the otherwise powerful structure, in the beginning of the 1990s Manber and Myers proposed a *suffix array* (SA) [MM90, MM93], a structure independently discovered also by Gonnet et al. [GBYS92], under the name of the PAT array. The suffix array is a ordered collection of n pointers to text suffixes, where the order corresponds to lexicographic ordering of the sequences (i.e., the suffixes) the pointers store references to. In a typical, convenient implementation (4-byte point-

ers, 1-byte characters), a suffix array needs $5n$ bytes of space, including the text. Searching in the suffix array consists in two binary searches: one is to find the beginning of the interval of the suffixes starting with pattern P, and the other to find the end of this interval. The comparisons refer both to the array of indexes and the underlying text As in each comparison up to m suffix characters need to be accessed, the worst-case search time is $O(m \log n)$ but it can be argued that the time is only $O(m \log_\sigma n + \log n)$ on average. There are several possibilities to speed up the search in a suffix array, either in theory or in practice. One is to keep an extra array of n values (of $\log n$ bits each, if we want this idea to work with an arbitrarily large m) storing the lengths of the common prefixes (LCP) between the current suffix and the previous suffix on the search path (which is uniquely defined) [MM93]. Thanks to the LCP table, the search time complexity goes down to $O(m + \log n)$ but in practice this idea not always helps [FF07]. Even without the LCP table, a similar technique can be used, namely monitoring the number of matching characters between P and both ends of the suffix interval we are currently in [MM93, FF07], to save on the pattern prefix comparisons, but in the worst case it does not help anything. To avoid doubling the space when the LCP table is added, a compromise variant is possible. The idea is to keep the LCP values only for $n/\log_2 n$ regularly spaced suffixes [GS03], thus losing its support in the last $\log\log n$ steps of the binary search. In this way the space for the LCP table drops from $n \log_2 n$ bits to n bits (in practice, this is an improvement from $4n$ bytes to $n/8$ bytes) and the time complexity changes to $O(m \log\log n + \log n)$. Another idea is to start the search from a hopefully narrow enough interval due to precomputed array of σ^k words [MM90, FF07]. In this way, the binary search over the SA is restricted to searching for the pattern suffix of $m - k$ characters. In a similar spirit, Baeza-Yates et al. proposed an external SA variant [BYBZ96], in which the suffix array is kept on disk, but the first symbols of regularly sampled suffixes, in their sorted order, are also cached in main memory, to save on the number of I/O operations.

From the theoretical point, an interesting question is whether some search strategy in the pure SA (without the LCP table or other auxiliary structures) can lead to better complexity than $O(m \log n)$ in the worst case. The answer is positive [Hir78b], the worst-case complexity can be improved if some asymmetric (instead of the standard halving) suffix selection in the current search interval is chosen, and the optimal algorithm, with a precise (complex) worst-case formula, was found by Andersson et al. in 1994 [AHHP94, AHHP01].

Interestingly, it was recently shown that sorting an array of strings in lexicographical order is *not* the optimal arrangement for searching an m-character pattern [FG04]. A different permutation, which can be efficiently obtained from the sorted set of strings, leads to the $O(m + \log n)$ worst-case search. If applied for the set of all suffixes of a text, this matches the search complexity of the suffix array with the LCP table, but without the extra space needed for the LCP. It is hard to say how practical this surprising discovery can be, since we are not aware of any experimental evaluation of this idea yet.

Some extra tables were added to the SA in a proposal by Abouelhoda et al. [AKO04], under the name of the extended suffix array (ESA). This structure achieves $O(m\sigma + occ)$ search time which is quite practical for DNA or other cases when the alphabet is very small. Unfortunately, the ESA takes $13n$ bytes (including the text), which is not so competitive to the suffix tree. It seems possible to decrease its search complexity to $O(m \log \sigma)$ but for the price of using even more space.

In a recent thorough work on engineering the ST and ESA (and other indexes) implementations [Gri07], Grimsmo observed that a ST variant based on dynamic arrays rather than sibling linked lists is up to 20 times faster in the construction and 10 times faster in search, but the price is that the array-based structure needs about 20% more space. He also noticed that if the number of pattern occurrences is large, then reporting them can be many times faster with a suffix array than with a suffix tree, but the suffix tree becomes the winner if the number of matches is small.

Building a suffix array is also an active research area. Standard sorting algorithms (e.g., merge sort) are inappropriate in this setting as they don't make use of the specific properties of text suffixes, and are thus practically slow and with superquadratic worst-case behavior. Yet in the seminal work by Manbers and Myers [MM93] an algorithm with $O(n \log n)$ worst-case complexity was presented, which is optimal for general alphabets. This algorithm is, however, slow in practice [Deo03, Sect. 4.3.1]. A much faster algorithm with the same complexity was given in 1999 by Larsson and Sadakane [LS99], a variant of which is currently one of the components of the suffix sorting algorithm implemented in bzip2 [Sew06], a popular compressor based on the BWT transform [BW94]. Interestingly, the fastest algorithms for typical inputs [MF04, Deo03, MP08], beating the Larsson–Sadakane algorithm about two or three times, are very slow in the worst case (even superquadratic). Most of them, however, have some protection against "popular" kinds of pathologies (e.g., long runs of the same symbols which occur often e.g. in human-made graphics).

The suffix array can be obtained easily from the suffix tree by writing down its leaves in left-to-right order (assuming that the childen in ST nodes are lexicographically ordered). Hence the ST over an integer alphabet can be constructed in $O(n)$ time, and the in-order traversal over the leaves is also linear in time, the suffix array construction can also be $O(n)$ in the worst case, a fact that was realized since the origin of the SA structure. Still, such a construction algorithm is not only very space consuming, but also slow. It was long an open question, whether a SA can be built directly in linear time. The positive answer was given in 2003, when as many as three different $O(n)$-time algorithms were presented [KS03, KSPP03, KA03]. All those algorithms have two drawbacks: they are relatively slow in practice and need an extra set of n pointers (some of them even more) to operate (i.e., they use $9n$ or more bytes in a real implementation, as opposed to $5n$ bytes used by many other algorithms). Hence a question whether a practical linear-time suffix sorting algorithm exists, remains open.

5.3 Early compact text indexes

All the indexes presented in the previous section need $O(n \log n)$ bits of space, with the constant at least 1, plus $n \log \sigma$ bits for the text itself. In a typical setting this translated to $5n$ bytes (achieved by the plain suffix array, the most economical from the presented indexes), which can be unacceptable for very large texts.

One of the early proposals for decreasing the storage occupied by the suffix tree was given in 1996 by Kärkkäinen and Ukkonen [KU96b]. Their idea was to sample only n/k suffixes of the text, in regular intervals, and build the ST over those suffixes only. The parameter k may be 4, for example. In this way, the space occupancy is significantly reduced, but the search gets slower. This is because one has to test for the occurrence of $P[0 \ldots m-1]$, and $P[1 \ldots m-1]$, etc., and $P[k-1 \ldots m-1]$ in the set of indexed suffixes, and for $k-1$ out of the k cases the found matches must be verified with the truncated prefix of P. In this way, the search gets at least k times slower, and the worst case behavior is disastrous. Also, it is required that $m \geq k$.

An akin idea is to sample the text suffixes on word boundaries, again used for the first time with the suffix tree [ALS96], and later also for DAWGs, compact DAWGs, and suffix arrays. Since those structures are not devised for full-text searching, we omit details, but a curious reader can be directed to [FF07], and the references therein. We only note that sparse indexes on words are easier to be built in both linear time (in n) and only $O(n_w)$ working

space, where n_w is the number of words, than corresponding indexes with an arbitrary (or even regular) sampled set, for whose analogous solutions are still unknown.

There are a number of other text indexes proposed in the 1990s, which rival with the suffix tree in space occupancy, but are not competitive to the suffix array in this aspect. Most of them are briefly reviewed e.g. in [GKS03].

The beginning of the new era in compact indexes can be attributed to the work of Kärkkäinen and Ukkonen [KU96a], who proposed in 1996 an index based on the Lempel–Ziv compression, with $O(n \log \sigma)$ bits of space in the worst case, and sublinear search time. What is most interesting, the size of that index was proportional to the kth-order entropy of the text, as it was linear in the number of blocks in the Lempel–Ziv parsing [LZ76] of the text. It means that not only the index size was dependent on the text it was built for, but also that the relation could be clearly expressed in well understood and widely used notions of the information theory. From these reasons, the Kärkkäinen–Ukkonen index can be called a *compressed index*. Still, the index couldn't get rid of the text itself.

A completely different succinct structure was presented in 2000 by Mäkinen, under the name of the compact suffix array [Mäk00, Mäk03a]. The author introduced the term of a *self-repetition* in a suffix array, which is understood as an interval of indexes in the SA with values shifted by 1 compared to another interval of indexes of the same size. The self-repetition can thus be replaced by a link to the corresponding area. Interestingly, in the original papers no entropy-related bound on the size of the index was given, but in a later work [MN04a] Mäkinen and Navarro showed it is bound within $O(H_k n \log n)$ bits, not counting the text which needs another $n \log \sigma$ bits. Akin to this index is the much more recently developed locally compressed suffix array (LCSA) [GN07b], which achieves $O(H_k \log(1/H_k) n \log n + n)$ bits, apart from the text itself, and handles counting queries in $O(m \log n)$ time, and locate queries in $O(occ + \log n)$ time for occ occurrences of the pattern.

In the same year 2000 another succinct index was proposed, dubbed the compressed suffix array (CSA) [GV00]. The search mechanism in CSA mimics the behavior of the classic SA, but instead of the array $\mathcal{A}$ of suffixes another array is used, with values of the so-called function Ψ, which is also a permutation of the numbers $[0 \ldots n-1]$, but with an interesting property of being compressible. The function Ψ maps suffix $T_{\mathcal{A}[i]\ldots n}$ to the next suffix in the text, $T_{\mathcal{A}[i]+1\ldots n}$, and is defined by the formula $\Psi(i) = i'$ such that $\mathcal{A}[i'] = (\mathcal{A}[i] + 1) \bmod n$. Fig. 5.3 shows an example. The property of the Ψ function that makes it compressible is that its successive values form (at

most) σ increasing sequences. More specifically, the Ψ values are increasing within each area of $\mathcal{A}$ which points to suffixes starting with the same symbol from the alphabet. Instead of storing the raw values of Ψ, we can use a well-known technique of differential coding (also known as delta coding or gap encoding). Apart from efficient encoding of the Ψ array, we also need random access to the sequence. A standard solution could be to sample every kth value from the list (for $k = \log n$, for example), encode its absolute value, and store it in an extra table, together with a pointer to its position in the compressed sequence. Accessing an arbitrary value of Ψ would be accomplished by finding the nearest absolute value and decoding all the successive values until the required one, which can be done in $O(k)$ overall time. In fact, better solutions are possible. Sadakane [Sad02] presented a CSA variant using $nH_0 \log\log n + O(n \log\log\sigma)$ bits, plus $n \log\sigma$ bits for the text being kept verbatim. Locating a pattern is clearly obtained in $O(\log\log n)$ time, and displaying the context of ℓ characters around a found pattern needs only $O(\ell)$ time, since the text is available in the explicit form. More interestingly, yet in 2000 Sadakane [Sad00] presented another CSA variant which did not require the text itself. The index space was essentially expressed in terms of order-0 entropy, counting was performed in $O(m \log n)$ time and locating a pattern needed $O(\log^{\varepsilon} n)$ time, where $0 < \varepsilon n \leq 1$ was a tradeoff between locate time and the actual index space. Later works by Grossi, Gupta and Vitter [GGV03, GGV04] showed that the space within the CSA framework may be decreased to optimal (apart perhaps lower order terms), i.e., nH_k bits, but the counting complexity has an additive factor of at least $\log^3 n$. Detailed tradeoffs between various existing variants are summarized in [NM07, Sect. 8.2].

5.4 Basic concepts of the FM-index

The compact suffix array of Mäkinen and the compressed suffix array of Grossi and Vitter were very interesting and quite practical concepts but the breakthrough was yet to come. The breakthrough was achieved yet in the year 2000, and it was the paper by Ferragina and Manzini, where they presented the first *self-index* (a few months ahead of Sadakane's work on CSA), that is a compressed index allowing to access an arbitrary subsequence of text T without storing (explicitly) T itself! Arguably, this was one of the greatest discoveries in algorithmics in the recent years. The Ferragina–Manzini index is based on the Burrows–Wheeler transform (BWT) [BW94], used earlier for compression, and was dubbed the FM-index (according to

i	$\mathcal{A}[i]$	Ψ	suffix $T_{\mathcal{A}[i]\ldots n}$
1	12	12	$
2	5	4	-a-tete$
3	7	11	-tete$
4	6	3	a-tete$
5	11	1	e$
6	4	2	e-a-tete$
7	9	9	ete$
8	2	10	ete-a-tete$
9	10	5	te$
10	3	6	te-a-tete$
11	8	7	tete$
12	1	8	tete-a-tete$

Figure 5.1: CSA example. T = tete-a-tete$

the authors, we should believe that the letters in the name stand for "fast" and "minute"). To understand how FM-index works, it is necessary to know the mechanism of the BWT. It is explained in the following subsection. An excellent survey of BWT-based and other full-text compressed indexes was written by Navarro and Mäkinen [NM07].

5.4.1 Burrows–Wheeler transform

As usual, we assume that to text T a unique symbol \$ will be appended, and that \$ is lexicographically smaller than all the other symbols in the alphabet. The Burrows–Wheeler transform [BW94] considers all cyclic shifts of the string $T\$$ and sorts them lexicographically. Because of the extra symbol \$ in all the cyclic shifts, the resulting order of sequences is equivalent to the output of any suffix sorting algorithm. The sorted sequences are placed in a conceptual matrix M of $(n+1) \times (n+1)$ size. The first column of this conceptual matrix will be called column F, while the last column of the matrix will be called column L. Fig. 5.4.1 illustrates.

The BWT output is only those two columns, plus one extra integer from 0 to n. As the cyclic shifts of the text are sorted, the column F must be formed of $\sigma + 1$ runs (the alphabet has been augmented with the \$ symbol, hence its size is now $\sigma + 1$) of identical characters, and can be straightforwardly represented in $O(\sigma \log n)$ bits (which is on the order of 1 KB in practice). Interestingly, the column L, which in fact is a permutation of the text, is

F L	F L
BABOON$	$BABOON
ABOON$B	ABOON$B
BOON$BA	BABOON$
OON$BAB	BOON$BA
ON$BABO	N$BABOO
N$BABOO	ON$BABO
$BABOON	OON$BAB
before BWT	after BWT

Figure 5.2: BWT example. T = BABOON$

also typically compressible (although, of course, not as well as the column F). The column F tends to have runs of same characters, and also often only a few characters – recurring again and again – in local areas, and those features are the premise of all BWT-based compression algorithms [BW94, Fen96, BK00, Deo02, Deo05, FGM06, Abe07].

The column F is also called the BWT sequence of text T, or in short BWT(T).

The beauty of the transform lies in its reversibility. Having got the columns F, L, and one additional integer, it is possible to recover T in linear time. The additional number is the position of the text terminator \$ in column L. The corresponding symbol in column F (the first B in Fig. 5.4.1 on the right) is the starting character of T.

Since the seminal work of Burrows and Wheeler [BW94], BWT has been used as a working horse for compression. The fact that the BWT sequence tends to contain long runs of identical characters can be explained by little diversity within a single context. From earlier compression research, especially on *prediction by partial matching* (PPM) coders [CW84, CT97], it was known that not only that longer contexts contain fewer distinct symbols than shorter contexts, but also that the "diversity" of symbol occurrence within individual contexts diminishes if the context length increases. The vague notion of diversity can be precisely defined as the 0th-order entropy, or, to avoid some technical problems, the *0th-order modified empirical entropy* introduced by Manzini in [Man01]:

$$H_k(T) = \begin{cases} (1/n)\sum_i n_i \log(|T|/n_i), & k = 0, \\ (1/n)\sum_{|w|=k} H_0(w_s)|w_s|, & k \geq 1. \end{cases} \tag{5.1}$$

Sorting the cyclic shifts of the text results in grouping similar contexts:

the longer the prefix that two shifts of T have in common, the closer they are in the conceptual matrix, and hence the closer their preceding characters are in column L. Note that, opposed to most PPM compression schemes, in BWT there is no limitation of the context length, which is especially helpful for compressing very redundant data. The lack of constraint on the maximum context length make BWT-based compressors similar to PPM* [CT97], a compressor from the PPM family with arbitrarily long contexts. However, PPM* struggles with large memory consumption and is slow, while most BWT-based compressors are relatively fast, especially in decompression. The benefits of the BWT approach come at a cost though: the actual character contexts are unknown (they cannot explicitly be taken into account during the coding, since the decoder would be unable to work), apart from order-1 contexts which can be obtained from F column. Recently it was discovered that some rudimentary knowledge about the contexts in BWT can be extracted [Deo05, Man04], but the loss in speed in both compression and decompression seems to be impractical for the mediocre improvement in compression ratio.

In most schemes, the post-BWT modeling and coding is simpler. For example, in the popular compressor bzip2 [Sew06], the BWT output is sequentially transformed by a second-stage transform, namely the move-to-front (MTF) algorithm [Rya80, BSTW86], which maintains a queue of σ alphabet symbols in order of their last occurrence in the processed so far part of the BWT sequence. The MTF flattens the symbol code distribution, resulting in for example about $n/2$ zeros for a typical English text, and dominance of other low values in the rest of the encoded sequence. The MTF output in bzip2 is then run-length compacted, and finally compressed with Huffman coding.

Other ideas attempting to improve the bzip2 performance are, among others:

- replacing MTF with another second-stage transform [BKS99, Bin00, Deo02, Abe07],
- using sophisticated modeling and/or coding of the output of the second stage transform [BK00, Deo02],
- applying data-specific preprocessing of the input, e.g., oriented for natural language texts [Gra99, AT05, SGD05],
- partitioning heterogeneous input data before further BWT and the following stages [Gre04], which makes sense e.g. for binary data or TAR'red non-uniform file collections.

Also, the compression ratio of bzip2 is seriously hampered on large files, since the maximum block size for BWT is there limited to 900 KB.

Most interest in BWT-based compression has been practice-oriented, where the proposed techniques are of heuristic nature and their performance is benchmarked over several popular corpora (e.g., Calgary corpus[1]), but recently we can observe more active research on the theoretical front. The first non-trivial upper bounds on the output size of any BWT-based compression algorithm were obtained by Manzini [Man01] in 1999, who proved that, roughly speaking, the compressed text size T is bounded by $5|T|{H_k}^*(T) + \Theta(1)$ for the BWT+MTF+RLE0 scheme, where RLE0 is run-length encoding of runs of zeros only. A similar bound derived in Manzini's work pertained the case of the compression scheme without the RLE step, but the constant 5 grew to 8.

Currently, the best upper bound [GM07] on the BWT-based compression ratio is approximately 2.7 times the kth-order modified empirical entropy of the text, where k is arbitrary. This result was achieved with a variation of the distance coding post-BWT transform [Bin00], and has no dependency on the original text size, i.e., can reasonably bound even very low entropy texts. If, however, the entropy of a given text is not so very low, then the known upper bound is tighter [KLV06].

5.4.2 Search mechanism (LF-mapping)

Searching in the sequence T^{bwt} resembles the inverse BWT transform. First, the interval of rows of $\mathcal{M}$ containing the occurrences of $P[m]$ is found, the next found interval corresponds to pattern suffixes $P[m-1 \dots m]$, and so on. The function $Occ(S, c, i)$ is needed, which returns the number of occurrences of symbol c in sequence $S[1 \dots i]$. (Sometimes we may omit the sequence parameter for Occ, to simplify notation.)

Alg. 32 presents the search algorithm [FM00]. It finds the interval of $\mathcal{A}$ containing the occurrences of the pattern P. In addition to function $Occ(S, c, i)$, the array $C[1 \dots \sigma]$ is needed, which stores in $C[c]$ the number of occurrences of characters $\{\$, 1, \dots, c-1\}$ in the text T. With this definition, $C[c] + 1$ is the position of the first occurrence of c in F (if any).

An efficient implementation of the Occ function, both in time and space, is the key to the success of FM-indexes. The problem of fast Occ queries over a text sequence was posed long before compressed text indexes started to appear (in order to simulate tree navigation in little space [Jac89]), but originally was considered only for a binary alphabet. Traditionally, telling the number of occurrences of 1s in a given bit sequence, up to a given

[1] http://corpus.canterbury.ac.nz/descriptions#calgary

Alg. 32 Count-Occs($T^{\mathrm{bwt}}, n, P, m$).

```
1  i ← m
2  sp ← 1; ep ← n
3  while ((sp ≤ ep) and (i ≥ 1)) do
4      c ← P[i − 1]
5      sp ← C[c] + Occ(T^bwt, c, sp − 1) + 1
6      ep ← C[c] + Occ(T^bwt, c, ep)
7      i ← i − 1
8  if (ep < sp) then return "not found" else return "found (ep − sp + 1) occs"
```

position, is called the *rank* operation, and the inverse question, about the position of ith bit 1 in a given bit sequence, is called the *select* operation [Jac89]. Both operations are vital components of FM-indexes. In particular, the generic function *Occ* is typically translated into the *rank* problem.

