7
Conclusion

Before finishing, we would like to give some extra material that might be of interest.

First, we believe that it is extremely useful to know of freely available tools for on-line text searching, so we cover the existing software of this kind we are aware of.

Second, we give pointers to other books, journals, conferences, and on-line resources one may want to read to enter deeper into the area of text searching. This is also of interest to readers with a specific algorithmic problem not addressed in this book and not solved by the available software.

Finally, we include a section with problems related to combinatorial pattern matching. The section aims at briefing over the different extensions to the basic text searching problem, explaining the main concepts and existing results, and pointing to more comprehensive material covering them.

Up to date information and errata related to this book will be available at `http://www.dcc.uchile.cl/~gnavarro/FPMbook`.

7.1 Available software

We present in this section a sample of freely available software for on-line pattern matching.

7.1.1 Gnu Grep

What it is GNU (`http://www.gnu.org`) is an organization devoted to the development of free software. One of its products, *Grep*, permits fast searching of simple strings, multiple strings, and regular expressions in a set of files. Approximate searching is not supported. *Gnu Grep* is twice as fast as the classical Unix *Grep*.

Grep reports the lines in the file that contain matches. However, there are many configuration options that permit reporting the lines that do not match, the number of lines that match, whole files containing matches, and so on. The software provides a very powerful syntax that includes operators that go beyond regular expressions.

How it works Simple strings are searched with a **Boyer-Moore-Gosper** search algorithm (similar to **Horspool**; see Section 2.3.2). Sets of patterns are searched using a **Commentz-Walter**–like algorithm (Section 3.3.1). Regular expressions are searched with a lazy deterministic automaton, that is, a DFA (Section 5.3.2) whose states and transitions are built as they are reached while scanning the text using forward scanning. To speed up the search of complex patterns, *Grep* tries to extract their longest necessary factors, which are used as a filter and searched as a set of strings. This technique is explained in Section 5.5.2. It permits *Grep* to decline smoothly in performance as the complexity of the search increases, obtaining in general excellent performance.

Where to get it The current stable version of *Gnu Grep* is 2.4.2 (March 2000). Its source code distribution, in **C** language, can be obtained and used for free from `http://www.gnu.org/software/grep/grep.html`, as well as from `ftp://ftp.gnu.org/pub/gnu/grep/`.

7.1.2 Wu and Manber's Agrep

What it is *Agrep* (for *a*pproximate *Grep*) was developed in 1992 by Sun Wu and Udi Manber at the University of Arizona, as the first of a series of tools for on-line and indexed searching that include *Glimpse*, *WebGlimpse* (`http://glimpse.cs.arizona.edu/`), and *Harvest* (`http://www.tardis.ed.ac.uk/harvest/`).

Agrep is an on-line pattern matching software capable of exact and approximate searching for simple strings, extended strings, and regular expressions, as well as exact searching for multiple strings. *Agrep* has a syntax and a set of options similar to *Grep*, albeit less powerful. Extended strings include wild cards and classes of characters. Other extensions are treated as regular expressions. The real novelty of *Agrep* with respect to *Grep* is its approximate searching ability. Also, it has more flexible reporting: Instead of just lines, a "record" delimiter can be defined to report matching records (e.g., whole e-mails in an e-mail archive).

How it works The algorithmic principles of *Agrep* have been described in [WM92b], and the software itself in [WM92a]. It does not use a uniform algorithm but a set of heuristics to deal with the different search problems. As a result, *Agrep* normally chooses the best algorithm, but it experiences sharp changes in its efficiency as a result of slight changes in the complexity of the search patterns. Moreover, there are many restrictions to the length of the patterns and to the combination of options permitted. Despite these shortcomings, *Agrep* is very fast in some very commonly used cases.

Simple strings are searched with a variant of the **Horspool** algorithm (Section 2.3.2) when their length does not exceed 400. Longer strings use a similar technique, but pairs of characters, instead of single characters, are used for building the shift table. Sets of strings are searched with the **Wu-Manber** algorithm described in Section 3.3.3.

For the rest of the patterns, *Agrep* relies on bit-parallelism, more specifically on extensions of **Shift-And** (Section 2.2.2). Classes of characters and wild cards are handled with the techniques described in Chapter 4 to extend **Shift-And**. Similarly, regular expressions are handled with the bit-parallel algorithm **BPThompson** (Section 5.4.1).

Finally, approximate searching is handled in two ways. For simple strings searched with low error levels, *Agrep* uses **PEX** (Section 6.5.1). The other cases are handled using the bit-parallel algorithm **BPR** (Section 6.4.1.1) and its extension to regular expressions (Section 6.7.3).

Where to get it Commercial use requires paying a fee, but *Agrep* can be used for free for academic purposes and by U.S. government organizations. The code is available in source form (**C** language).

Older versions of *Agrep* can be obtained from `ftp://ftp.cs.arizona.edu`. The latest version is 3.0, from 1994, and it can be obtained by downloading *Glimpse* (any version after 1994) from `http://webglimpse.net/download.html`. Look for a top-level subdirectory called `"agrep/"`.

A Windows version of *Agrep* can be obtained from `http://www.tgries.de/agrep`.

7.1.3 Navarro's Nrgrep

What it is *Nrgrep* (for *n*ondeterministic *r*everse *Grep*) is an on-line pattern matching software developed in 2000 by Gonzalo Navarro at the University of Chile. Functionality is similar to that of *Agrep*. Multiple string matching, however, is not supported by *Nrgrep*.

How it works *Nrgrep* is based entirely on the **BNDM** algorithm and its extensions, presented in [NR00, NR99a, NR01a]. A description of the software is given in [Nav01b]. The fact that it is built on a single technique means that its efficiency degrades smoothly with the complexity of the search problem, unlike *Agrep*. The software demonstrates the flexibility of the **BNDM** approach, and it is very fast when searching complex patterns and regular expressions, exactly or allowing errors.

Single strings are searched with the basic **BNDM** algorithm of Section 2.4.2. The software supports extended strings, in particular, classes of characters and optional and repeatable characters, extending **BNDM** as shown in Chapter 4. Regular expressions are searched using **Regular-BNDM** (Section 5.5.3).

Approximate searching is handled, as in *Agrep*, in two possible ways. First, **PEX** can be used as in Section 6.5.1, and the pieces searched using **Multiple BNDM** (Section 3.4.1). Second, **ABNDM** can be used (Section 6.5.2). This permits skipping characters and using the technique not only for simple strings but also for extended strings and regular expressions (Section 6.7.3).

A fact that contributes to the smoothness of the efficiency of *Nrgrep* as a function of the pattern complexity is that it automatically selects the best factor of the pattern for the purpose of filtering the search, and also detects the correct type of pattern regardless of its syntax, in order to apply the simplest possible search algorithm. Finally, if the search cost with **BNDM** is predicted to be too high, it switches to forward scanning (**Shift-And**).

Where to get it *Nrgrep* source code, in **C** language, can be freely downloaded from `http://www.dcc.uchile.cl/~gnavarro/pubcode`. The code is version 1.1 (2001).

7.1.4 Mehldau and Myers' Anrep

What it is *Anrep* was built by Gerhard Mehldau and