More formally, let us define:

Def. 5.4.1. *Let* $B[1 \ldots n]$ *be a binary sequence. Then,* $rank(B, 1, i) = |\{1 \leq j \leq i : B[j] = 1\}|$*, where* $i \in [1 \ldots n]$*. Analogously,* $rank(B, 0, i) = |\{1 \leq j \leq i : B[j] = 0\}|$*, where* $i \in [1 \ldots n]$*. Now,* $select(B, 1, i) = j$*, such that* $rank(B, 1, j) = i$*, and, if* $j > 1$*,* $rank(B, 1, j - 1) = i - 1$*, where* $i \in [1 \ldots n]$*. Analogously,* $select(B, 0, i) = j$*, such that* $rank(B, 0, j) = i$*, and, if* $j > 1$*,* $rank(B, 0, j - 1) = i - 1$*, where* $i \in [1 \ldots n]$*.*

Note that *rank* functions for symbol 0 and 1 are complementary, that is, $rank(B, 0, i) = i - rank(B, 1, i)$ and vice versa, but this does not hold in the case of *select*. Where it does not lead to confusion, we are going to use a shorter notation, $rank(B, i)$ and $select(B, i)$, to denote the number of 1s in the considered sequence. Generalization of the above definition to alphabets larger than binary is straightforward. In some FM-index variants, the *select* query is not used.

Apart from the old question how to make those operations both fast and using little space, in compressed indexes we are faced with a newer problem: how to deal with alphabets larger than binary. We address those issues later. So far we only state that both *rank* and *select* can be handled in $O(1)$ time for binary sequences, using $o(n)$ bits apart from the sequence B itself.

5.4.3 Locating occurrences and displaying the text

In many practical scenarios one would like to know the positions of found pattern occurrences in the text, and often also its context, i.e., several char-

acters in front and just after each found match. Those particular queries are called *locate* (or *report*) and *display*.

Now we present the *locate* technique developed by Ferragina and Manzini [FM00]. In terms of the FM-index, the problem boils down to telling the position $pos(P)$ in T of each suffix which is prefixed with P in the matrix $\mathcal{M}$. This technique can easily be adapted to other indexes from the FM family (e.g., FM-Huffman, described later).

To this end, we sample T at regular intervals of size ℓ. The sampling parameter ℓ controls the sampling density and poses a natural space/time tradeoff. The sampled suffixes will be *logically* marked. A simple solution could be to assign a single bit to each suffix of T, and use 1s to denote the sampled bits, but this needs (at least) n bits overhead which might be a dominating term in the Ferragina–Manzini algorithm. To overcome this problem, they use a 2-level scheme (a careful reader will easily notice a similarity to the classic *rank* solution presented later in this chapter), partitioning the rows of $\mathcal{M}$ into buckets of size $\Theta(\log^2 n)$, and using one packet B-tree structure per bucket to store only the marked rows within the bucket, using their distance from the beginning of the bucket as the key. This allows to spend $O(\log^2 n)$ steps, each performed in constant time on a RAM machine. Overall, the *locate* operation (after the counting phase) for all pattern occurrences takes $O(occ \log^2 n)$ time in their scheme, with space penalty only $O(n/\log n)$ bits (which is absorbed in space complexity by other components of their index).

Yet in the same work Ferragina and Manzini refined this solution to obtain $O(m + occ \log^{1+\varepsilon} n)$ time, with space of $O(H_k(T) + \log\log n/\log^{\varepsilon} n)$ bits, for any $k \geq 0$. The main weakness of their algorithm is however the assumption of a constant-size alphabet. The complexities of their solution, concerning both space and time, have a sharp dependence on the alphabet size; discussing possible ways to mitigate this dependence will be the subject of Sect. 5.7 and 5.8. We note that with using a superalphabet Ferragina and Manzini managed to reduce the $1+\varepsilon$ exponent to ε only, but from a practical point this variant gets even more intractable.

Now, let us give the space and time complexities of the Ferragina and Manzini index, not ignoring the alphabet size. The FM-index needs up to $5H_k n + O\left((\sigma \log\sigma + \log\log n)\frac{n}{\log n} + n^{\gamma}\sigma^{\sigma+1}\right)$ bits of space, where $0 < \gamma < 1$. The time to search for a pattern and obtain the number of its occurrences in the text is $O(m)$. The text position of each occurrence can be found in $O(\sigma \ \log^{1+\varepsilon} n)$ time, for some $\varepsilon > 0$ that appears in the sublinear terms of the space complexity. Finally, the time to display a text substring of length

L is $O(\sigma\ (L+\log^{1+\varepsilon} n))$. The last operation is important not only to show a text context around each occurrence, but also because a self-index replaces the text and hence it must provide the functionality of retrieving any desired text substring.

5.5 *Rank* and *select* in theory

In Sect. 5.4.2 we mentioned that efficient implementation of the *Occ* function is crucial for both speed and space occupancy of any FM-index, and that calculating *Occ* boils down to answering *rank* queries. Applications of the related query, *select*, will be discussed later. We assume here that the alphabet for *rank* and *select* is binary.

Constant-time solutions for both *rank* and *select* can be trivially obtained via storing all answers explicitly. This requires $O(n \log n)$ bits though, hence is prohibitive. A natural question arises, how much space is needed, in addition to the bit-vector B, to resolve those queries in $O(1)$ time. The answer is that both structures need $o(n)$ bits.

5.5.1 Constant-time *rank*

The solution for *rank* having both required properties is simple [Jac89, Mun96, Cla96]. The sequence B is divided into blocks of size $b = \lfloor \log_2 n/2 \rfloor$. Consecutive blocks are grouped into superblocks of size $s = b\lfloor \log_2 n \rfloor$.

For each superblock j, $0 \le j \le \lfloor n/s \rfloor$, we store a number $R_s[j] = rank(B, js)$. Array R_s needs overall $n/b = O(n/\log n)$ bits as each $R_s[j]$ needs $\log_2 n$ bits and there are $n/s = n/(b \log_2 n)$ entries.

For each block k of superblock $j = k$ div $\lfloor \log_2 n \rfloor$, $0 \le k \le \lfloor n/b \rfloor$, we store a number $R_b[k] = rank(B, kb) - rank(B, js)$. Array R_b needs $(n/b) \log_2 s = O(n \log\log n/\log n)$ bits since there are n/b blocks overall and each $R_b[k]$ value needs $\log_2 s = \Theta(\log\log n)$ bits.

Finally, for every bit stream S of length b and for every position i inside S, we precompute $R_p[S, i] = rank(S, i)$. This requires $O(2^b \times b \times \log b) = O(\sqrt{n} \log n \log\log n)$ bits.

Those structures need $O(n/\log n + n \log\log n/\log n + \sqrt{n} \log n \log\log n) = o(n)$ bits. They permit computing *rank* in constant time as follows:

$$rank(B, i) = R_s[\lfloor i/s \rfloor] + R_b[\lfloor i/b \rfloor] + R_p[B[\lfloor i/b \rfloor \times b+1 \ldots \lfloor i/b \rfloor \times b+b], i \bmod b]$$

This structure can be implemented with little effort and works fast. Yet, consider its extra space, for example, for $n = 2^{30}$ bits. R_s poses a space

overhead of 6.67%, R_b of 60%, and R_p of 0.18%. Overall, the $o(n)$ additional space is 66.85% of n, which is hardly negligible.

5.5.2 Constant-time *select*

The constant time solution to $select(B, j)$ is significantly more complex than for $rank(B, i)$. Jacobson's attempt [Jac89] reached $O(\log \log n)$ time, while the first $O(1)$-time algorithm was presented by Clark [Cla96]. His *select* structure requires $\frac{3n}{\lceil \log_2 \log_2 n \rceil} + O(\sqrt{n} \log n \log \log n)$ bits of extra space. The algorithm is rather complicated and we do not present it here.

Instead, a constant-time implementation devised by the author of the dissertation will be presented, which uses $O(n \log \log n / \sqrt{\log n})$ bits of space in addition to the vector B itself. Note that the space use is slightly lower than for Clark's structure (although not as good as the most succinct achievements, to be listed later), but the main advantage of our construction lies in its simplicity. After developing it, we noticed it resembles the Clark's algorithm simplification presented in [NM07, Sect. 6.1], still, our solution is more succinct.

The problem considered here is to report the position of jth bit 1 in bit-vector $B[0 \ldots n-1]$ for an arbitrary $j = 1 \ldots m$, where m is the total number of bits 1 in B.

First, we store explicit answers for every $j = ik$, $i = 0 \ldots \lceil m/k \rceil - 1$, where k is a parameter fixed later; now we only state that $k = O(\text{polylog}(n))$. In total, those answers occupy $m \log n / k$ bits, which is at most $n \log n / k$ bits. In the further considerations we assume that B is dense, i.e., m is close to n, and we avoid the term m in space formulas.

In this way, we obtained $O(n/k)$ superblocks, of varying sizes, but never smaller than k bits. We consider three kinds of superblocks, so they need to be labeled, using in practice 2 bits each. The cost of labeling is then $O(n/k)$ bits. If a superblock size is at most $bs_1 = O(\log^2 n / \log \log n)$ bits, where the constant is set suitably (to be explained later), we divide it into blocks of $c \log n$ bits, where again c is a small enough constant, then calculate the rank for each block boundary relative to the beginning of the superblock, and pack those ranks into a single machine word. Indeed, the amount of blocks considered in a superblock is at most $O(\log n / \log \log n)$, the superblock size is $O(\text{polylog}(n))$, hence the ranks with the superblock need only $O(\log \log n)$ bits each, so the total amount of bits is $O(\log n)$, which fits a machine word. Answering $select(B, j)$, if j falls in this kind of a superblock, is based on lookups. First we find the $(c \log n)$-bit block that contains the desired position in $O(1)$ time, and search the block again in

$O(1)$ time. Both operations use 2-dimensional lookup tables (LUTs), that is, they accept two arguments. Concerning the first LUT, one argument is a bit-vector being the concatenation of "relative" ranks of size $O(\log\log n)$ bits each, for all $O(\log n/\log\log n)$ blocks in a superblock, the other is the position of index j in the superblock, that is $j \bmod k$, which is stored on $\log k = O(\log\log n)$ bits. The constants mentioned above can be set in a way to make the pair of arguments use up to e.g. $\log n/2$ bits. The answers returned by this LUT need $O(\log\log n)$ bits. The arguments for the second LUT are all $O(\log n/\log\log n)$ block bits and again $j \bmod k$, and again suitable constants must be set. Overall, the LUTs can be bounded with e.g. $O(\sqrt{n}\log\log n)$ bits of space.

In the second case the superblock size is between bs_1 and $bs_2 = \log^3 n$. There are at most $n/(bs_1+1) = O(n\log\log n/\log^2 n)$ such superblocks, i.e., at most $O(nk\log\log n/\log^2 n)$ set bits in them. Pointing those set bits relatively to the superblock beginning gives an extra $O(\log bs_2) = O(\log\log n)$ factor, i.e., in total we need $O(nk(\log\log n)^2/\log^2 n)$ bits.

The third, final case is when the superblock size is above bs_2. In this case, the k answers are stored explicitly on $\log n$ bits each, but there are at most $n/\log^3 n$ such superblocks, hence up to $nk/\log^2 n$ bits are used.

The query time in all cases is clearly $O(1)$. Let us sum up the space terms, in the order of introducing them: $O(n\log n/k + n/k + \sqrt{n}\log\log n + n\log\log n\log n/\log^2 n + nk/\log^3 n + nk(\log\log n)^2/\log^2 n) = O(n\log n/k + n\log\log n\log n/\log^2 n + nk(\log\log n)^2/\log^2 n)$ bits. This is minimized for $k = O(\log^{1.5} n/\log\log n)$, which leads to overall space $O(n\log\log n/\sqrt{\log n} + n\log\log n/\log n + n\log\log n/\sqrt{\log n}) = O(n\log\log n/\sqrt{\log n})$ bits. This is slightly better in asymptotic terms than in Clark's solution, as promised.

To sum up, the presented $O(1)$-time solutions for *rank* and *select* need $O(n\log\log n/\log n)$ and $O(n\log\log n/\sqrt{\log n})$ bits, respectively. An interesting question is whether those complexities, although sublinear in n, can still be improved. The answer is negative for *rank*, but positive for *select*. Only recently it was found that the space lower bounds for both problems are same, $O(n\log\log n/\log n)$ bits in addition to the sequence itself, and were proven for *rank* by Miltersen [Mil05] and for *select* by Golynski [Gol06]. Note that now the upper and lower bounds for both problems are same. We point out that those results hold in the so-called systematic model, in which the vector B is represented "as is". In the unrestricted setting, the bounds are better and now the upper and lower bounds are essentially equal [Păt08, Păt09].

5.5.3 *Rank* and *select* for compressed sequences

So far, we assumed that the vector B is uncompressed, hence in total the structure we work with requires $n + o(n)$ bits. Interestingly, further refinements [Pag99, RRR02] achieved constant time on the same queries by using only $nH_0(B)+o(n)$ bits overall, where $H_0(B)$ is the zero-order entropy of B. Note that in this case, even the *access* query, i.e., determining $B[i]$ given i, is not trivial to handle. Finally, Sadakane and Grossi [SG06] gave a solution in which the main space term is only $nH_k(B)$, i.e., the kth-order entropy of B, and the constant time complexities preserved. This is an ultimate result, in a sense, but for rare sequences the lower order space terms may dominate, hence the problem is not yet closed; for other results, see [MN07b] and references therein.

If the alphabet is larger than binary, then all operations can be done in $O(\log\log\sigma)$ time using $n(\log_2\sigma + o(\log\sigma))$ bits of space [GMR06].

5.6 *Rank* and *select* in practice

In this section, we examine several practical variants supporting the operations *rank* and *select*, and we experimentally compare them against the classic solutions. We also consider a novel operation, *selectNext*, being a special kind of *select*, which, as we show, can be executed in a much simpler and faster way than the general *select*. All those results, including extensive experimental tests, were presented in [GGMN05].

5.6.1 *Rank* queries via popcounting

The term *popcount* (population count) refers to counting how many bits are set in a bit array. We note that table R_p can be replaced by popcounting, as $R_p[S,i] = popcount(S \;\&\; 1^i)$. This permits removing the second argument of R_p, which makes the table smaller. In terms of time, we perform an extra *and* operation in exchange for either a multiplication or an indirection to handle the second argument. The change is clearly beneficial.

Popcounting can be implemented either with bit manipulation in a single computer word or with table lookup. A simple, yet efficient example of the first kind is the formula:

```
popc = { 0, 1, 1, 2, 1, 2, 2, 3, ... }
popcount = popc[x & 0xFF] + popc[(x >> 8) & 0xFF]
         + popc[(x >> 16) & 0xFF] + popc[x >> 24]
```

where *popc* is a precomputed popcount table indexed by bytes $(0, 1, \ldots, 255)$.

The width of the argument of the precomputed table was fixed at 8 bits and b at 32 bits, hence requiring 4 table accesses. In a more general setup, we can choose $b = \log(n)/k$ and the width of the table argument to be $\log(n)/(rk)$, for integer constants r and k. Thus the number of table accesses to compute *popcount* is r and the space overhead for table R_b is $k \log\log(n)/\log(n)$. We cannot choose too small r and k though, as the the size of table *popc* is $n^{\frac{1}{rk}} \log\log(n)$. Clearly, we need $rk > 1$, which yields a space/time tradeoff.

In practice, b should be a multiple of 8 because the solutions to *popcount* work at least by chunks of whole bytes. With the setting $s = b \log n$, and considering the range $2^{16} < n \leq 2^{32}$ to illustrate, the overall extra space (not counting R_p) is 112.5% with $b = 8$, 62.5% with $b = 16$, 45.83% with $b = 24$ and 34.38% with $b = 32$.

We have tried the reasonable (k, r) combinations for $b = 16$ and $b = 32$: (1) $b = 32$ and a 16 KB *popc* table needing 2 accesses for *popcount*, (2) $b = 16$ and a 16 KB *popc* table needing 1 access for *popcount*, (3) $b = 16$ and a 256-byte *popc* table needing 2 accesses for *popcount*, and (4) $b = 32$ and a 256-byte *popc* table needing 4 accesses for *popcount*. Other choices require too much space or too many table accesses. We have also excluded $b = 8$ because its space overhead is too high and $b = 24$ because it requires non-aligned memory accesses.

Figure 5.3 (left) shows execution times for $n = 2^{12}$ to $n = 2^{30}$ bits. For each size we randomly generate 200 arrays and average the times of 1 000 000 *rank* queries over each. We compare the four alternatives above as well as the mentioned method that does not use tables. As it can be seen, the combination (4), that is, $b = 32$ making 4 accesses to a table of 256 entries, is the fastest in most cases, and when it is not, the difference is negligible.

On the other hand, it is preferable to read word-aligned numbers than numbers that occupy other number of bits such as $\log n$, which can cross word boundaries and force reading two words from memory. In particular, we have considered the alternative $s = 2^8$, which permits storing R_b elements as bytes. The space overhead of R_b is thus only 25% with $b = 32$ (and 50% for $b = 16$), and accesses to R_b are byte-aligned. The price for such a small s is that R_s gets larger. For example, for $n = 2^{20}$ it is 7.81%, but the sum is still inferior to the 34.38% obtained with the basic scheme $s = b \log n$. Actually, for little more space, we could store R_s values as full 32-bit integers (or 16-bit if $\log n \leq 16$). The overhead factor due to R_s becomes now 32/256 (or 16/256), which is at most 12.5%. Overall, the space overhead is 37.5%,

close to the non-aligned version. Figure 5.3 (left) shows that this alternative is the fastest, and it will be our choice for popcount-based methods.

Note that up to $n = 2^{20}$ bits, the original bit array together with the additional structures need at most 176 KB with $b = 32$, and 208 KB with $b = 16$. Thus the 256 KB cache of the test machine accommodates the whole structure. However, for $n = 2^{22}$, we need 512 KB just for the bit array. Thus the cache hit ratio decreases as n grows, which explains the increase in query times that should be constant. For a systematic study of the cache effect the reader is referred to [GGMN05].

5.6.2 *Rank* queries using a single level plus sequential scan

At this point we still follow the classical scheme in the sense that we have two levels of blocks, R_s and R_b. This forces us to make two memory accesses in addition to accessing the bit array block. We consider now the alternative of using the same space to have a single level of blocks, R_s, with one entry each $s = 32k$ bits, and using a single 32-bit integer to store the ranks. To answer a $rank(B, i)$ query, we would first find the latest R_s entry that precedes i, and then sequentially scan the array, popcounting in chunks of $w = 32$ bits, until reaching the desired position, as follows:

$$\begin{aligned} rank(B,i) &= R_s[\lfloor i/s \rfloor] + \\ &\quad \sum_{j=\lfloor(\lfloor i/s \rfloor s)/w\rfloor+1}^{\lfloor i/w \rfloor w-1} popcount(B[jw+1 \ldots jw+w]) + \\ &\quad popcount(B[\lfloor i/w \rfloor w+1 \ldots \lfloor i/w \rfloor w+w] \ \& \ 1^{i \bmod w}) \end{aligned}$$

Note that the sequential scan accesses at most k memory words, and the space overhead is $1/k$. Thus we have a space/time tradeoff. For example, with $k = 3$ we have approximately the same space overhead as in our preferred two-level version.

Figure 5.3 (right) compares the execution time of different tradeoffs against the best previous alternatives. For the alternative of using only one level of blocks, we have considered extra spaces of 5%, 10%, 25%, 33% (close to the space of our best two-level alternative), and 50%.

One can see that the direct implementation of the theoretical solution is far from competitive: it wastes the most space and is among the slowest. Our two-level popcount alternative is usually the fastest by far, showing that the use of two levels of blocks plus an access to the bit array is normally better than using the same space (and even more) for a single level of blocks.

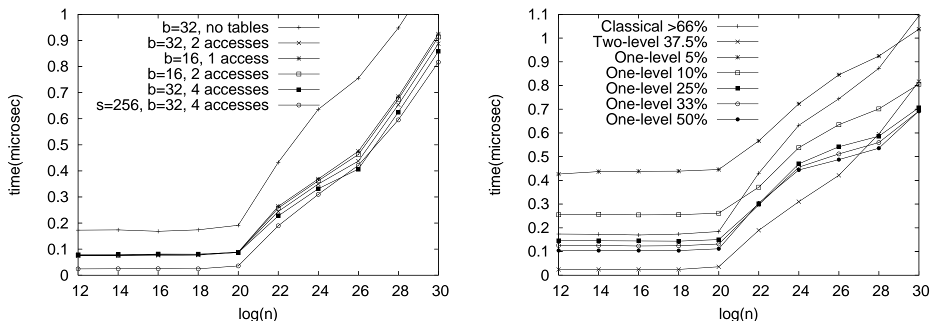

Figure 5.3: (Left) Comparison of different popcount methods to solve *rank*. (Right) Comparison of different mixed approaches to solve *rank*: classical scheme, popcounting with two levels of blocks, and popcounting with one level of blocks

Yet, note that the situation is reversed for large n. The reason is the locality of reference of the one-level versions: They perform one access to R_s and then a few accesses to the bit array (on average, 1 access with 50% overhead, 1.5 accesses with 33% overhead and 2 accesses with 25% overhead). Those last accesses are close to each other, thus from the second on they are surely cache hits. On the other hand, the two-level version performs three accesses (R_s, R_b, and the bit array) with no locality among them. When the cache hit ratio decreases significantly, those three nonlocal accesses become worse than the two nonlocal accesses (plus some local ones) of the one-level versions.

Thus, which is the best choice among one and two levels depends on the application. Two levels is usually better, but for large n one can use even less space and be faster. Yet, there is no fixed concept of what is "large", as other data structures may compete for the cache and thus the real limit can be lower than in presented experiments.

5.6.3 *Select* queries

A simple, yet $O(\log n)$ time, solution to $select(B, j)$, is to binary search in B the position i such that $rank(B, i) = j$ and $rank(B, i-1) = j-1$. Hence, the same structures used to compute $rank(B, i)$ in constant time can be used to compute $select(B, j)$ in $O(\log n)$ time.

More efficient than using $rank(B, i)$ as a black box is to take advantage of its layered structure, so as to first binary search for the proper superblock using R_s, then binary search that superblock for the proper block using R_b, and finally binary search for the position inside the correct block.

For the search in the superblock of s bits, there are three alternatives:

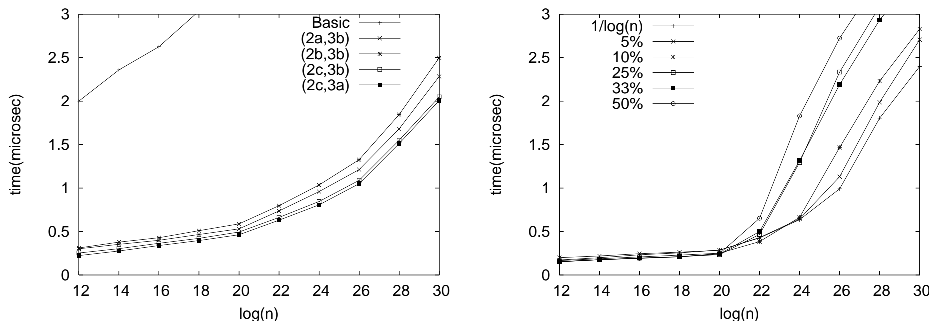

Figure 5.4: (Left) Comparison of different methods to solve *select* by binary search. (Right) comparison of different space overheads for *select* based on binary search

($2a$) binary search using R_b, ($2b$) sequential search using R_b (since there are only a few blocks inside a superblock), and ($2c$) sequential search using popcount. The latter alternative consists of simply counting the number of bits set inside the superblock, and has the advantage of not needing array R_b at all. For the search in the last block of b bits, binary search makes little sense because *popcount* proceeds anyway byte-wise, so we have considered two alternatives: ($3a$) bytewise search using popcount plus bit-wise search in the final byte, and ($3b$) sequential bit-wise search in the b bits.

In the case of *select*, the density of the bit array may be significant. We have generated bit arrays of densities (fraction of bits set) from 0.001 to 1. For each density we randomly generated 50 different arrays of each size. For each array, the averages for the times of 400 000 *select* queries are given.

The results for the binary search version are almost independent of the density of bits set in B, hence Fig. 5.4 (left) shows only the case of density 0.4. We first compare alternatives $(2a, 3b)$, $(2b, 3b)$ and $(2c, 3b)$. Then, as $(2c, 3b)$ turns out to be the fastest, we consider also $(2c, 3a)$, which is consistently the best. Other plots demonstrate that the basic binary search (not level-wise) is much slower than any other solution. In this experiment we have used $b = 32$ and $s = b \log n$.

Note that the best alternative only requires space for R_s (that is, $1/32$), as all the rest is solved with sequential scanning. Once it is clear that using a single level is preferable for *select*, we consider speeding up the accesses to R_s by using 32-bit integers (or 16-bits when $\log n \leq 16$). Moreover, we can choose any sampling step of the form $s = k \times b$ so that the sequential scan accesses at most k blocks and we pay $1/k$ overhead.

Figure 5.4 (right) compares different space overheads, from 5% to 50%.

We also include the case of $1/\log n$ overhead, which is the space needed by the version where R_s stores $\log n$ bit integers instead of 32 bits. It can be seen that these word-aligned alternatives are faster than those using exactly $\log n$ bits for R_s. Moreover, there is a clear cache effect as n grows. For small n, higher space overheads yield better times as expected, albeit the difference is not large because the binary search on R_s is a significant factor that smoothes the differences in the sequential search. For larger n, the price of the cache misses during the binary search in R_s is the dominant factor, thus lower overheads take much less time because their R_s arrays are smaller and their cache hit ratios are higher. The sequential search, on the other hand, is not so important because only the first access may be non-local, all the following ones are surely cache hits. Actually, the variant of $1/\log n$ overhead is finally the fastest because for $n = 30$ it is equivalent to 3.33% overhead.

The best alternative is the one that balances the number of cache misses during binary search on R_s with those occurring in the sequential search on the bit array. It is interesting, however, that a good solution for *select* requires little space.

The original Clark's $O(1)$-time solution (cf. Sect. 5.5) was also implemented and Fig. 5.5 shows its execution times (different lines for different densities of the bit arrays). We note that, although the *select* time is $O(1)$, there are significant differences for different array sizes or different densities of the arrays (albeit of course those differences are bounded by a constant). Note, for example, that for density 0.001 the search is extremely fast, as in this case the structure boils down to storing all answers explicitly.

To understand all the plots, knowledge of Clark's solution is needed, hence we omit a detailed discussion. We only point out the strange behavior at density 0.01 (times grow and then decrease with n), which is not erroneous. This has to do with the final step of Clark's procedure, sequential scanning, which takes relatively long (but constant of course) time. For larger n, at a fixed density, the scanning is performed over a smaller area, and this basically explains the strange phenomenon.

The plot also shows the time for the binary search versions using 5% and 50% space overhead. For very low densities (up to 0.005 and sometimes 0.01), Clark's implementation is superior. However, we note that for such low densities, the *select* problem is trivially solved by explicitly storing all the positions of all the bits set (that is, precomputing all answers), at a space overhead that is only 32% for density 0.01. Hence this case is not interesting. For higher densities, our binary search versions are superior up to $n = 2^{22}$ or 2^{26} bits, depending on the space overhead we chose (and

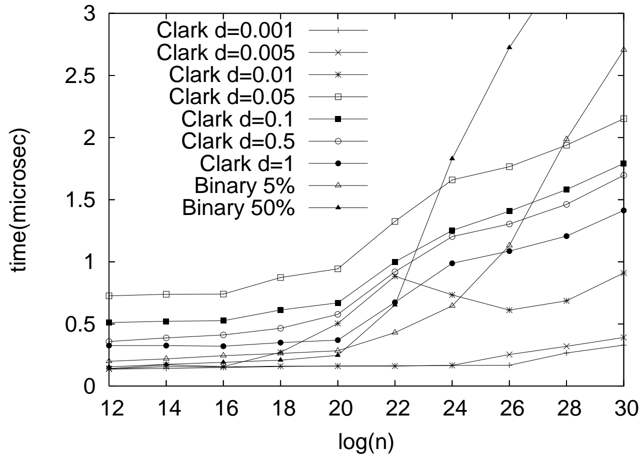

Figure 5.5: Comparison of Clark's *select* on different densities and two binary search based implementations using different space overheads

hence on how fast we want to be for small n). After some point, however, the $O(\log n)$ nature of the binary search solution shows up, and the constant-time solution of Clark finally takes over. It should be emphasized that Clark's implementation imposes a space overhead of 60% at least.

5.6.4 $SelectNext$ queries

There are applications which require a restricted version of *select*, namely $select(B, i+1)$ given $j = select(B, i) + 1$. This operation is called here $selectNext(B, j)$, and it gives the position of the first bit set in $B[j \dots n]$, or $n+1$ if no such bit set exists. As we will see in Sect. 5.8, this operation may be useful (in theory, at least) in locating found patterns in the FM-Huffman index. Obviously, it is possible to make use of the formula $selectNext(B, j) = select(B, 1 + rank(B, j-1))$. Instead we propose [GMN04a] a much simpler and more efficient solution, inspired by the *rank* structure.

We divide B like in the standard *rank* implementation, into blocks and superblocks of sizes b and s, respectively. For each superblock j, $1 \leq j \leq \lfloor n/s \rfloor$ we store a number $N_s[j] = selectNext(B, j \times s + 1)$. Array N_s needs overall $O(n/\log n)$ bits since each $N_s[j]$ value needs $O(\log n)$ bits.

For each block k of superblock $j = k$ div $\lfloor \log n \rfloor$, $1 \leq k \leq \lfloor n/b \rfloor$, we store $N_b[k] = selectNext(B[j \times s + 1 \dots (j+1) \times s], k \times b - j \times s + 1)$. Array N_b needs $O(n \log\log n / \log n)$ bits since each $N_b[k]$ needs $O(\log\log n)$ bits, as it represents a position inside a superblock of length $O(\log^2 n)$.

Finally, for every bit stream S of length b and for every position i inside S, we precompute $N_p[S,i] = selectNext(S,i)$. This requires $O(2^b \times b \times \log b) = O(\sqrt{n} \log n \log\log n)$ bits. Note that $N_p[S,i] = b+1$ if $S[i \ldots b]$ contains all zeros.

As before, the structures require $O(n \log\log n / \log n)$ bits, with exactly the same overhead as for *rank*. With them, *selectNext*(B,i) can be answered in $O(1)$ time, as follows.

(1) Compute $i_b = \lfloor i/b \rfloor b$ and then $pos = N_p[B[i_b+1 \ldots i_b+b], i-i_b+1]$. If $pos \leq b$, then there is a bit set in $B[i \ldots i_b+b-1]$ and we just return $i_b + pos$.
(2) Otherwise, $selectnext(B,i) = selectnext(B, i_b+b)$, so find the answer corresponding to the beginning of the next block. Compute $pos = N_b[(\lfloor i/b \rfloor + 1]$. If $pos \leq s$, then return $\lfloor (i_b+b)/s \rfloor s + pos$.
(3) Otherwise, there are all zeros in $B[i \ldots i_s+s-1]$ where $i_s = \lfloor i/s \rfloor s$, so $selectnext(B,i) = selectnext(B, i_s+s)$. Return $N_s[\lfloor i/s \rfloor + 1]$.

An even simpler solution, alternative to *selectNext*(B,i), is to sequentially scan all the bits in $B[i \ldots n]$ until finding a bit set. We search word by word rather than bit by bit. When a non-zero word is finally found, we use a precomputed table that, for every byte, tells the position of the first bit set. Then, in at most four access to the table, we find the position of the first bit set in the word where the sequential scanning stopped. We have considered other alternatives such as (1) using two accesses to a table of 2^{16} entries for the last step, and (2) going by chunks of 16 bits and performing one single access to a table of 2^{16} entries for the last step, but these were slightly worse.

Experiments show that the brute force solution (sequential scan) not only requires much less extra space but also is consistently faster, even with densities as low as 1000 (that is, where we have to scan 500 bits on average to find the answer). It was also observed that the constant-time solution worsens for lower densities because it is more probable to require more accesses (1, 2 or 3 table accesses). Finally, we point out that the solutions for *selectNext* are significantly faster than any solution for general *select*.

5.7 The wavelet tree

The main drawback of the FM-index is its severe dependence on the alphabet size, both in space and time (locate and display queries). Although Ferragina and Manzini removed this weakness in a practical implementation

[FM01], the resulting algorithm was a heuristic without interesting worst-case complexities.

One of the first FM-indexes removing the sharp dependence on σ was the *succinct suffix array* (SSA) from the technical report by Mäkinen and Navarro [MN04b, Chap. 3] published in April 2004; its SSA name was given in [MN05c]. A few months later Grabowski et al. [GMN04b] presented another variant, FM-Huffman, with similar complexities (with space worse by a constant though). Both algorithms belong to the simplest existing self-indexes, yet their space is bound in terms of order-0 entropy; for compressible data those indexes may occupy less space than original text. Both indexes handle the counting query in $O(m \log \sigma)$ time in the worst case and in $O(m(1 + H_0))$ time on average.

At the same conference (SPIRE 2004) Ferragina et al. [FMMN04] presented the *alphabet-friendly FM-index* (AF-FM), based on the idea of parititioning the BWT sequence, where the space was bounded in terms of order-k entropy, more precisely $nH_k(T) + o(n \log \sigma)$ bits. The worst-case time for counting remained $O(m(1 + \log \sigma))$.

Yet another index from this family was the *run-length FM-index* (RL-FM) [MN04b, MN05c], using more space than AF-FM, $O(nH_k(T) \log \sigma)$ bits, but decreasing the counting time to $O(m)$ if $\sigma = \text{polylog}(n)$.

Finally, Ferragina et al. [FMMN07] improved their AF-FM index, retaining its space, to reach $O(m(1 + \log \sigma / \log \log n))$ time for pattern counting, which is $O(m)$ for alphabets of polylogarithmic size. This achievement can be called the ultimate FM-index.

All those mentioned results, with the exception of the FM-Huffman by Grabowski et al. (to be presented in detail in the next section), are based on a simple ingenious data structure called the *wavelet tree* (WT) [GGV03]. This data structure allows to convert each $Occ(c, i)$ query into $\log \sigma$ *rank* operations for binary sequences. The idea is to decompose in the top level (the root) of WT the original alphabet $\{1, 2, \ldots, \sigma\}$ into two groups: the left half $\{1, 2, \ldots, \sigma/2\}$ and right half $\{\sigma/2 + 1, \sigma/2 + 2, \ldots, \sigma\}$; symbols from the first group will be denoted in the root with 0s and the symbols from the second groups with 1s. This idea is applied recursively down to the leaves; as the tree is balanced, any traversal from the root to a leaf requires visiting $\Theta(\log \sigma)$ levels.

Alg. 33, taken from [NM07, Sect. 9.4], shows in detail how the *Occ* function is computed on a (binary) wavelet tree. The full invocation is WT-$Occ(c, i, 1, \sigma, root)$, B^v denotes the bit vector at tree node v, while v_l and v_r are its left and right children.

Since *rank* over binary sequences can be easily implemented in $O(1)$ time,

Alg. 33 WT-Occ(c, i, 1, σ, *root*)

```
1   if σ1 = σ2 then return i
2   σm = ⌊(σ1 + σ2)/2⌋
3   if c ≤ σm
4       then return WT-Occ(c, rank0(B^v, i), σ1, σm, vl)
5       else return WT-Occ(c, rank1(B^v, i), σm + 1, σ2, vr)
```

it is clear that WT-$Occ(c, i, 1, \sigma, root)$ is accomplished in $O(\log \sigma)$ time.

Mäkinen and Navarro [MN04b] postulated to replace the original balanced WT with a Huffman-shaped one; that is, the more frequent symbols in the text have shorter paths from root to leaf and respectively rarer symbols have longer paths. This reduced the average *Occ* time to $O(H_0(T))$ (which is bounded by $\log \sigma$), but deteriorates the worst case to $O(\log n)$. In the same work they however presented a length-limited Huffman coding variant where the longest codeword has $O(\log \sigma)$ bits, hence the worst case for *Occ* remains logarithmic in the alphabet size. The compression loss of this Huffman variant compared to original one is negligible. The same length-limited Huffman idea can be used with FM-Huffman [GMN04b].

We note yet that the improved AF-FM index [FMMN07] is based on multi-ary wavelet trees, more sophisticated variant (generalization) which allows to retain constant time *rank* for a sequence over an alphabet of size $O(\log^{\varepsilon} n)$, and additionally keep this sequence compressed, in order-0 entropy related amount of space.

The last of major (at least from the practical point) discoveries in FM-indexes belongs again to Mäkinen and Navarro [MN07a]. They eventually noticed that using zero-order compressed *rank* (e.g., from [RRR02]) together with a single wavelet tree is enough to achieve H_k-related space of the index, thus eliminating the sequence partitioning mechanism of AF-FM.

5.8 FM-Huffman and its variants

In the previous section we showed a way to mitigate the large alphabet size dependence of the FM-index. It was based on wavelet trees. In the current section, an alternative approach is presented. In the basic variant, we Huffman-compress the text and then, as in the FM-index, apply the Burrows–Wheeler transform over it. The resulting structure can be regarded as an FM-index built over a binary sequence. This way we obtain only a mild (logarithmic in the worst case) dependence on the alphabet size.

The proposed index needs up to $n(2H_0 + 3 + \varepsilon)(1 + o(1))$ bits of space,

for any $0 < \varepsilon < 1$. It solves counting queries in $O(m(H_0 + 1))$ average time. The text position of each occurrence can be located in worst-case time $O\left(\frac{1}{\varepsilon}(H_0 + 1)\log n\right)$. Any text substring of length L can be displayed in $O((H_0 + 1)L)$ average time, in addition to the mentioned worst-case time required to locate a text position. In the worst case all the terms $(H_0 + 1)$ in the time complexities become $\log n$.

We also study several variants of the original index that reduce the term 2 in front of the space complexity, such as based on K-ary Huffman and Kautz–Zeckendorf (also called Fibonacci) coding [Kau65, Zec72]. Our experimental results show that our index, albeit not among the most succinct, is faster than the others in many practical cases, even if we let the other indexes use much more space. Furthermore, our index is attractive for its simplicity.

The idea was first presented in [GMN04b], and then developed and tested in an optimized implementation in [GMNS05, PGNS06, GNP$^+$06].

5.8.1 Basic idea

We start with presenting the original idea, based on binary Huffman. The notation used in this section will correspond to the one from Sect. 5.4. From now on assume T already contains the terminator \$ at the end[2]. To begin, this text T will be Huffman-compressed into a binary stream T' and the codeword beginnings marked in Th (the final index will not store T' nor Th). The idea is that, instead of searching T for P, we can Huffman-encode P into P' and search the binary text T' for P'. There is a problem though: we have to ensure that the occurrences of P' are codeword-aligned.

Def. 5.8.1. *Let $T'[1 \ldots n']$ be the binary stream resulting from Huffman-compressing T, where $n' < (H_0 + 1)\, n$ since (binary) Huffman poses a maximum representation overhead of 1 bit per symbol. Let $Th[1 \ldots n']$ be a second binary stream such that $Th[i] = 1$ iff i is the starting position of a Huffman codeword in T'. In the Huffman code, we ensure that the last bit assigned to the end marker "\$" is zero.*

The reason for the final condition will be clear later. Note that this can always be done, by making the node corresponding to "\$" a left child of its parent in the Huffman tree.

[2] Thus the term nH_0 will refer to this new text with terminator included. The difference with the term nH_0 corresponding to the text without the terminator is only $O(\log n)$, and will be absorbed by the $o(n)$ terms that will appear later in the space complexity.

Structure. We apply the Burrows–Wheeler transform over text T', so as to obtain $B = (T')^{\mathrm{bwt}}$. Yet, in order to have a binary alphabet, T' will not have its own special terminator character "\$" (note that the end marker of T is encoded in binary at the end of T', just as any other character of T). To formally define B we resort to the suffix array $\mathcal{A}'$ of T', yet the final index will not store $\mathcal{A}'$.

Def. 5.8.2. *Let $\mathcal{A}'[1\ldots n']$ be the suffix array for text T', that is, a permutation of $[1\ldots n']$ such that $T'[A'[i]\ldots n'] < T'[A'[i+1]\ldots n']$ in lexicographic order, for all $1 \leq i < n'$. In these lexicographic comparisons, if a string x is a prefix of y, we assume $x < y$.*

Our index will represent $\mathcal{A}'$ in succinct form, via array B and another array Bh used to track the codeword beginnings in $(T')^{\mathrm{bwt}}$.

Def. 5.8.3. *Let $B[1\ldots n']$ be a binary stream such that $B[i] = T'[\mathcal{A}'[i]-1]$ (except that $B[i] = T[n']$ if $\mathcal{A}'[i] = 1$). Let $Bh[1\ldots n']$ be another binary stream such that $Bh[i] = Th[\mathcal{A}'[i]]$. This tells whether position i in $\mathcal{A}'$ points to the beginning of a codeword.*

Searching. The mechanism of searching over B with be basically the same as in the FM-index. Again we need array C and function Occ, now applied to T' and B. As the alphabet of T' (and other used sequences) is binary, C and Occ can easily be computed in constant time using the function *rank*. The extra space is sublinear in the length of the respective sequence. Note that our C array has only two entries, which are easily precomputed. Similarly, Occ can be expressed in terms of *rank*:

$$\begin{array}{rcl|rcl} C[0] & = & 0 & Occ(B,0,i) & = & i - rank(B,i) \\ C[1] & = & n - rank(B,n') & Occ(B,1,i) & = & rank(B,i) \end{array}$$

Therefore, formulas $C[c] + Occ(T^{\mathrm{bwt}}, c, i)$ in the search algorithm of Alg. 32 are solved in FM-Huffman by using *rank* on B.

There is a small twist, however, because we are not putting a terminator to our binary sequence T' and hence no terminator appears in B. Let us call "#" ($\# < 0 < 1$) the terminator that should appear in T', so that it is not confused with the terminator "\$" of T. In the position $p_{\#}$ such that $\mathcal{A}'[p_{\#}] = 1$, we should have $B[p_{\#}] = \#$. Instead, we are setting $B[p_{\#}]$ to the last bit of T'. This is the last bit of the Huffman codeword assigned to the terminator "\$" of T, and it is zero according to Definition 5.8.1. Hence the correct B sequence would be of length $n' + 1$, starting with 0 (which

Alg. 34 Huff-FM_Count(B, Bh, n', P', m').

```
1   i ← m'
2   sp ← 1; ep ← n'
3   while (sp ≤ ep) and (i ≥ 1) do
4   if P'[i] = 0 then
5       sp ← (sp − 1) − rank(B, sp − 1) + [sp − 1 < p_#] + 1
6       ep ← ep − rank(B, ep) + [ep < p_#]
7   else sp ← n' − rank(B, n') + rank(B, sp − 1) + 1
8       ep ← n' − rank(B, n') + rank(B, ep)
9   i ← i − 1
10  if ep < sp return 0 else return rank(Bh, ep) − rank(Bh, sp − 1)
```

corresponds to $T'[n']$, the character preceding the occurrence of "#"), and it would have $B[p_\#] = \#$. To obtain the right mapping to our binary B, we must add 1 to $C[0] + Occ(B, 0, i)$ when $i < p_\#$. The computation of $C[1]+Occ(B, 1, i)$ remains unchanged. Overall, formula $C[c]+Occ(T^{\text{bwt}}, c, i)$ is computed as follows

$$C[c] + Occ(T^{\text{bwt}}, c, i) \quad = \quad \begin{cases} i - rank(B, i) + [i < p_\#], & \text{if } c = 0, \\ n - rank(B, n') + rank(B, i), & \text{if } c = 1, \end{cases} \tag{5.2}$$

where $p_\# = (\mathcal{A}')^{-1}[1]$.

Therefore, by preprocessing B to solve *rank* queries, we can search B exactly as in the FM-index. Our search pattern is not the original P, but its binary encoding $P'[1 \ldots m']$ using the Huffman code we applied to T.

The answer to that search, however, is different from that of the search of T for P. The reason is that the search of T' for P' returns the number of suffixes of T' that start with P'. Certainly these include the suffixes of T that start with P, but also other suffixes of T' that do not start a Huffman codeword, yet start with P'.

Array Bh now comes into play to filter out those spurious occurrences. In the range $[sp \ldots ep]$ found by the search of B' for P', every bit set in $Bh[sp \ldots ep]$ represents a true occurrence. Hence the true number of occurrences can be computed as $rank(Bh, ep) - rank(Bh, sp - 1)$. Alg. 34 shows the final search algorithm.

Analysis. The index stores B and Bh, each of $n' < (H_0 + 1)\, n$ bits. The extra space required by the *rank* structures is $o(n') = o((H_0 + 1)\, n)$. The only dependence on σ is that we must store the Huffman code, for which $\sigma \log n$ bits is sufficient (say, using a canonical Huffman tree). Thus our index requires at most $2n(H_0 + 1)(1 + o(1)) + \sigma \log n$ bits. The latter term is

$o(n)$ even for very large alphabets, $\sigma = o(n/\log n)$. Note that alternative indexes achieving kth-order compression [FMMN04, GGV03, GGV04, MN05c] require $\sigma = O(n^{1/k})$. The space of our index will grow slightly in the next sections due to additional requirements for locating and displaying queries.

Let us now consider the time for counting queries. If we assume that the characters in P have the same distribution of T (which holds in particular if P is randomly chosen from T, or generated by the same statistical source), then the length of P' is $m' < m(H_0 + 1)$. This is the number of steps to search B in Alg. 34, so the search complexity is $O(m(H_0 + 1))$. Since $H_0 \leq \log \sigma$, our time is better than the $O(m \log \sigma)$ complexity of several indexes [FMMN04, GGV03, GGV04][3].

We now analyze our worst-case search cost, which depends on the maximum height of a Huffman tree with total frequency n. Consider the longest root-to-leaf path in the Huffman tree. The leaf symbol has frequency at least 1. Let us traverse the path upwards and consider the (sum of) frequencies encountered in the other branch at each node. These numbers must be, at least, 1, 1, 2, 3, 5, ..., that is, the Fibonacci sequence $F(i)$. Hence, a Huffman tree with depth d needs that the text is of length at least $n \geq 1 + \sum_{i=1}^{d} F(i) = F(d+2)$ [WMB99, pp. 397]. Therefore, the maximum length of a codeword is $F^{-1}(n) - 2 = \log_\phi(n) - 2 + o(1)$, where $\phi = (1 + \sqrt{5})/2$.

Thus, the encoded pattern P' cannot be longer than $O(m \log n)$ and this is also the worst-case search cost. This matches the worst-case search cost of the original CSA, while our average case is better. It is actually possible to reduce our worst-case time to $O(m \log \sigma)$, without altering the average search time nor the space usage, by forcing the Huffman tree to become balanced after level $(1 + x) \log \sigma$, for some suitable constant $x > 0$. This length-limited Huffman variant, mentioned in Sect. 5.7, is analyzed in detail in [MN04b].

5.8.2 Locate and display

Locating occurrences and displaying the context around the matches, i.e., several characters just preceding and just following each occurrence, belong to basic functionalities of any indexes. Note also that since self-indexes replace the text, in general one needs to extract arbitrary text substrings from the index.

[3] In practice, those indexes can also achieve $O(m(H_0+1))$ average time using Huffman-shaped wavelet trees.

The mechanism implementing those functionalities, originally given by Ferragina and Manzini [FM00] and summarized in Sect. 5.4.3, can straightforwardly be adapted to FM-Huffman (and other indexes from the FM family).

First the time complexities for the locate operation in other indexes should be presented. Given the suffix array interval that contains the *occ* occurrences found, the FM-index locates each such position in $O(\sigma \log^{1+\varepsilon} n)$ time, for any $0 < \varepsilon < 1$ (which affects the sublinear space component). Note that we do not assume here a constant alphabet (as Ferragina and Manzini did). The CSA can locate each occurrence in $O(\log^{\varepsilon} n)$ time, where ε is paid in the space, nH_0/ε. Similarly, a text substring of length L can be displayed in time $O(\sigma(L + \log^{1+\varepsilon} n))$ by the FM-index and $O(L + \log^{\varepsilon} n)$ by the CSA.

In this section we show that our index – in spite of making use of the same mechanism – can do better than the FM-index, although not as well as the CSA. Using $(1+\varepsilon)\,n$ additional bits, we can locate each occurrence in time $O(\frac{1}{\varepsilon}(H_0 + 1)\log n)$ and display a text context in time $O(L \log \sigma + \log n)$ in addition to locating time. On average, if random text positions are involved, the overall complexity to display a text interval is $O((H_0 + 1)(L + \frac{1}{\varepsilon}\log n))$.

A first problem is how to extract, in $O(occ)$ time, the *occ* positions of the bits set in $Bh[sp \dots ep]$. This is easy using the *select* function. Actually we need a simpler version, $selectNext(Bh, j)$, which gives the first 1 in $Bh[j \dots n]$.

Let $r = rank(Bh, sp - 1)$. Then, the positions of the bits set in Bh are $select(Bh, r + 1)$, $select(Bh, r + 2)$, ..., $select(Bh, r + occ)$. We recall that $occ = rank(Bh, ep) - rank(Bh, sp - 1)$. This can be expressed using *selectNext*: The positions $pos_1 \dots pos_{occ}$ can be found as

$$\begin{aligned} pos_1 &= selectNext(Bh, sp), \\ pos_{i+1} &= selectNext(Bh, pos_i + 1). \end{aligned}$$

To complete the locating and displaying processes, we need additional structures.

Structure. We sample T' at approximately regular intervals, so that only codeword beginnings can be sampled. A sampling parameter $0 < \varepsilon < 1$ will control the density of the sampling and the corresponding space/time tradeoff.

Def. 5.8.4. *Given $0 < \varepsilon < 1$, let $\ell = \lceil \frac{2n'}{\varepsilon n} \log n \rceil$ be the* sampling step. *Our sampling of T' is a sequence $\mathcal{S}[1 \dots \lfloor \frac{\varepsilon n}{2 \log n} \rfloor]$, so that $\mathcal{S}[i]$ is the first*

position of the codeword that covers position $1 + \ell(i-1)$ *in* T'*, that is,* $\mathcal{S}[i] = select(Th, rank(Th, 1 + \ell(i-1)))$.

Our index will include three additional structures called ST, TS, and S. TS is an array storing the positions of $\mathcal{A}'$ that point to the sampled positions in T', in increasing text position order.

Def. 5.8.5. $TS[1 \ldots \lfloor \frac{\varepsilon n}{2 \log n} \rfloor]$ *is an array such that* $TS[i] = j$ *iff* $\mathcal{A}'[j] = \mathcal{S}[i]$.

Array ST is formed using the same positions of $\mathcal{A}'$, now sorted by position in $\mathcal{A}'$ and storing their position in T.

Def. 5.8.6. $ST[1 \ldots \lfloor \frac{\varepsilon n}{2 \log n} \rfloor]$ *is an array such that* $ST[i] = rank(Th, \mathcal{A}'[j])$*, where* j *is the* i*th position in* $\mathcal{A}'$ *that points to a position present in* $\mathcal{S}$.

Finally, $S[i]$ tells whether the ith entry of $\mathcal{A}'$ that points a codeword beginning, points to sampled a text position. S will be further processed for *rank* queries.

Def. 5.8.7. $S[1 \ldots n]$ *is a bit array such that* $S[i] = 1$ *iff* $\mathcal{A}'[select(Bh, i)]$ *is in* $\mathcal{S}$.

Locating. We have to determine the text position corresponding to an entry $\mathcal{A}'[i]$ for which $Bh[i] = 1$, that is, a valid occurrence. Use bit array $S[rank(Bh, i)]$ to determine whether $\mathcal{A}'[i]$ points or not to a codeword beginning in position in $ST[rank(S, rank(Bh, i))]$ and we are done. Otherwise, just as with the FM-index, determine position i' whose value is $\mathcal{A}'[i'] = \mathcal{A}'[i] - 1$. Repeat this process, which corresponds to moving backward bit by bit in T', until a new codeword beginning is found, that is, $Bh[i'] = 1$. Then check again whether this position is sampled, and so on until finding a sampled codeword beginning. If we finally obtain position pos after d repetitions, the answer is $pos + d$ as we have moved backward d positions in T.

It is left to specify how we determine i' from i. In the FM-index, this is done via the LF-mapping, $i' = C[T^{\text{bwt}}[i]] + Occ(T^{\text{bwt}}, T^{\text{bwt}}[i], i)$. In our index, the LF-mapping over $\mathcal{A}'$ is implemented using Eq. (5.2). Alg. 35 gives the pseudocode for locating the text position of the occurrence at $B[i]$. It is invoked for each $i = select(Bh, r + k)$, $1 \le k \le occ$, $r = rank(Bh, sp - 1)$.

Alg. 35 Huff-FM_Locate(i,B,Bh,S,ST).

```
1   d = 0
2   while S[rank(Bh, i)] = 0 do
3       do if B[i] = 0 then i = i − rank(B, i) + [i < p#]
4           else i = n′ − rank(B, n′) + rank(B, i)
5       while Bh[i] = 0
6       d = d + 1
7   return d + ST[rank(S, rank(Bh, i))]
```

Alg. 36 Huff-FM_Display(l,r,B,Bh,S,ST,TS).

```
1   j ← min{k, ST[rank(S, rank(Bh, TS[k]))] > r} /* binary search */
2   i ← TS[j]
3   p ← ST[rank(S, rank(Bh, i))]
4   L ← ⟨ ⟩
5   while p ≥ l do
6       do L = B[i] × L
7           if B[i] = 0 then i ← i − rank(B, i) + [i < p#]
8           else i ← n′ − rank(B, n′) + rank(B, i)
9       while Bh[i] = 0
10  p ← p − 1
11  Huffman-decode the first r − l + 1 characters from list L
```

Displaying. In order to display a text substring $T[l \dots r]$ of length $L = r - l + 1$, we start by binary searching TS for the smallest sampled text position larger than r. Let j be the index found in TS. Given value $i = TS[j]$, we know that $S[rank(Bh, i)] = 1$ as i is a sampled entry in $\mathcal{A}'$. The corresponding position in T is $ST[rank(S, rank(Bh, i))]$.

Once we find the first sampled text position that follows r, we know its corresponding position i in $\mathcal{A}'$. From there on, we move backwards in T' (via the LF-mapping over $\mathcal{A}'$), position by position, until reaching the first bit of the codeword for $T[r+1]$. Then, we obtain the L preceding characters of T, by further traversing T' backwards, now collecting all its bits until reaching the first bit of the codeword for $T[l]$. The bit stream collected is reversed and Huffman-decoded to obtain $T[l \dots r]$. Alg. 36 shows the pseudocode.

Analysis. Array TS requires $\frac{\varepsilon n}{2}(1+o(1))$ bits, since there are n'/ℓ entries and each entry needs $\log n' \leq \log n + O(\log\log n)$ bits. Array ST requires other $\frac{\varepsilon n}{2}$ bits, as its entries require $\log n$ bits. Finally, array S preprocessed for *rank* queries requires $n(1+o(1))$ bits. Overall, we spend $(1+\varepsilon)n(1+o(1))$ additional bits of space for locating and displaying queries. This raises the final space requirement to $n(2H_0 + 3 + \varepsilon)(1 + o(1)) + \sigma \log n$ bits. We point out that the additive term $3n$ is pessimistic (it assumes the worst-case

redundancy of Huffman coding) and for most real distributions it turns into not much more than $1n$.

Let us now consider the time for locating. This corresponds to the maximum distance between two consecutive samples in T', as we traverse it backwards until finding a sampled position. In Sect. 5.8.1 we mentioned that no Huffman codeword can be longer than $\log_\phi n - 2 + o(1)$ bits. Therefore, the distance between two consecutive samples in T', after the adjustment to codeword beginnings, cannot exceed

$$\ell + \log_\phi n - 2 + o(1) \leq \frac{2}{\varepsilon}(H_0+1)\log n + \log_\phi n - 1 + o(1) = O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right),$$

which is therefore the worst-case locating complexity.

For the displaying time, each of the L characters obtained costs us $O(H_0+1)$ on average because we obtain the codeword bits one by one. In the worst case they cost us $O(\log n)$. Note that we might have to traverse some additional characters from the next sampled position until reaching the text area of interest. Finally, we must consider the $O(\log n)$ time for the binary search of TS. Summing up, the time complexity is $O((H_0+1)(L+\frac{1}{\varepsilon}\log n))$ on average and $O(L\log n + (H_0+1)\frac{1}{\varepsilon}\log n)$ in the worst case.

Theorem 5.8.1. *Given a text $T[1\ldots n]$ over an alphabet σ and with zero-order entropy H_0, the FM-Huffman index requires up to $n(2H_0+3+\varepsilon)(1+o(1)) + \sigma\log n$ bits of space, for any constant $0 < \varepsilon < 1$ fixed at construction time. It can count the occurrences of $P[1\ldots m]$ in T in average time $O(m(H_0+1))$ and worst-case time $O(m\log n)$. Each such occurrence can be located in worst-case time $O(\frac{1}{\varepsilon}(H_0+1)\log n)$. Any text substring of length L can be displayed in time $O((H_0+1)(L+\frac{1}{\varepsilon}\log n))$ on average and $O((L+(H_0+1)\frac{1}{\varepsilon})\log n)$ in the worst case.*

5.8.3 K-ary Huffman

While storing B seems necessary as we are using zero-order compression of T, doubling the space requirement to store Bh seems a waste of space. In this section we explore a way to reduce the size of Bh. Instead of using Huffman over a binary coding alphabet, we can use a coding alphabet of $k > 2$ symbols, so that each symbol needs $\lceil \log k \rceil$ bits. Varying the value of k yields interesting time/space tradeoffs. We use only powers of 2 for k values, so that each symbol can be represented without wasting space.

The space usage varies in different aspects. The size of B increases since Huffman's compression ratio degrades as k grows. B has length $n' < (H_0^{(k)} +$

$1)\,n$ symbols, where $H_0^{(k)}$ is the zero-order entropy of the text computed using base k logarithm, that is, $H_0^{(k)} = H_0/\log_2 k$. Therefore, the size of B is bounded by $n' \log k = (H_0 + \log k)n$ bits. The size of Bh, on the other hand, is reduced since it needs one bit per symbol, that is n' bits.

The total space used by B and Bh structures is then $n'(1 + \log k) < n(H_0^{(k)} + 1)(1 + \log k)$, which is not larger than the space requirement of the binary version, $2n(H_0 + 1)$, for $1 \le \log k \le H_0$. In particular, if we choose $\log k = \alpha H_0$, then the space is upper bounded by $n((1+\alpha)H_0 + 1 + 1/\alpha)$, which is optimized at $\alpha = 1/\sqrt{H_0}$ (that is, $\log k = \sqrt{H_0}$). Using this optimal α value, the overall space required by B and Bh is $n(\sqrt{H_0} + 1)^2 < n(H_0 + 1)(1 + 2/\sqrt{H_0})$. The original overhead factor of 2 over pure Huffman compression has been reduced to $1 + O(1/\sqrt{H_0})$.

The space for the *rank* structures changes as well. The *rank* structure for Bh is computed in the same way of the binary version, and therefore its size is reduced to $o(H_0^{(k)} n)$ bits. To solve $Occ(B, c, i)$ queries, we must build the sublinear-size *rank* structures over σ virtual binary sequences $B_c[1 \ldots n']$, so that $B_c[i] = 1$ iff $B[i] = c$. Therefore, $Occ(B, c, i) = rank(B_c, i)$ can be computed in constant time. The size of those *rank* structures adds up $o(kH_0^{(k)} n)$ bits. (The solution for *rank* requires accessing the bit vectors B_c, but one can use B itself instead.)

If we use the optimum k derived above, the space for the *rank* structures is $o(n2^{\sqrt{H_0}}/\sqrt{H_0})$ extra space, which turns out to be still $o(n)$ (more precisely, $O(n/\log\log n)$) for $H_0 \le (\log\log n)^2$. This value is reasonably large in practice.

Regarding the time complexities, the pattern has average length less than $m(H_0^{(k)}+1)$ symbols. This is the counting complexity, which is reduced as we increase k. Using the value $k = 2^{\sqrt{H_0}}$ that optimizes the space complexity, the counting time is $O(m\sqrt{H_0})$. On the other hand, the counting time can be made $O(m)$ by using a constant α. For locating queries and displaying text, we need the same additional structures TS, ST and S as for the binary version. The k-ary version can locate the position of an occurrence in $O\left(\frac{1}{\epsilon}(H_0^{(k)} + 1)\log n\right)$ time, which is the maximum distance between two sampled positions. Similarly, the time used to display a substring of length L becomes $O((H_0^{(k)} + 1)(L + \frac{1}{\epsilon}\log n))$ on average and $O(L\log n + (H_0^{(k)} + 1)\frac{1}{\epsilon}\log n)$ in the worst case. Again, with the optimum k, $H_0^{(k)}$ is $\sqrt{H_0}$, and it can be made $O(1)$ by using a constant α.

5.8.4 Kautz–Zeckendorf coding

The previous section aimed at reducing the size of Bh in exchange for increasing the size of other structures. In this section we attempt to get rid of the Bh array completely, by replacing Huffman coding with another for which the bit stream itself enables synchronization at codeword boundaries. Our solution is based on a representation of integers first advocated by Kautz [Kau65] for its synchronization properties, that presents each number in a unique form as a sum of Fibonacci numbers. This technique is better known from a work by Zeckendorf [Zec72], therefore we will call it *Kautz–Zeckendorf (KZ) coding.* (The name of Fibonacci coding is also used in the literature.)

Consider the (slightly displaced) Fibonacci sequence $1, 2, 3, 5, 8, 13, \ldots$, that is, $f_1 = 1$, $f_2 = 2$, and $f_{i+2} = f_{i+1} + f_i$. It is easy to prove by induction that any integer N can be uniquely decomposed into a sum of Fibonacci numbers, where each summand is taken at most once and no two consecutive numbers are used in the decomposition. (If two consecutive numbers f_i and f_{i+1} appear in the decomposition, we can use f_{i+2} instead.) Thus we can represent N as a bit vector, whose ith bit is set iff the ith Fibonacci number is used to represent N. Obviously, no two consecutive bits can be set in this representation. This can be generalized to k consecutive ones [Kau65]. The recurrence is now $f_i = i$ for $i \leq k$ and $f_{i+k} = f_{i+k-1} + f_{i+k-2} + \ldots + f_{i+1} + f_i$. In this representation we do not permit a sequence of k consecutive numbers in the decomposition, and thus no stream of k 1s appears in the bit vector.

We use this encoding as follows. The source symbols are sorted by frequency and then the KZ codeword of number N is assigned to Nth most frequent symbol for all N. In addition, all the encodings are prepended with a sequence of k 1s followed by one 0.

If, during the LF-mapping, we read a 0 and then k successive 1s from T', we know that we are at a codeword beginning. Thus, Bh is no longer needed. This is expected to outweigh the fact that the encoding is not optimal as Huffman. An important side-effect is also that there is no need for *select* (or *selectNext*) to find the successive matches: they all are in a contiguous range in $\mathcal{A}'$. All the rest of the operatory remains unchanged.

5.8.5 Other space-time tradeoffs

There are other alternatives to binary Huffman coding, apart from k-ary Huffman. Most of them are intended to diminish the Bh array or even eliminate it completely. We present several of these ideas in this section. None of them, however, has been implemented so far. We also want to

stress that most of the analyses presented across this section are only rough and intended to get an idea of which techniques could be interesting for future development.

Replacing** Bh **with the Raman et al. structure. Raman et al. [RRR02] showed how to represent a bit vector Bh of size n' in only $n'H_0(Bh) + O(n' \log\log n' / \log n')$ bits of space, with *rank* and *select* queries handled in $O(1)$ time. Thus we can use this structure for replacing Bh and its *rank* helper structure. (Note that using this representation for B is unlikely to bring any gain, as the amounts of 0s and 1s in a Huffman stream are almost equal, which implies $H_0(B)$ being almost 1.) Let us first disregard the sublinear part of the space complexity. The entropy-related component of the size is $n'H_0(Bh) = n \log_2 \frac{n'}{n} + (n'-n) \log_2 \frac{n'}{n'-n}$. The second component is easily upper bounded by $(n'-n)\log_2(1+\frac{n}{n'-n}) \leq n/\ln(2)$. Now, n'/n is the average Huffman codeword length, which is upper bounded by H_0+1 (being H_0 the zero-order entropy of the text T). Therefore, the space to represent Bh using this alternative structure is at most $n(1/\ln(2) + \log_2(H_0+1))$. This size (albeit not our pessimistic upper bound) is never larger than the $n(H_0+1)$ bits of our plain representation, and it can be significantly less if H_0 is far away from 1.

For example, for the English text we use in Sect. 5.9, we have $H_0 = 4.8$, and this is also very close to the average Huffman length. In this case, Raman et al.'s structure requires less than 66% of the size of the original Bh, which on the overall yields 17% less space than our structure on binary Huffman, and even 8% less than the version based on 4-ary Huffman. The time complexities are not affected.

Those promising space estimations are based on the assumption that the sublinear component is negligible. However, a detailed analysis of the sublinear-space part of the structure of Raman et al. reveals that it would pose a significant space overhead compared to our simpler solution. For example, even for large $n' = 2^{30}$, the overhead of the sublinear part would still be around 100%, while ours is 37.5% (we use the fast *rank* implementation from [GGMN05], presented in Sect. 5.6.1). Overall, for $n' \leq 2^{30}$, the solution requires at least 20% more space than our simpler solution, even with $k = 2$. In addition, we note that implementing the structure of Raman et al. is more complex than succinct structures considered so far for the FM-Huffman index. In even more theoretical setting, the Raman et al. structure can be replaced with a more recent one from Sadakane and Grossi; see Sect. 5.5.3 and references therein.

Cumulative** rank **for k-ary variants. A careful reader may notice some asymmetry in the *rank* structure details for the binary and the k-ary implementations. The difference is that for the binary variant we needed a single *rank* structure, i.e., storing the counts of the 1s only. Calculating the function *Occ* also for a given symbol 0 was straightforward. In the k-ary variant ($k > 2$), we have exactly k *rank* structures. In this subsection we show how to implement the *rank* operations using only $k - 1$ such structures for k alphabet symbols. Unfortunately, this proposal does not seem practical.

The idea is to store in the jth *rank* structure ($j = 0 \ldots k - 2$) not the counts of symbols j, but rather the counts of all the symbols $0 \ldots j$. Let us denote $crank_j$ this modified structure storing cumulative $rank_j$ values. The original $rank_j$ function values are obtained now as follows: $rank_j(B, i) = crank_j(B, i) - crank_{j-1}(B, i)$, for $j = 1 \ldots k-2$, $rank_0(B, i) = crank_0(B, i)$, and $rank_{k-1}(B, i) = i - crank_{k-2}(B, i)$.

Note that in $k - 2$ out of k cases obtaining the *rank* value requires two accesses to the *crank* structure, which contrasts with the original *rank* use, where only one access is required. This also suggests that the only case where this idea may be considered as potentially practical is the case of the smallest k, i.e., $k = 4$. In space, we clearly use 75% of the original *rank* which, according to Table 5.1 from Sect. 5.9, decreases the overall space factor from $1.52n$ to $1.40n$ on English text. Note that the *rank* structure for the *Bh* array is unchanged. In terms of search time, however, this idea is expected to make searches about 1.5 times slower. This is because a pattern search is dominated by the *rank* operations, and here for symbols 0 and 3 a single *rank* operation is used while for the two remaining symbols we need two operations. Under the realistic assumption of approximately equal probability of each of the k symbols in the compressed stream, we obtain the 1.5 slowdown factor.

***Sparse representation of** Bh.* Imagine a very large alphabet and also a relatively large zero-order entropy H_0 of the text. In this case the array *Bh* will consist of mostly zeros. More precisely, between $n'/(H_0 + 1)$ and n'/H_0 bits from *Bh* will be zeros. The theoretical idea we propose is to replace the original *Bh* with a multilevel structure. The top level array will store $\lceil n'/h \rceil$ bits, where h is some parameter to fix soon. Each bit of this array corresponds to an h-bit chunk in *Bh*, and it will be 1 iff its chunk contains some bit set. If h is small enough compared to H_0, most of those h-bit chunks will contain all zeros, which is beneficial since the *rank* structures will not be built over those chunks. Performing *rank* for *Bh* will be replaced

in this variant with two *rank* operations: one for the top level array which will return the number of non-zero h-bit chunks from the beginning of Bh, and one for the second level of the structure, which stores information about 1s in the non-zero chunks. It is easy to find that the optimal h in terms of minimizing the worst-case space is $O(\sqrt{H_0+1})$, for which we need at most $2n\sqrt{H_0+1}$ bits (approximately) for such Bh substitute.

It is possible to generalize this idea to more than two levels. With i levels we obtain $O((i+1)n(H_0+1)^{(1/(i+1))})$ space, with $O(i)$ access cost to our Bh representation. The space for Bh is optimized for $i = \ln(H_0+1)$, where it reaches $en \ln(H_0+1)$ bits.

Note that each access to Bh costs $O(\log(H_0+1))$ in time. Recall that the array Bh is accessed only twice in a counting query, but many times in locating and displaying queries.

5.9 Experimental results

Implementation of our indexes (original FM-Huffman, its k-ary and the KZ versions), required making some practical considerations that differ from the theoretical ones. The main difference is the calculation of *rank* and *Occ*, where we used the solution described in [GGMN05], for the older index variants (binary and k-ary FM-Huffman), and a somewhat different *rank* implementation (see [PGNS06, Sect. 3]) for FM-KZ. The new indexes will be called FM-KZ1 and FM-KZ2, corresponding to the parameters $k = 1$ and $k = 2$, respectively.

In this section we show experimental results on counting, reporting and displaying queries and compare the efficiency to existing indexes. The results were presented in the paper [PGNS06]. The indexes used for the experiments were the FM-index implemented by Navarro [Nav04], Sadakane's CSA [Sad00], the RLFM index [MN05c], the SSA index [MN05c] and the LZ index [Nav04]. Other indexes whose implementations were available at the time of experiments were not included because they are not comparable to the FM-Huffman / FM-KZ index due either to their large space requirement or their high search times.

We considered three types of text for the experiments: 80 MB of English text obtained from the TREC-3 collection [4] (files `WSJ87-89`), 60 MB of DNA and 55 MB of protein sequences, both obtained from the BLAST database of the NCBI[5] (files `month.est_others` and `swissprot` respec-

[4]Text Retrieval Conference, http://trec.nist.gov

[5]National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov

tively).

The experiments were conducted on an Intel Xeon processor at 3.06 GHz, 2 GB of RAM and 512 KB cache, running Gentoo Linux 2.6.10. The codes were compiled with `gcc 3.3.5` using optimization option `-O9`.

Now we show the results regarding the space used by our index and later the results of the experiments divided in query type.

5.9.1 Space results

For the experiments we considered the binary, the 4-ary, and the KZ versions of our index. It is interesting to know how the space requirement of the Huffman-based index varies according to the parameter k. Table 5.1 (top) shows the space that the index takes as a fraction of the text for different values of k and the three types of files considered. These values do not include the space required to report positions and display text. It should also be mentioned that the value corresponding to the row $k = 8$ for DNA actually corresponds to $k = 5$, since this is the total number of symbols to code in this file. Similarly, the value of row $k = 32$ for the protein sequence corresponds to $k = 24$.

One can see that the space requirements are lowest for $k = 4$. For higher values this space increases, although staying reasonable until $k = 16$. With higher values the spaces are too high for these indexes to be comparable to the rest. It would be interesting to study the time performance to the versions of the index with $k = 8$ and $k = 16$. With $k = 8$, for some technical reason, we do not expect an improvement on the query time since $\log k$ is not a power of 2 and therefore the computation of Occ is slower. The version with $k = 16$ could lead to a reduction in query time, but the access to 4 machine words for the calculation of Occ could negatively affect it. It is important to say that these values are only relevant for the English text and proteins, since it makes no sense to use them for DNA.

It is also interesting to see how the space requirement of the index is divided among its different structures. Table 5.1 (bottom) shows the space used by each of the structures for the index with $k = 2$ and $k = 4$ for the three types of texts considered.

For higher values of k the space used by B will increase since the use of more symbols for the Huffman codes increases the resulting space. On the other hand, the size of Bh decreases at a rate of $\log k$ and so do its *rank* structures. However, the space of the *rank* structures of B increases rapidly, as we need k structures for an array that reduces its size at a rate of $\log k$, which is the reason of the large space requirement for high values of k.

Table 5.1: (Top) Space requirement of FM-Huffman for different values of k. (Bottom) Detailed comparison of $k = 2$ versus $k = 4$. The spaces used by the Huffman table, the constant-size tables for *rank*, and array C, are omitted (negligible).

k	Fraction of text		
	English	DNA	Proteins
2	1.68	0.76	1.45
4	1.52	0.74	1.30
8	1.60	0.91	1.43
16	1.84	—	1.57
32	2.67	—	1.92
64	3.96	—	—

Structure	**FM-Huffman** $k = 2$			**FM-Huffman** $k = 4$		
	Space [MB]			Space [MB]		
	English	DNA	Proteins	English	DNA	Proteins
B	48.98	16.59	29.27	49.81	18.17	29.60
Bh	48.98	16.59	29.27	24.91	9.09	14.80
$Rank(B)$	18.37	6.22	10.97	37.36	13.63	22.20
$Rank(Bh)$	18.37	6.22	10.97	9.34	3.41	5.55
Total	134.69	45.61	80.48	121.41	44.30	72.15
Text	80.00	60.00	55.53	80.00	60.00	55.53
Fraction	1.68	0.76	1.45	1.52	0.74	1.30

Now, let us take a look at the FM-KZ1 and FM-KZ2 space/time behavior. For DNA, the FM-KZ1 is a clear winner: among the fastest and definitely the most succinct, also it is hard to imagine a simpler full-text index (as the encoding is merely the unary code).

On the English text, FM-KZ2 is takes about $1.0n$ space, much less than other indexes from our family, but is also considerably slower, e.g. more than 1.5 times slower than FM Huffman with $k = 4$.

5.9.2 Counting queries

For the three files, we show the search time as a function of the pattern length, varying from 10 to 100, with a step of 10. For each length we

used 1000 patterns taken from random positions of each text. Each search was repeated 1000 times. Figure 5.6 (left) shows the time for counting the occurrences for each index and for the three files considered. As the CSA index needs a parameter to determine its space for this type of queries, we adjusted it so that it would use approximately the same space that the binary FM-Huffman index.

We also show the average search time per character along with the minimum space requirement of each index to count occurrences. Unlike the CSA, the other indexes do not need a parameter to specify their size for counting queries. Therefore, we show a point as the value of the space used by the index and its search time per character. For the CSA index we show a line to resemble the space-time tradeoff for counting queries. The time per character for each pattern length is the search time divided by the value of the length. The time per character shown on the plot is the average of these times for each length. Figure 5.6 (right) shows the search time per character for each index and for each type of text.

5.9.3 Reporting queries

We measured the time that each index took to search for a pattern and report the positions of the occurrences found. From the English text and the DNA sequence we took 1000 random patterns of length 10. From the protein sequence we used patterns of length 5. We measured the time per occurrence reported varying the space requirement for every index except the LZ, which has a fixed size. For the CSA we set the two parameters, namely the size of the structures to report and the structures to count, to the same value, since this turns out to be optimal. Figure 5.7 shows the times per occurrence reported for each index as a function of its size.

5.9.4 Displaying text

We measured the time that each index took to show the first character of a text context around the occurrences found. More precisely, this is the time of searching for a pattern, locating the position of an occurrence and showing one character of the text in the context area of the position located. We measured this time as a function of the size used by each index. We used the same 1000 patterns as in the reporting experiment. Figure 5.8 (left) shows the average time to display the first character as a function of the space needed for each index and for each type of text.

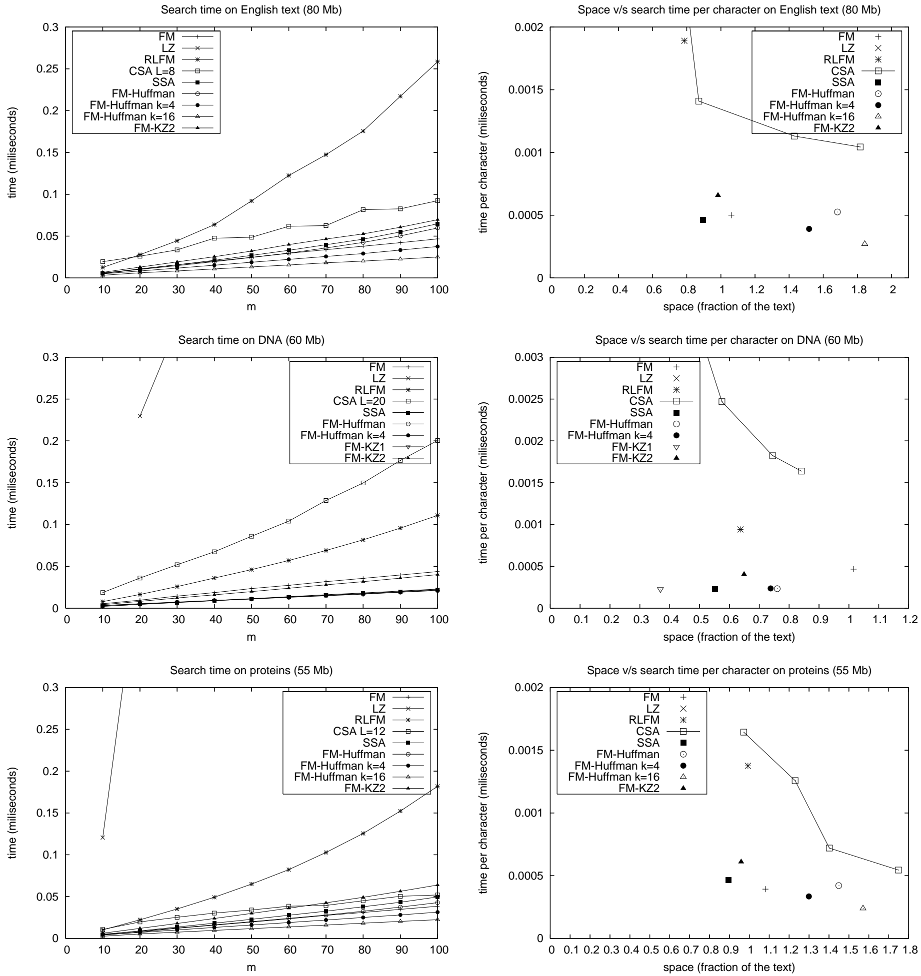

Figure 5.6: (Left) Search time as a function of the pattern length, English (80 MB), DNA (60 MB) and proteins (55 MB). The times of the LZ index do not appear on the English text plot, as they range from 0.5 to 4.6 ms. In the DNA plot, the time of the LZ index for $m = 10$ is 2.6. The reason of this increase is the large number of occurrences of these patterns, which influences the counting time for this index. (Right) Average search time per character as a function of the size of the index

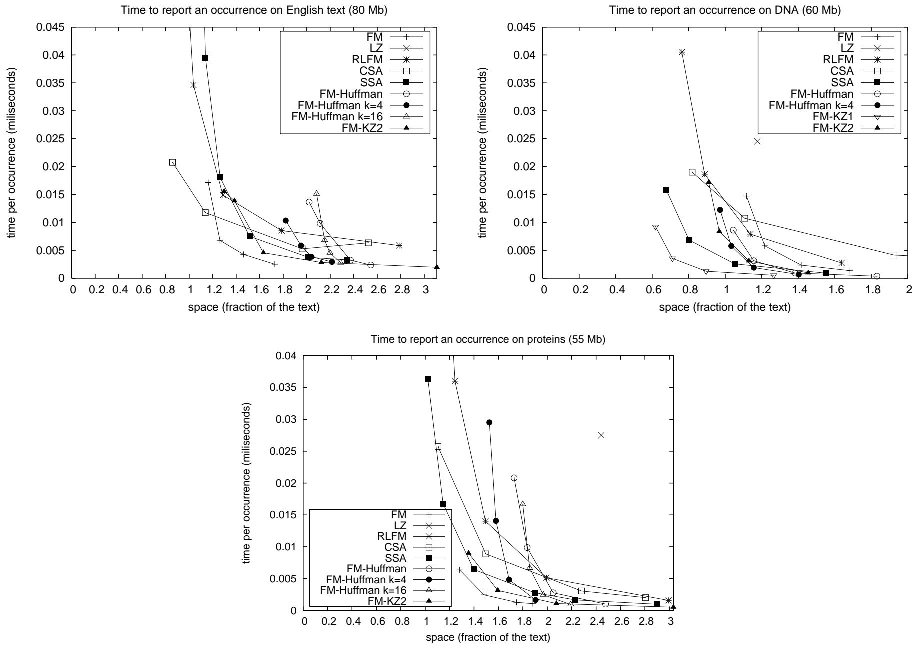

Figure 5.7: Time to report the positions of the occurrences as a function of the size of the index. We show the results of searching on 80 MB of English text, 60 MB of DNA and 55 MB of proteins

In addition, we measured the time to display a context per character displayed. That is, we searched for the 1000 patterns and displayed 100 characters around each of the positions of the occurrences found. We subtracted from this time the time to display the first character and divided it by the amount of characters displayed. Figure 5.8 (right) shows this time along with the minimum space required for each index for the counting functionality, since the display time per character does not depend on the size of the index. This is not true for the CSA index, whose time to display per character does depend on its size. For this index we show the time measured as a function of its size.

5.9.5 Analysis of results

We can see that our FM-Huffman $k = 4$ and $k = 16$ indexes are among the fastest for counting queries for the three types of files. The binary FM-Huffman index takes the same time that $k = 4$ version for DNA and it is a little bit slower that the FM-index for the other two files. As expected,

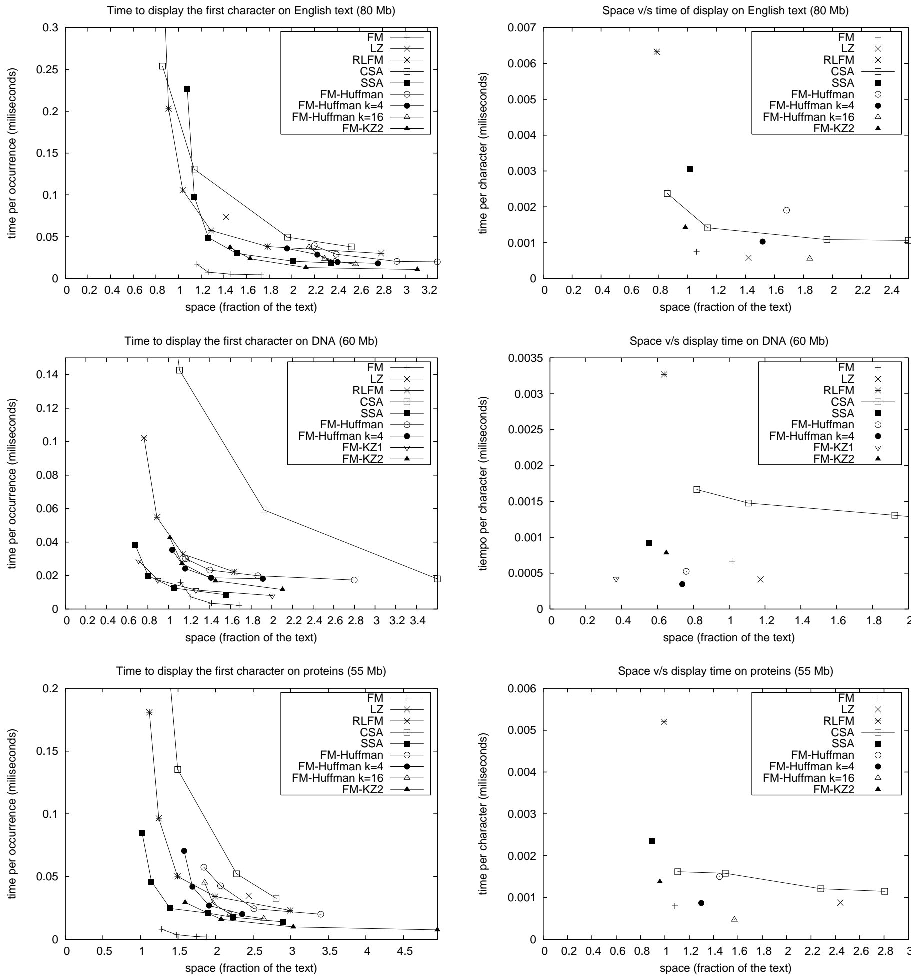

Figure 5.8: (Left) Time to show the first character of a text context around the positions of the occurrences as a function of the size of the index. From top to bottom, we show the results of searching 80 MB of English text, 60 MB of DNA and 55 MB of proteins. In the plot of the DNA sequence, the point corresponding to the LZ index is covered. Its value is: space=1.18, time=0.03. (Right) Time per character displayed around an occurrence and space for each index

all those versions are faster than CSA, RLFM and LZ, the latter not being competitive for counting queries. Regarding the space usage, the FM-index turns out to be a better tradeoff alternative for the English text and protein sequences, since it uses less space than our index and has low search times. For DNA, all the Huffman based versions of our index are good alternatives, considering their low space requirement and search time.

Still, the new player, FM-KZ index, is a particularly good choice for DNA. It is way ahead of the competition in the space use, while belonging to the fastest. At the same time its simplicity is striking.

Considering both speed and space use, for the English text and the proteins, the SSA index is the best choice, still, our variants come close, especially for proteins.

For reporting queries, our index loses to the FM-index for English and proteins, mainly because of its large space requirement. Also, it only surpasses the RLFM and CSA for large space usages. For DNA, however, our index, with the two versions, is better than the FM-index. This reduction in space is due to the low zero-order entropy of the DNA, which makes our index compact and fast.

Regarding the time for displaying the first character, the FM-index is faster than our index. Again, our index takes more space than the other indexes to get competitive time for English and proteins, and reduces its space for DNA. Regarding display time per character, our index with $k = 4$ is the fastest for DNA with a low space requirement, becoming an interesting alternative for this type of query.

The version of our index with $k = 4$ improved both the space and time with respect to the binary version and it became a very good alternative for counting and reporting queries, especially for DNA, due to the low zero-order entropy of this text.

5.10 Recent advancements in compressed indexes

The progress in the area of compressed indexes is amazingly fast. Basically, we could list seven research directions here:

- Novel *rank*/*select* implementations, either theoretical or practical.
- Succinct suffix trees.
- Adding dynamic capabilities.
- External indexes.
- Word-based compressed indexes.
- Indexing structured text (XML in particular).

- Handling more advanced queries, including approximate pattern matching.

Now we are going to briefly address the mentioned approaches.

New** rank/select **ideas. In Sect. 5.5.3 we mentioned some order-0 and even order-k entropy bound solutions for *rank* and *select*. It took a few years until experimental works for this topic started to appear. Sadakane and Okanohara [OS07] presented a couple of practical variants for compressed *rank* and *select*, while Claude and Navarro [CN08] proposed novel implementations of the solutions of Raman et al. [RRR02] and Golynski [GMR06]. Even without keeping the sequence in compressed form, there still seems some room for improvement: Vigna [Vig08] showed efficient *rank* and *select* variations designed for 64-bit computer registers. The so-called "broadword computing" is a recent trend, whose impact on indexes may go beyond *rank* and *select* implementations, as the very recent work [Gog09] on speeding up queries with CSA demonstrates.

Succinct suffix trees. Attempts to represent a suffix tree compactly have a long history, but the first fully-functional compressed ST was presented only recently, in 2007, by Sadakane [Sad07]. The space used in Sadakane's solution was proportional to order-zero entropy of the text. This could be improved to order-k entropy using techniques from [GGV03], but in any case the structure has an overhead of $6n$ bits. This was removed in a variant by Russo et al. [RNO08b], and soon the same result was presented even in a dynamic setting [RNO08a]. Yet another variant, with space use in between the Sadakane and Russo et al. structures, was presented in [FMN08].

Supporting dynamism. Yet the seminal paper by Ferragina and Manzini [FM00] discussed how their scheme can be modified to work with a collection of texts (e.g., web pages), which may shrink or grow over time, because of *insert* or *delete* operations applied to whole texts (still, their time complexities for the update operations hold only in the amortized sense). Other ideas and novel algorithms were given in, among others, [CHL04, MN08].

External indexes. For huge enough text collections, even compressed indexes may not fit in main memory. An interesting question is then if they can compete in speed with existing non-compressed external full-text indexes, particularly the String B-tree [FG99] and the disk-based SA [BYBZ96] mentioned earlier in this chapter. The existing structures are usually (non-

trivially) tailored variants of existing main-memory compressed indexes; e.g., the structure from [AN07] is a disk-based variant of LZ-index and the one from [GN07a] is based on the FM-index. Especially the latter is a successful data structure, according to experiments presented in [GN07a]: for compressible data (like XML) it seems more interesting, both in counting and locating, than other tested indexes, including the widely acclaimed String B-tree.

Word-based compressed indexes. Texts in most human languages are naturally segmented on space boundaries, but the notion of a "word" is helpful even in some mildly artificial data, like computer logs. The success of word-targeted compression codes, described in the previous chapter, encourages to use them also for indexing. In fact, compression has been widely applied in inverted indexes, but in full-text indexes was introduced rather recently [Fer08, FNP08]. Experiments in the latter of the cited works show that for NL text (English) a word-based index from the FM family can reach about 40% or less of the original size, at a practical speed. This is a significant improvement in comparison with character-based compressed indexes (cf. [FGNV09]).

Indexing structured text. Ferragina et al. [FLMM06] proposed a BWT-related transform for labeled trees (XML structure in particular), which, combined with compression, enables fast searching in and navigation over the tree. Very recently, Brisaboa et al. [BCN09] presented a structure called XML wavelet tree which represents an XML in compressed form and permits to answer XPath queries more efficiently than without compression. Undoubtedly, this research area is only starting to develop and many refinements of existing techniques or perhaps even major discoveries are yet to come.

Extended functionality. This area is still underdeveloped, with a few works on approximate matching in compressed setting [HHLS06, CLS+06, RNO07]. A compressed index with position-restricted searching was proposed in [CHSV06]. Succinct suffix trees, mentioned above, may be able to serve other matching models, with applications e.g. in bioinformatics.

The amazing progress in compressed indexes may revolutionize not only the field of stringology, but even the way we perceive data structures in general. It is conceivable to think that "compressed" is going to mean not just "using less space" but "better in every aspect".

Chapter 6

Conclusions

String matching is an exciting research field, since very easily formulated problems have often rather sophisticated solutions and the abundance of algorithmic approaches to classic tasks (exact matching, k-mismatches, k-differences, etc.) proves the intrinsic richness of those fundamental combinatorial problems. In this dissertation we focused on selected research areas, as the whole field is much too broad for the scope of this study.

In algorithmics, there are basically two kinds of results: one is solving a particular problem (via presenting a novel algorithm for a given problem with better properties than its predecessors, demonstrating a new competitive data structure, showing an innovative analysis for an old algorithm etc.) and the other is to present a more general technique, an idea which, if applied appropriately, may serve to bring progress to many (sometimes seemingly not too related) problems.

In this book we presented a fair amount of new string matching algorithms, addressing many particular matching problems, but we are most happy with two simple yet efficient bit-parallel techniques which we worked out in collaboration with Kimmo Fredriksson. One is about sampling text in regular intervals, which transforms a single pattern matching task into a multiple pattern matching filter, and yet is suprisingly simple to implement using bit logic and efficient especially on modern hardware (as the experimental results contained in Chap. 1 clearly show). The other is to replace multi-bit counters in algorithms that do maintain counters (which are not uncommon), with a set of nested counters, the smallest of them used most of the time and flushed out periodically, thus enabling to handle more counters at a time (in the amortized sense) and improving overall worst-case time.

We have reached several goals in our research. First, the mentioned novel techniques proved useful, in practice and/or theory, for a number of classic string matching problems: single and multiple exact matching, matching with local swaps, (δ, γ)-matching, k-mismatches, episode matching, and more. Second, we significantly broadened the set of tools for matching with gaps, a class of models important in bioinformatics, but also in music information retrieval. For example, one result we achieved sort of "en passent" was an effective handling of negative gaps for (δ, α)-matching, using our bit-parallel dynamic programming algorithm, a problem previously considered hard. Third, our results added a brick or two to confirm the importance of compression in modern algorithmics. Searching a pattern directly in our q-gram based compressed text was shown to be faster than in non-compressed text. Although this is not the first achievement of this kind, our scheme is a very simple example of compression supporting full-text searches, i.e. working with arbitrary data. Another (minor) result in the area of online compressed matching was an idea of making known dense codes (e.g., (s, c)-DC), for a static text, even slightly denser, with using byte combinations not occurring in the actual compressed stream. Finally, we proposed one of the simplest known compressed full-text indexes (from the FM family), offering fairly good space/time tradeoffs, and analyzed a number of its modifications.

A number of avenues explored in this work can be pursued further. For example, the Shift-Or related technique of sampling text in regular intervals can be adapted for matching over packed data, in particular, binary stream; from the results contained in [FL09] we can expect our technique to be a competitive one for this problem as well. Another future task (currently under development) is to apply Matryoshka counters to improve yet a few bit-parallel algorithms, e.g. the one from [HN06] for local similarity computation with unitary cost weights. The q-gram based scheme for compression and search may be improved in some ways, as pointed out in conclusions to Chap. 4, but another research possibility for compressed searching is to get back to its roots and devise efficient bit-wise Huffman based implementations; the problem concerns match verification, which can be served with an extra stream of synchronizing data or a modification of Huffman coding with almost no loss in compression and a small guaranteed synchronization delay [Bis08].

The most general, yet strong, conclusion that we have come to is that stringology is full of surprises and new light may be cast even on very old problems. The revolution in hardware architecture which has started in the last few years (broadword computing, multi-core CPUs, GPU massively parallel programming, Flash memories as an alternative to spinning hard

disks) should also affect strongly the algorithmic designs and promote new, more accurate, theoretical models of computations. It also seems certain, for the predictable future, that the paradigms of bit-parallelism, making use of wide machine words, and compression, which turns data into their denser, more informative representations, will become almost ubiquitous in state-of-the-art algorithms of tomorrow.

Bibliography

[AB92] A. Amir and G. Benson. Two-dimensional periodicity and its applications. In *Proc. 3rd ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 440–452. SIAM, 1992.

[Abe07] J. Abel. Incremental frequency count – a post BWT-stage for the Burrows–Wheeler compression algorithm. *Software–Practice and Experience*, 37(3):247–265, 2007.

[ABF94] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM Journal on Computing*, 23(2):313–323, 1994.

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[AC91] A. Apostolico and M. Crochemore. Optimal canonization of all substrings of a string. *Information and Computation*, 95(1):76–95, 1991.

[ACH+01] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. In *Proc. 12th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 279–288. SIAM, 2001.

[ACR99] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In *Proc. 26th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 1725, pages 291–306. Springer, 1999.

[AD86] L. Allison and T. I. Dix. A bit string longest common subsequence algorithm. *Information Processing Letters*, 23(6):305–310, 1986.

[ADKF75] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a direct graph. *Soviet Mathematics, Doklady*, 11:1209–1210, 1975. Original in Russian in Doklady Akademii Nauk SSSR, vol. 194, 1970.

[AdlFN05] J. Adiego, P. de la Fuente, and G. Navarro. Combining structural and textual contexts for compressing semistructured databases. In *Proc. Int. Mexican Conference in Computer Science (ENC'05)*, pages 68–73, Puebla, Mexico, 2005. IEEE CS Press.

[AE05] A. N. Arslan and Ö. Egecioglu. Algorithms for the constrained longest common subsequence problems. *International Journal of Foudations of Computer Science*, 16(6):1099–1109, 2005.

[AG86] A. Apostolico and R. Giancarlo. The Boyer–Moore–Galil string searching strategies revisited. *SIAM Journal on Computing*, 15(1):98–105, 1986.

[AG87] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1–4):315–336, 1987.

[AHHP94] A. Anderson, T. Hagerup, J. Hastad, and O. Petersson. The complexity of searching a sorted array of strings. In *Proc. 26th ACM Symposium on the Theory of Computing (STOC)*, pages 317–325. ACM Press, 1994.

[AHHP01] A. Andersson, T. Hagerup, J. Hastad, and O. Petersson. Tight bounds for searching a sorted array of strings. *SIAM Journal on Computing*, 30(5):1552–1578, 2001.

[AKO04] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[ALM02] O. Arbell, G. M. Landau, and J. S. B. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2002.

[ALP00] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In *Proc. 11th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 794–803. SIAM, 2000.

[ALPU05] A. Amir, O. Lipsky, E. Porat, and J. Umanski. Approximate matching in the l_1 metric. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3537, pages 91–103. Springer, 2005.

[ALS96] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1075, pages 102–115. Springer, 1996.

[ALS99] A. Apostolico, G. M. Landau, and S. Skiena. Matching for run-length encoded strings. *Journal of Complexity*, 15:4–16, 1999.

[AN07] D. Arroyuelo and G. Navarro. A Lempel–Ziv text index on secondary storage. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 83–94, 2007.

[ANdlF07] J. Adiego, G. Navarro, and P. de la Fuente. Lempel–Ziv compression of highly structured documents. *Journal of the American Society for Information Science and Technology (JASIST)*, 58(4):461–478, 2007.

[ANZ97] M. D. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. 4th South American Workshop on String Processing*, pages 2–20. Carleton University Press, 1997.

[Apo97] A. Apostolico. String editing and longest common subsequences. In *Handbook of Formal Languages*, volume 2 Linear Modeling: Background and Application, chapter 8, pages 361–398. Springer, 1997.

[AR99] C. Allauzen and M. Raffinot. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, France, 1999.

[AT05] J. Abel and W. J. Teahan. Universal text preprocessing for data compression. *IEEE Transactions on Computers*, 54(5):497–507, 2005.

[BAHM07] J. Barbay, L. Castelli Aleardi, M. He, and J. I. Munro. Succinct representation of labeled graphs. Technical Report CS-2007-11, University of Waterloo, Ontario, Canada, 2007.

[BCN09] N. Brisaboa, A. Cerdeira, and G. Navarro. A compressed self-indexed representation of XML documents. In *Proc. 13th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, LNCS 5714, pages 273–284, 2009.

[BFG07] Ph. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on Ziv–Lempel compressed texts. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 52–62. Springer, 2007.

[BFNE03] N. Brisaboa, A. Fariña, G. Navarro, and M. Esteller. (s,c)-dense coding: An optimized compression code for natural language text databases. In *Proc. 10th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2857, pages 122–136. Springer, 2003.

[BFNP04] N. Brisaboa, A. Fariña, G. Navarro, and J. L. Paramá. Simple, fast, and efficient natural language adaptive compression. In *Proc. 11th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 230–241. Springer, 2004.

[BFNP05] N. Brisaboa, A. Fariña, G. Navarro, and J. L. Paramá. Efficiently decodable and searchable natural language adaptive compression. In *Proc. 28th Int. Conference on Research and Development in Information Retrieval (SIGIR)*, pages 234–241. ACM Press, 2005.

[BFNP06] N. Brisaboa, A. Fariña, G. Navarro, and J. L. Paramá. Improving semistatic compression via pair-based coding. In *Proc. 6th Int. Conference on Perspectives of System Informatics (PSI'06)*, LNCS 4378, pages 124–134, 2006.

[BFNP07] N. Brisaboa, A. Fariña, G. Navarro, and J. L. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[BGS72] M. Beeler, R. W. Gosper, and R. Schroeppel. HAKMEM. MIT AI Memo 239, 1972. http://www.inwap.com/pdp10/hbaker/hakmem/algorithms.html#item179 (link verified Oct. 23, 2009).

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. 7th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, pages 39–48. IEEE Computer Society, 2000.

[Bil09] Ph. Bille. Fast searching in packed strings. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5577, pages 116–126, 2009.

[Bin00] E. Binder. Usenet group: comp.compression, 2000.

[BINP03] N. Brisaboa, E. Iglesias, G. Navarro, and J. L. Paramá. An efficient compression code for text databases. In *Proc. 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481. Springer, 2003.

[Bis08] M. T. Biskup. Guaranteed synchronization of Huffman codes. In *Proc. Data Compression Conference (DCC)*, pages 462–471, Snowbird, UT, 2008. IEEE Computer Society Press.

[BK00] B. Balkenhol and S. Kurtz. Universal data compression based on the Burrows–Wheeler transformation: Theory and practice. *IEEE Transactions on Computers*, 49(10):1043–1053, 2000.

[BKS99] B. Balkenhol, S. Kurtz, and Y. M. Shtarkov. Modifications of the Burrows and Wheeler data compression algorithm. In *Proc. Data Compression Conference (DCC)*, pages 188–197, 1999.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[BMM97] A. Brodnik, P. B. Miltersen, and J. I. Munro. Trans-dichotomous algorithms without multiplication – some upper and lower bounds. In *Proc. 5th Workshop on Algorithms and Data Structures (WADS)*, LNCS 1272, pages 426–439. Springer, 1997.

[BR99] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In *Proc. Prague Stringology Club Workshop '99*, pages 16–28, Czech Technical University, Prague, Czech Republic, 1999. Collaborative Report DC–99–05.

[Bre93] D. Breslauer. Saving comparisons in the Crochemore–Perrin string matching algorithm. In *Proc. 1st Annual European Symposium on Algorithms (ESA)*, LNCS 726, pages 61–72. Springer, 1993.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, CA, 1994.

[BY89a] R. A. Baeza-Yates. *Efficient text searching.* Ph. D. Thesis, Department of Computer Science, University of Waterloo, Ontario, Canada, 1989.

[BY89b] R. A. Baeza-Yates. Improved string searching. *Software–Practice and Experience*, 19(3):257–271, 1989.

[BYBZ96] R. A. Baeza-Yates, E. F. Barbosa, and N. Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[BYG89] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In *Proc. 12th Int. Conference on Research and Development in Information Retrieval (SIGIR)*, pages 168–175. ACM Press, 1989.

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[BYN96] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1075, pages 1–23, Laguna Beach, CA, 1996. Springer.

[BYN97] R. A. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. 5th Workshop on Algorithms and Data Structures (WADS)*, LNCS 1272, pages 174–184. Springer, 1997.

[BYN99] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[BYN00] R. A. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *J. of the American Society for Information Science (JASIS)*, 51(1):69–82, January 2000.

[BYN04] R. A. Baeza-Yates and G. Navarro. *Text Searching: Theory and Practice*, pages 565–597. Studies in Fuzzyness and Soft Computing 148. Springer, 2004. ISBN 3-540-20907-7.

[CCF05a] D. Cantone, S. Cristofaro, and S. Faro. An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA)*, LNCS 3503, pages 428–439. Springer, 2005.

[CCF05b] D. Cantone, S. Cristofaro, and S. Faro. On tuning the (δ, α)-sequential-sampling algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proc. 6th Int. Conference on Music Information Retrieval (ISMIR)*, 2005.

[CCF08] D. Cantone, S. Cristofaro, and S. Faro. New efficient bit-parallel algorithms for the δ-matching problem with α-bounded gaps in musical sequences. In *Proc. 12th Prague Stringology Conference (PSC)*, pages 170–184, 2008.

[CCG+94] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CCI+99] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In *Proc. 10th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, pages 129–144, 1999.

[CCI05] P. Clifford, R. Clifford, and C. S. Iliopoulos. Faster algorithms for δ, γ-matching and related problems. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3537, pages 68–78. Springer, 2005.

[CGL04] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th ACM Symposium on the Theory of Computing (STOC)*, pages 91–100. ACM, 2004.

[CGR99] M. Crochemore, L. Gąsieniec, and W. Rytter. Constant-space string-matching in sublinear average time. *Theoretical Computer Science*, 218(1):197–203, 1999.

[CH92] R. Cole and R. Hariharan. Tighter bounds on the exact complexity of string matching. In *Proc. 33rd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–609. IEEE Computer Society Press, 1992.

[CH97] R. Cole and R. Hariharan. Tighter upper bounds on the exact complexity of string matching. *SIAM Journal on Computing*, 26(3):803–856, 1997.

[CH98] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. In *Proc. 9th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 463–472. SIAM, 1998.

[CHL04] H.-L. Chan, W.-K. Hon, and T. W. Lam. Compressed index for a dynamic collection of texts. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 445–456, 2004.

[CHSV06] Y.-F. CHien, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed text indexing and range searching. Technical Report TR-06-021, Purdue University, Department of Computer Science, 2006.

[CI04] R. Clifford and C. S. Iliopoulos. Approximate string matching for music analysis. *Soft Computing*, 8:597–603, 2004.

[CIM+02] M. Crochemore, C. S. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic Journal of Computing*, 9(1):54–65, 2002.

[CIN+05] M. Crochemore, C. S. Iliopoulos, G. Navarro, Y. Pinzon, and A. Salinger. Bit-parallel (δ, γ)-matching suffix automata. *Journal of Discrete Algorithms*, 3(2–4):198–214, 2005.

[CIPR00] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. In *Proc. 11th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, pages 75–86, 2000.

[CL92] W. I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 664, pages 175–184. Springer, 1992.

[CL94] W. I. Chang and E. L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

[CL04] C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms.* King's College Publications, 2004.

[Cla96] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Ontario, Canada, 1996.

[CLP98] C. Charras, T. Lecroq, and J. D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1448, pages 55–64. Springer, 1998.

[CLS+06] H.-L. Chan, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. A linear size index for approximate pattern matching. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 49–59. Springer, 2006.

[CM94] W. I. Chang and T. G. Marr. Approximate string matching and local similarity. In *Proc. 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 807, pages 259–273. Springer, 1994.

[CM05] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc. 12th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3772, pages 1–12. Springer, 2005.

[CM06] J. S. Culpepper and A. Moffat. Phrase-based pattern matching in compressed text. In *Proc. 13th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4209, pages 337–345. Springer, 2006.

[CMRS98] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Data compression using antidictionaries. Technical Report IGM 98-10, Institut Gaspard-Monge, Université de Marne-la-Vallée, France, 1998.

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187. Springer, 2008.

[CO06] L. P. Coelho and A. L. Oliveira. Dotted suffix trees a structure for approximate text indexing. In *Proc. 13th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4209, pages 329–336. Springer, 2006.

[Col91] L. Colussi. Correctness and efficiency of the pattern matching algorithms. *Information and Computation*, 95(2):225–251, 1991.

[CP91] M. Crochemore and D. Perrin. Two-way string-matching. *The Journal of the ACM*, 38(3):651–675, 1991.

[CT97] J. G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–75, 1997.

[CW84] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Computers*, 32:396–402, 1984.

[Dal02] H. Dalianis. Evaluating a spelling support in a search engine. In *Proc. 6th Int. Conference on Applications of Natural Language to Information Systems–Revised Papers*, LNCS 2553, pages 183–190, London, UK, 2002. Springer.

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[DC05] S. Deorowicz and M. G. Ciura. Correcting spelling errors by modelling their causes. *International Journal of Applied Mathematics and Computer Science*, 15(2):275–285, 2005.

[Deo02] S. Deorowicz. Second step algorithms in the Burrows–Wheeler compression algorithm. *Software–Practice and Experience*, 32(2):99–111, 2002.

[Deo03] S. Deorowicz. *Universal lossless data compression algorithms.* Ph. D. Thesis, Silesian University of Technology, Gliwice, Poland, 2003.

[Deo05] S. Deorowicz. Context exhumation after the Burrows–Wheeler transform. *Information Processing Letters*, 95(1):313–320, 2005.

[Deo06] S. Deorowicz. Speeding up transposition-invariant string matching. *Information Processing Letters*, 100(1):14–20, 2006.

[Deo07] S. Deorowicz. Fast algorithm for the constrained longest common subsequence problem. *Theoretical and Applied Informatics*, 19(2):91–102, 2007.

[Deo09] S. Deorowicz. An algorithm for solving the longest increasing circular subsequence problem. *Information Processing Letters*, 109(12):630–634, 2009.

[Deu96] P. Deutsch. [RFC 1951] DEFLATE Compressed Data Format Specification version 1.3, 1996.

[DFG+97] G. Das, R. Fleischer, L. Gąsieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1264, pages 12–27. Springer, 1997.

[DG09a] S. Deorowicz and Sz. Grabowski. A hybrid algorithm for the longest common transposition-invariant subsequence problem. *Computing and Informatics*, 2009. Accepted.

[DG09b] S. Deorowicz and Sz. Grabowski. On two variants of the longest increasing subsequence problem. In *Proc. Int. Conference on Man-Machine Interactions (ICMMI)*, pages 541–549, 2009.

[DHPT09] B. Durian, J. Holub, H. Peltola, and J. Tarhio. Tuning BNDM with q-grams. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 29–37. SIAM, 2009.

[DO09] S. Deorowicz and J. Obstój. Constrained longest common subsequence computing algorithms in practice. Technical report, Silesian University of Technology, Gliwice, 2009. http://sun.aei.polsl.pl/~sdeor/pub/tr_do2009.pdf.

[Döm64] B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, 3:29–46, 1964.

[EGGI92] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming I: Linear cost functions. *The Journal of the ACM*, 39(3):519–545, 1992.

[Eli75] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

[Fen96] P. M. Fenwick. Block sorting text compression–final report. Report 130, Department of Computer Science, The University of Auckland, New Zealand, 1996.

[Fer08] P. Ferragina. String algorithms and data structures. *CoRR*, abs/0801.2378, 2008.

[FF07] P. Ferragina and J. Fischer. Suffix arrays on words. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 328–339. Springer, 2007.

[FG99] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *The Journal of the ACM*, 46:236–280, 1999.

[FG04] G. Franceschini and R. Grossi. No sorting? Better searching! In *Proc. 45th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 491–498, 2004.

[FG05a] K. Fredriksson and Sz. Grabowski. Efficient algorithms for (δ, α)-matching. Report A-2005-2, Department of Computer Science, University of Joensuu, 2005.

[FG05b] K. Fredriksson and Sz. Grabowski. Practical and optimal string matching. In *Proc. 12th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3772, pages 374–385. Springer, 2005.

[FG06a] K. Fredriksson and Sz. Grabowski. Efficient algorithms for (δ, γ, α)-matching. In *Proc. 11th Prague Stringology Conference (PSC)*, pages 29–40, 2006.

[FG06b] K. Fredriksson and Sz. Grabowski. Efficient algorithms for pattern matching with general gaps and character classes. In *Proc. 13th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4209, pages 267–278. Springer, 2006.

[FG06c] K. Fredriksson and Sz. Grabowski. Efficient bit-parallel algorithms for (δ, α)-matching. In *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA)*, LNCS 4007, pages 170–181. Springer, 2006.

[FG06d] K. Fredriksson and Sz. Grabowski. A general compression algorithm that supports fast searching. *Information Processing Letters*, 100(6):226–232, 2006.

[FG08a] K. Fredriksson and Sz. Grabowski. Efficient algorithms for (δ, γ, α) and $(\delta, k_\Delta, \alpha)$-matching. *International Journal of Foudations of Computer Science*, 19(1):163–184, 2008.

[FG08b] K. Fredriksson and Sz. Grabowski. Efficient algorithms for pattern matching with general gaps, character classes and transposition invariance. *Information Retrieval*, 11(4):335–357, 2008.

[FG09a] K. Fredriksson and Sz. Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009.

[FG09b] K. Fredriksson and Sz. Grabowski. Fast convolutions and their applications in approximate string matching. In *Pre-Proc. 20th Int. Workshop on Combinatorial Algorithms (IWOCA 2009)*, pages 363–372, 2009.

[FG09c] K. Fredriksson and Sz. Grabowski. Nested counters in bit-parallel string matching. In *Proc. 3rd Int. Conference on Language and Automata Theory and Applications (LATA)*, LNCS 5457, pages 338–349. Springer, 2009.

[FGM06] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in BWT compression. In *Proc. 14th Annual European Symposium on Algorithms (ESA)*, LNCS 4168, pages 756–767. Springer, 2006.

[FGNV09] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:article 12, 2009. 30 pages.

[FL08] S. Faro and T. Lecroq. Efficient variants of the backward-oracle-matching algorithm. In *Proc. 12th Prague Stringology Conference (PSC)*, pages 146–160, 2008.

[FL09] S. Faro and T. Lecroq. An efficient matching algorithm for encoded DNA sequences and binary strings. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5577, pages 106–115. Springer, 2009.

[FLMM06] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *Proc. 15th Int. Conference on World Wide Web*, pages 751–760. ACM, 2006.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[FM01] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 269–278. SIAM, 2001.

[FMMN04] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 150–160. Springer, 2004.

[FMMN07] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[FMN06] K. Fredriksson, V. Mäkinen, and G. Navarro. Flexible music retrieval in sublinear time. *International Journal of Foudations of Computer Science*, 17(6):1345–1364, 2006.

[FMN08] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 152–165. Springer, 2008.

[FN04] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, 9:article 1.4, 2004. 45 pages.

[FN07] K. Fredriksson and F. Nikitin. Simple compression code supporting random access and fast string matching. In *Proc. 6th Workshop on Efficient and Experimental Algorithms (WEA)*, LNCS 4525, pages 203–216. Springer, 2007.

[FNP08] A. Fariña, G. Navarro, and J. L. Paramá. Word-based statistical compressors as natural language compression boosters. In *Proc. Data Compression Conference (DCC)*, pages 162–171. IEEE Computer Society Press, 2008.

[FP74] M. J. Fischer and M. Paterson. String matching and other products. In *Proc. SIAM-AMS Complexity of Computation*, pages 113–125, 1974.

[Fre75] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

[Fre00] K. Fredriksson. Fast algorithms for string matching with and without swaps. http://www.cs.uku.fi/~fredriks/pub/papers/sm-w-swaps.pdf, 2000.

[Fre02] K. Fredriksson. Faster string matching with super–alphabets. In *Proc. 9th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2476, pages 44–57. Springer, 2002.

[Fre03] K. Fredriksson. Shift-Or string matching with super-alphabets. *Information Processing Letters*, 87(4):201–204, 2003.

[FT03] K. Fredriksson and J. Tarhio. Processing of Huffman compressed texts with a super-alphabet. In *Proc. 10th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2857, pages 108–121. Springer, 2003.

[Gag94] Ph. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994.

[Gal78] R. G. Gallager. Variation on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[GB06] Sz. Grabowski and W. Bieniecki. Simple techniques for plagiarism detection in student programming projects. In *Proc. XIV Konferencja Sieci i Systemy Informatyczne–Teoria, Projekty, Wdrożenia*, pages 225–228, Łódź, 2006.

[GBYS92] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval Data Structures & Algorithms*. Prentice-Hall, 1992.

[GD08] Sz. Grabowski and S. Deorowicz. Nice to be a chimera: A hybrid algorithm for the longest common transposition-invariant subsequence problem. In *Proc. Int. Conference on Modern Problems of Radio Engineering, Telecommunications, and Computer Science (TCSET)*, pages 50–54, Lviv, Ukraine, 2008.

[GF08] Sz. Grabowski and K. Fredriksson. Bit-parallel string matching under Hamming distance in $O(n\lceil m/w \rceil)$ worst case time. *Information Processing Letters*, 105(5):182–187, 2008.

[GG92] Z. Galil and R. Giancarlo. On the exact complexity of string matching: upper bounds. *SIAM Journal on Computing*, 21(3):407–437, 1992.

[GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 841–850. SIAM, 2003.

[GGV04] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. 15th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 636–645. SIAM, 2004.

[GKS03] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software–Practice and Experience*, 33(11):1035–1049, 2003.

[GM07] T. Gagie and G. Manzini. Move-to-front, distance coding, and inversion frequencies revisited. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 71–82. Springer, 2007.

[GMN04a] Sz. Grabowski, V. Mäkinen, and G. Navarro. First Huffman, then Burrows–Wheeler: A simple alphabet-independent FM-index. Technical Report TR/DCC-2004-4, University of Chile, Department of Computer Science, 2004.

[GMN04b] Sz. Grabowski, V. Mäkinen, and G. Navarro. First Huffman, then Burrows–Wheeler: An alphabet-independent FM-index. In *Proc. 11th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 210–211. Springer, 2004.

[GMNS05] Sz. Grabowski, V. Mäkinen, G. Navarro, and A. Salinger. A simple alphabet-independent FM-index. In *Proc. 10th Prague Stringology Conference (PSC)*, pages 230–244, 2005.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 368–373. ACM Press, 2006.

[GN04] Sz. Grabowski and G. Navarro. $O(mn \log \sigma)$ time transposition invariant LCS computation. Technical Report TR/DCC-2004-6, University of Chile, Department of Computer Science, 2004. ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/transpszymon.ps.gz.

[GN07a] R. González and G. Navarro. A compressed text index on secondary memory. In *Proc. 18th Int. Workshop on Combinatorial Algorithms (IWOCA)*, pages 80–91. College Publications, UK, 2007.

[GN07b] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[GNP$^+$06] Sz. Grabowski, G. Navarro, R. Przywarski, A. Salinger, and V. Mäkinen. A simple alphabet-independent FM-index. *International Journal of Foudations of Computer Science*, 17(6):1365–1384, 2006.

[Gog09] S. Gog. Broadword computing and Fibonacci code speed up compressed suffix arrays. In *Proc. 8th International Symposium on Experimental Algorithms (SEA)*, LNCS 5526, pages 161–172. Springer, 2009.

[Gol06] A. Golynski. Optimal lower bounds for rank and select indexes. In *Proc. 33rd Int. Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4051, pages 370–381. Springer, 2006.

[GPR95] L. Gąsieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: sequential sampling. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 937, pages 78–89. Springer, 1995.

[Gra99] Sz. Grabowski. Text preprocessing for Burrows–Wheeler block-sorting compression. In *Proc. VII Konferencja Sieci i Systemy Informatyczne–Teoria, Projekty, Wdrożenia*, pages 229–239, Łódź, 1999.

[Gra08] Sz. Grabowski. Making dense codes even denser. *Automatyka*, 12(3):769–779, 2008.

[Gre04] I. Grebnov. The grzipII program. http://magicssoft.ru/?folder=projects\&page=GRZipII, 2004.

[Gri07] N. Grimsmo. On performance and cache effects in substring indexes. Technical Report IDI-TR-2007-04, NTNU, Department of Computer and Information Science, Sem Salands vei 7-9, NO-7491 Trondheim, NORWAY, 2007.

[GRRR04] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 159–172. Springer, 2004.

[GS83] Z. Galil and J. Seiferas. Time-space optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.

[GS03] Sz. Grabowski and Ł. Sturgulewski. An alternative traversal of the suffix array for the worst case, 2003. Talk at Forum Informatyki Teoretycznej (FIT).

[GT86] H. Gajewska and R. E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, 1986.

[Gus97] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, Cambridge, 1997.

[GV00] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on the Theory of Computing (STOC)*, pages 397–406. ACM Press, 2000.

[Har71] M. C. Harrison. Implementation of the substring test by hashing. *Communications of the ACM*, 14(12):777–779, 1971.

[Har95] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing and Management*, 31(3):271–289, 1995.

[HD05] J. Holub and B. Durian. Fast variants of bit parallel approach to suffix automata. Talk given in *The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation*, 2005. http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf.

[Hea78] H. S. Heaps. *Information Retrieval-Computational and Theoretical Aspects.* Academic Press, 1978.

[HF04] L. He and B. Fang. Linear nondeterministic dawg string matching algorithm. In *Proc. 11th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 70–71. Springer, 2004.

[HFN05] H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM Journal of Experimental Algorithmics*, 10:article 2.6, 2005. 27 pages. Special issue for best papers of *WEA'04*.

[HHLS06] T. N. D. Huynh, W.-K. Hon, T. W. Lam, and W.-K. Sung. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 352(1-3):240–249, 2006.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[Hir78a] D. S. Hirschberg. An information-theoretic lower bound for the longest common subsequence problem. *Information Processing Letters*, 7(1):40–41, 1978.

[Hir78b] D. S. Hirschberg. A lower worst-case complexity for searching a dictionary. In *Proc. 16th Annual Allerton Conference on Communication, Control, and Computing*, pages 50–53, 1978.

[HN06] H. Hyyrö and G. Navarro. Bit-parallel computation of local similarity score matrices with unitary weights. *International Journal of Foudations of Computer Science*, 17(6):1325–1344, 2006.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software–Practice and Experience*, 10(6):501–506, 1980.

[HS77] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[Huf52] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[Hyy01] H. Hyyrö. Explaining and extending the bit-parallel algorithms of Myers. Technical Report A-2001-10, University of Tampere, Finland, 2001.

[Hyy04] H. Hyyrö. Bit-parallel lcs-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, pages 16–27, University of Sydney, Australia, 2004.

[IR07] C. S. Iliopoulos and M. S. Rahman. New efficient algorithms for LCS and constrained LCS problem. In *Proc. 3rd Algorithms and Complexity in Durham Workshop*, Durham, UK, September 2007.

[IR08a] C. S. Iliopoulos and M. S. Rahman. Indexing circular patterns. In *Proc. Workshop on Algorithms and Computation*, Dhaka, Bangladesh, February 2008.

[IR08b] C. S. Iliopoulos and M. S. Rahman. A new model to solve the swap matching problem and efficient algorithms for short patterns. In *Proc. 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 4910, pages 316–327. Springer, 2008.

[Jac89] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1989.

[Joh82] D. B. Johnson. A priority queue in which initialization and queue operations take $O(\log\log D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2676, pages 200–210. Springer, 2003.

[Kau65] W. Kautz. Fibonacci codes for synchronization control. *IEEE Transactions on Information Theory*, 11:284–292, 1965.

[KB00] S. Kurtz and B. Balkenhol. Space efficient linear time computation of the Burrows and Wheeler transformation. In *Numbers, Information and Complexity*, pages 375–383. Kluwer Academic Publishers, 2000.

[KLV06] H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows–Wheeler based compression. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 282–293. Springer, 2006.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[KNM03] J. Kuri, G. Navarro, and L. Mé. Fast multipattern search algorithms for intrusion detection. *Fundamenta Informaticae*, 56(1–2):23–49, 2003.

[Knu73] D. E. Knuth. *The art of computer programming: Sorting and searching*, volume 3. Addison–Wesley, Reading, MA, 1973.

[Knu85] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, 1985.

[KNU03] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching on Ziv–Lempel compressed text. *Journal of Discrete Algorithms*, 1(3/4):313–338, 2003.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th Int. Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 943–955. Springer, 2003.

[KS05] S. T. Klein and D. Shapira. Pattern matching in Huffman encoded texts. *Information Processing and Management*, 41(4):829–841, 2005.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2676, pages 186–199. Springer, 2003.

[KST94] J.-Y. Kim and J. Shawe-Taylor. Fast string matching using an n-gram algorithm. *Software–Practice and Experience*, 24(1):79–88, 1994.

[KTSA99] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1645, pages 1–13. Springer, 1999.

[KU96a] J. Kärkkäinen and E. Ukkonen. Lempel–Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing*, pages 141–155. Carleton University Press, 1996.

[KU96b] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2ndAnnual International Conference on Computing and Combinatorics (COCOON)*, LNCS 1090, pages 219–230, 1996.

[Kuk92] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[Lec07] T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.

[LNP05] K. Lemström, G. Navarro, and Y. Pinzon. Practical algorithms for transposition-invariant string-matching. *Journal of Discrete Algorithms (JDA)*, 3(2–4):267–292, 2005.

[LS99] J. N. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.

[LU00] K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. of Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, pages 53–60, 2000.

[LV86] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43(2–3):239–249, 1986.

[LV89] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22:75–81, 1976.

[Mäk00] V. Mäkinen. Compact suffix array. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1848, pages 305–319. Springer, 2000.

[Mäk03a] V. Mäkinen. Compact suffix array — a space efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003. Special Issue - Computing Patterns in Strings.

[Mäk03b] V. Mäkinen. *Parameterized approximate string matching and local-similarity-based point-pattern matching*. PhD thesis, Department of Computer Science, University of Helsinki, August 2003.

[Man94] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 807, pages 113–124. Springer, 1994.

[Man01] G. Manzini. An analysis of the Burrows–Wheeler transform. *The Journal of the ACM*, 48(3):407–430, 2001. Prelim. version in SODA'99.

[Man04] M. A. Maniscalco. A solution for context based blocksort compression: The M03 algorithm. http://www.michael-maniscalco.com/papers/m03.pdf, 2004.

[Mas27] H. V. Masters. *A study of spelling errors.* Ph. D. Thesis, University of Iowa, 1927. Unpublished.

[MB95] A. Moffat and T. A. H. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 23(2):262–272, 1976.

[Meh84] K. Mehlhorn. *Data structures and algorithms 1: sorting and searching.* Springer, 1984.

[Mel96] B. Melichar. String matching with k differences by finite automata. In *Proc. 13th International Conference on Pattern Recognition*, volume II., pages 256–260. IEEE Computer Society Press, 1996.

[MF04] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[MHM$^+$01] S. Mitarai, M. Hirao, T. Matsumoto, A. Shinohara, M. Takeda, and S. Arikawa. Compressed pattern matching for SEQUITUR. In *Proc. Data Compression Conference (DCC)*, pages 469+, Snowbird, UT, 2001. IEEE Computer Society Press.

[Mil05] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. 16th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 11–12. SIAM, 2005.

[MM90] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proc. 1st ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 319–327. SIAM, 1990.

[MM91] G. Mehldau and G. Myers. A system for pattern matching applications on biosequences. *Computer Applications in the Biosciences*, 9(3):299–314, 1991.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MN04a] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 420–433. Springer, 2004.

[MN04b] V. Mäkinen and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical Report C-2004-20, Department of Computer Science, University of Helsinki, 2004.

[MN05a] M. G. Maaß and J. Nowak. Text indexing with errors. Technical Report TUM-I0503, Fakultät für Informatik, TU München, mar 2005.

[MN05b] M. G. Maaß and J. Nowak. Text indexing with errors. In *Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3537, pages 21–32. Springer, 2005.

[MN05c] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[MN07a] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226. Springer, 2007.

[MN07b] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[MN08] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008. 38 pages.

[MNF58] G. A. Miller, E. B. Newman, and E. A. Friedman. Length-frequency statistics for written English. *Information and Control*, 1:370–389, 1958.

[MNU03] V. Mäkinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. *Algorithmica*, 35:347–369, 2003.

[MNU05] V. Mäkinen, G. Navarro, and E. Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 56(2):124–153, 2005.

[MNZ97] E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *Proc. 4th South American Workshop on String Processing*, pages 95–111. Carleton University Press, 1997.

[MNZBY00] E. Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[Mof89] A. Moffat. Word-based text compression. *Software–Practice and Experience*, 19(2):185–198, 1989.

[MP80] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[MP08] M. A. Maniscalco and S. J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithmics (JEA)*, 12:1–23, 2008.

[MR97] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[Mun96] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, London, UK, 1996. Springer.

[MW94] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Winter 1994 Technical Conference*, pages 23–32, San Francisco, CA, 1994.

[Mye86] E. W. Myers. Incremental alignment algorithms and their applications. Report TR-90-25, Department of Computer Science, University of Arizona, Tucson, AZ, 1986.

[Mye96] E. W. Myers. Approximate matching of network expression with spacers. *Journal of Computational Biology*, 3(1):33–51, 1996.

[Mye98] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1448, pages 1–13. Springer, 1998.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *The Journal of the ACM*, 46(3):395–415, 1999.

[Nav98] G. Navarro. *Approximate Text Searching*. PhD thesis, Department of Computer Science, University of Chile, December 1998. ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/thesis98.ps.gz.

[Nav01a] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[Nav01b] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software–Practice and Experience*, 31:1265–1312, 2001.

[Nav04] G. Navarro. Indexing text using the Ziv–Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[NC06] G. Navarro and E. Chávez. A metric index for approximate string matching. *Theoretical Computer Science*, 352(1–3):266–279, 2006.

[NGMD05] G. Navarro, Sz. Grabowski, V. Mäkinen, and S. Deorowicz. Improved time and space complexities for transposition invariant string matching. Technical Report TR/DCC-2005-4, University of Chile, Department of Computer Science, 2005. ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/mnloglogs.ps.gz.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[NMN+00] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[NMW97] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3):103–116, 1997.

[NR98] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1448, pages 14–33. Springer, 1998.

[NR00] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, 5(4), 2000.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[NR03] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003.

[NST05] G. Navarro, E. Sutinen, and J. Tarhio. Indexing text with approximate q-grams. *Journal of Discrete Algorithms (JDA)*, 3(2–4):157–175, 2005.

[NT00] G. Navarro and J. Tarhio. Boyer–Moore string matching over Ziv–Lempel compressed text. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1848, pages 166–180, Montreal, Canada, 2000. Springer.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007.

[Pag99] R. Pagh. Low redundancy in static dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th Int. Colloquium on Automata, Languages*

and Programming (ICALP), LNCS 1644, pages 595–604. Springer, 1999.

[Păt08] M. Pătraşcu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008.

[Păt09] M. Pătraşcu. A lower bound for succinct rank queries. *CoRR*, abs/0907.1103, 2009.

[Pen03] Ch.-L. Peng. An approach for solving the constrained longest common subsequence problem. Master's thesis, Department of Computer Science and Engineering, National Sun Yat-sen University, Taiwan, 2003. http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/getfile?URN=etd-0828103-125439&filename=etd-0828103-125439.pdf.

[PGNS06] R. Przywarski, Sz. Grabowski, G. Navarro, and A. Salinger. FM-KZ: An even simpler alphabet-independent FM-Index. In *Proc. 11th Prague Stringology Conference (PSC)*, pages 226–240, 2006.

[PS80] W. Paul and J. Simon. Decision trees and random access machines. In *ZUERICH: Proc. Symp. Logik und Algorithmik*, pages 331–340, 1980.

[PT03] H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In *Proc. 10th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2857, pages 80–94. Springer, 2003.

[PW05] Y. J. Pinzón and S. Wang. Simple algorithm for pattern-matching with bounded gaps in genomic sequences. In *Proc. Int. Conference of Numerical Analysis and Applied Mathematics (ICNAAM)*, pages 827–831, 2005.

[Riv77] R. L. Rivest. On the worst case behavior of string searching algorithms. *SIAM Journal on Computing*, 6(4):669–674, 1977.

[RNO07] L. Russo, G. Navarro, and A. Oliveira. Approximate string matching with Lempel-Ziv compressed indexes. In *Proc. 14th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 265–275. Springer, 2007.

[RNO08a] L. Russo, G. Navarro, and A. Oliveira. Dynamic fully-compressed suffix trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 191–203, 2008.

[RNO08b] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. 8th Latin American Symposium (LATIN)*, LNCS 4957, pages 362–373, 2008.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 233–242. SIAM, 2002.

[RTT02] J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2373, pages 42–52. Springer, 2002.

[Rya80] B. Y. Ryabko. Data compression by means of a book stack. *Problems of Information Transmission*, 16(4):16–21, 1980.

[Ryt99] W. Rytter. Algorithms on compressed strings and arrays. In *Proc. 26th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 1725, pages 48–65. Springer, 1999.

[Sad00] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th Int. Symposium on Algorithms and Computation (ISAAC)*, LNCS 1969, pages 410–421. Springer, 2000.

[Sad02] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 225–232. SIAM, 2002.

[Sad07] K. Sadakane. Compressed suffix trees with full functionality. *Theoretical Computer Science*, 41(4):589–607, 2007.

[Sch61] C. Schensted. Largest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[Sew06] J. Seward. The bzip2 program. http://www.bzip.org/, 2006.

[SG06] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 1230–1239. ACM Press, 2006.

[SGD05] P. Skibiński, Sz. Grabowski, and S. Deorowicz. Revisiting dictionary-based compression. *Software–Practice and Experience*, 35(15):1455–1476, 2005.

[SGS06] J. Swacha, Sz. Grabowski, and P. Skibiński. Efektywna reprezentacja dokumentów XML (in Polish). In *Badania Operacyjne i Systemowe (BOS 2006), vol. 3*, pages 355–366, Szczecin, Poland, 2006. Akademicka Oficyna Wydawnicza EXIT.

[SGS07] P. Skibiński, Sz. Grabowski, and J. Swacha. Fast transform for effective XML compression. In *Proc. 9th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics*, pages 323–326, Polyana, Ukraine, 2007.

[SGS08] P. Skibiński, Sz. Grabowski, and J. Swacha. Effective asymmetric XML compression. *Software–Practice and Experience*, 38(10):1027–1047, 2008.

[Shk02] D. Shkarin. PPM: One step to practicality. In *Proc. Data Compression Conference (DCC)*, pages 202–211, Snowbird, UT, 2002. IEEE Computer Society Press.

[Sim93] I. Simon. String matching algorithms and automata. In *Proc. 1st South American Workshop on String Processing*, pages 151–157, Universidade Federal de Minas Gerais, Brazil, 1993.

[Smi91] P. D. Smith. Experiments with a very fast substring search algorithm. *Software–Practice and Experience*, 21(10):1065–1074, 1991.

[SS01] K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.

[SS07] P. Skibiński and J. Swacha. Combining efficient XML compression with query processing. In *Proc. 11th East European Conference on Advances in Databases and Information Systems (ADBIS)*, LNCS 4690, pages 330–342. Springer, 2007.

[SSG08] P. Skibiński, J. Swacha, and Sz. Grabowski. A highly efficient XML compression scheme for the Web. In *Proc. 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 4910, pages 766–777. Springer, 2008.

[ST07] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 71–83. SIAM, 2007.

[Ste92] G. A. Stephen. String search. Report TR-92-gas-01, University College of North Wales, 1992.

[STSA99] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1645, pages 37–49, Warwick University, UK, 1999. Springer.

[Sun90] D. M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[SZM03] W. Sun, N. Zhang, and A. Mukherjee. A dictionary-based multi-corpora text compression system. In *Proc. Data Compression Conference (DCC)*, page 448, Snowbird, UT, 2003. IEEE Computer Society Press.

[Tis08] A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008.

[TMK+02] M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukumachi, T. Shinohara, and S. Arikawa. Processing text files as is: Pattern matching over compressed tests. In *Proc. 9th Int. Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2476, pages 170–186. Springer, 2002.

[TP97] J. Tarhio and H. Peltola. String matching in the DNA alphabet. *Software–Practice and Experience*, 27(7):851–861, 1997.

[TS08] F. Transier and P. Sanders. Intersection in integer inverted indices. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 3–12. SIAM, 2008.

[Tsa03] Y. T. Tsai. The constrained common subsequence problem. *Information Processing Letters*, 88:173–176, 2003.

[TSM+01] M. Takeda, Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Speeding up string pattern matching by text compression: The dawn of a new era. *Transactions of Information Processing Society of Japan*, 42(3):370–384, 2001.

[Ukk85a] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985.

[Ukk85b] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1–3):132–137, 1985.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Vig08] S. Vigna. Broadword implementation of rank/select queries. In *Proc. 7th Workshop on Efficient and Experimental Algorithms (WEA)*, LNCS 5038, pages 154–168. Springer, 2008.

[Wan03] R. Wan. *Browsing and Searching Compressed Documents*. Ph. D. Thesis, University of Melbourne, Australia, 2003.

[WC76] C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *The Journal of the ACM*, 23(1):13–16, 1976.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[WF74] R. A. Wagner and M. Fischer. The string-to-string correction problem. *The Journal of the ACM*, 21(1):168–173, 1974.

[WM92a] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, 1992.

[WM92b] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[WM94] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[Yao79] A. C. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979.

[YC08] I.-H. Yang and Y.-C. Chen. Fast algorithms for the constrained longest increasing subsequence problems. In *Proc. 25th Workshop on Combinatorial Mathematics and Computation Theory*, pages 226–231, 2008.

[Zec72] E. Zeckendorf. Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres Lucas. *Bulletin de la Société Royale des Sciences de Liège*, 41:179–182, 1972.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

List of Symbols and Abbreviations

Abbreviation	Description
T	text
P	pattern
n, m	text length, pattern length
Σ	alphabet
σ	alphabet size
occ	number of pattern occurrences
w	machine word size, in bits
$\log n$	binary logarithm of n
r	number of patterns in multiple pattern matching
D	binary state vector (in bit-parallel algorithms, e.g. Shift-Or)
B	preprocessing table with σ bit-vectors (in bit-parallel algorithms, e.g. Shift-Or)
$<<$, $>>$	bit-wise shift to left, shift to right, both with zero padding
$\&$, $\vert$, $\sim$, $\hat{}$	bit-wise `and`, `or`, `not`, `xor`
KMP	Knuth–Morris–Pratt exact matching algorithm
BMH	Boyer–Moore–Horspool exact matching algorithm
NFA	non-deterministic finite automaton
BNDM	backward non-deterministic DAWG matching
AC	Aho–Corasick multiple matching algorithm
AOSO	average-optimal Shift-Or algorithm
FAOSO	fast variant of the AOSO algorithm
$ed(A, B)$	edit (Levenshtein) distance between sequences A and B
$d_{\mathrm{ID}}(A, B)$	indel distance between sequences A and B
$d_{\mathrm{H}}(A, B)$	Hamming distance between sequences A and B
$id(A, B)$	the minimum number of symbols inserted to sequence A to transform it into sequence B
LCS	longest common subsequence problem
LCTS	longest common transposition-invariant subsequence problem

Abbreviation	Description
R	number of all matching character pairs in LCS and related algorithms
ℓ	length of the output sequence in LCS and related algorithms
MB	molecular biology
MIR	music information retrieval
α	maximum allowed gap size
δ	maximum allowed error difference on individual symbol
γ	maximum allowed sum of symbol differences for the whole pattern
ETDC	end-tagged dense code
(s,c)-DC	dense code with parameters: stoppers (s), continuers (c)
Dict	dictionary of q-grams
BWT	Burrows–Wheeler transform
PPM	prediction by partial matching compression algorithm
M	conceptual matrix of sorted suffixes of the text
T^{bwt}	output of the Burrows–Wheeler transform applied to text T
$Occ(X,c,i)$	number of occurrences of symbol c in the prefix $X[1\dots i]$
C	array of size σ storing the counts of occurrences of alphabet symbols in text T
$rank(V,b,i)$	number of occurrences of binary symbol b in binary sequence $V[1\dots i]$
$select(V,b,i)$	index j such that $rank(V,b,i)=j$ for a binary sequence V
$selectNext(V,b,i)$	index j of the next occurrence of binary symbol b in binary sequence $V[i\dots n]$
$H_k(T)$	order-k empirical entropy of text T
T'	Huffman-compressed T (binary stream)
B	$(T')^{\mathrm{bwt}}$ (only in Chap. 5)
n'	size of the array $B=(T')^{\mathrm{bwt}}$, in bits
Bh	auxiliary bit array of size n' for signalling codeword boundaries in T'
$\mathcal{A}'$	suffix array for T'
TS, ST, S	auxiliary arrays for locate and display in the FM-Huffman index
l, r	boundaries of the text area to extract in display queries
L	length of the text excerpt in display queries; $L=r-l+1$

List of Figures

1.1 BMH example . . . 15
1.2 Shift-Or example . . . 20
1.3 AOSO example . . . 23
1.4 AC-automaton example . . . 30

2.1 Dynamic programming for Levenshtein distance calculation . . . 46
2.2 An NFA for recognizing the pattern "ROSES" with at most two Levenshtein errors . . . 51
2.3 LCTS, % matches vs. % transpositions (transpositions sorted by frequency) . . . 72
2.4 LCTS, overall processing time of the hybrid algorithm with varying threshold of the minimal number of matches in transpositions handled by the HS component. . . . 73
2.5 LCTS, overall processing time of the hybrid algorithm with varying threshold of the total percentage of matches handled by the HS component. . . . 73

3.1 Row-wise SPD for (δ, α)-matching, $O(\sqrt{\delta}n)$ time preprocessing . . . 89
3.2 (δ, α)-matching. Tiling the dynamic programming matrix with $w \times 1$ vectors ($w = 8$). . . . 102
3.3 (δ, γ, α)-matching. Tiling the dynamic programming matrix with $C = \lfloor w/(\ell+1) \rfloor \times 1$ vectors ($C = 8$). . . . 106
3.4 A building block for a systolic array detecting δ-matches with α-bounded gaps. . . . 113
3.5 Running times for (δ, α)-matching, in seconds . . . 126
3.6 Running times for transposition invariant (δ, α)-matching, in seconds . 127
3.7 Running times for (δ, γ, α)-matching, in seconds . . . 129

5.1 CSA example. T = tete-a-tete$ . . . 164
5.2 BWT example. T = BABOON$ . . . 165
5.3 Comparison of different methods to solve *rank* . . . 176

5.4 (Left) Comparison of different methods to solve *select* by binary search. (Right) comparison of different space overheads for *select* based on binary search . . . 177
5.5 Comparison of Clark's *select* on different densities and two binary search based implementations using different space overheads . . . 179
5.6 (Left) Search time as a function of the pattern length. (Right) Average search time per character as a function of the size of the index. . . . 199
5.7 Time to report the positions of the occurrences as a function of the size of the index . . . 200
5.8 (Left) Time to show the first character of a text context around the positions of the occurrences as a function of the size of the index. (Right) Time per character displayed around an occurrence and space for each index. . . . 201

List of Tables

1.1 Exact matching. Searching speed in megabytes per second for different algorithms on Pentium4. 35
1.2 Exact matching. Searching speed in megabytes per second for different algorithms on UltraSPARC IIIi. Left: DNA; right: natural language. . 35
1.3 Searching speed in megabytes per second for different algorithms on Core 2 Duo 37
1.4 k-mismatches. Searching speed in megabytes per second for Average-Optimal Shift-Add on Core 2 Duo. 38
1.5 Multiple pattern search. Searching speed in megabytes per second for different algorithms on Intel Core 2 Duo. 39

2.1 LCTS, MUSIC, 32-bit implementation 74
2.2 LCTS, MUSIC, 64-bit implementation 74
2.3 LCTS, RANDOM-128, 64-bit implementation 74
2.4 LCTS, GAUSS-128, 64-bit implementation 75

4.1 Comparison of compression ratios 148
4.2 Dictionary sizes and the numbers of unique q-grams for various files . . 148
4.3 The effect of varying q on the dictionary size and the overall compression (Dickens/ETDC) 148
4.4 Comparison of decompression times 149
4.5 Search times in seconds for short and long patterns 150
4.6 Denser encoding. Comparison of compression ratios in word based schemes. 152
4.7 Denser encoding. Comparison of compression ratios in q-gram based schemes. 153

5.1 (Top) Space requirement of FM-Huffman for different values of k. (Bottom) Detailed comparison of $k = 2$ versus $k = 4$. 197

SUMMARY

The presented dissertation focuses on various exact and approximate matching problems for textual data. Text should be understood rather broadly, including natural language, molecular biology and music information retrieval data.

The work consists of five chapters, each dedicated to a separate problem. In the order of presentation, they deal with exact string matching, approximate string matching, matching with gaps (which could be considered a subclass of approximate string matching problems but the amount of contained material should justify a separate chapter), online compressed search and compressed full-text indexes. The author contributed to each of those research areas. Each chapter, however, starts with background presentation to place the author's achievements in proper context.

Many of the algorithms proposed in the dissertation are based on bit-parallelism, i.e., a modern technique of making use of individual bits in a CPU word. In particular, two new bit-parallel techniques have been presented, one for efficient matching in the average case, the other to reduce time complexities in the worst case of bit-parallel algorithms making use on counters. Those are rather general techniques and they have been successfully applied for multiple known string matching problems.

It has been shown that the problems of matching with gaps can be attacked from very different angles, and the arsenal of existing techniques in this area has been significantly expanded. The new results comprise the algorithmic techniques of bit-parallelism, sparse dynamic programming, compact bit-parallel NFA simulations and filtering.

New algorithms are also presented for full-text searching in compressed data; they are both simple and efficient.

Apart from theoretical analyses, most of the proposed algorithms have been experimentally validated on modern hardware and the achieved results usually place them among very competitive ones.

Charakterystyka zawodowa autora

Szymon Grabowski urodził się w roku 1973 w Kole. W 1996 r. ukończył studia na Wydziale Matematyki, Fizyki i Chemii Uniwersytetu Łódzkiego i uzyskał tytuł magistra informatyka. Następnym etapem jego kariery zawodowej były studia doktoranckie na Wydziale Elektrotechniki i Elektroniki Politechniki Łódzkiej, a także praca na stanowisku asystenta w Katedrze Informatyki Stosowanej. Tematyka badawcza jego pracy doktorskiej dotyczyła rozpoznawania obrazów, a ściślej klasyfikatorów typu najbliższy sąsiad. W roku 2003 Sz. Grabowski obronił z wyróżnieniem na Akademii Górniczo-Hutniczej w Krakowie pracę doktorską nt. „Konstrukcja klasyfikatorów minimalnoodległościowych o strukturze sieciowej". W grudniu 2003 r. został powołany na stanowisko adiunkta w Katedrze Informatyki Stosowanej PŁ.

Po obronie doktoratu autor rozprawy zmienił swoją podstawową tematykę badawczą, koncentrując się na algorytmach tekstowych, często z wykorzystaniem kompresji danych i technik tzw. równoległości bitowej. Konkretne problemy, którymi się zajmował, dotyczą m. in. wyszukiwania przybliżonego, skompresowanych indeksów tekstowych i kompresji XML.

Szymon Grabowski jest autorem lub współautorem około 80 artykułów naukowych, w tym 14 z tzw. listy filadelfijskiej. Na konferencji międzynarodowej *String Processing and Information Retrieval (SPIRE)* w roku 2006 otrzymał nagrodę za najlepszy artykuł (wraz ze współautorem, Kimmo Fredrikssonem). Uczestniczył w dwóch projektach badawczych (NATO i MNiSW), w ramach pierwszego z nich odbył w roku 2001 krótki staż zagraniczny w USA. Od sierpnia 2008 r. jest członkiem kolegium redakcyjnego pisma International Journal of Computer Mathematics. Był recenzentem pracy doktorskiej Jana Lánský'ego z Uniwersytetu Karola w Pradze.

Działalność dydaktyczna autora rozprawy obejmuje algorytmikę, rozpoznawanie obrazów i przedmioty związane z programowaniem komputerów.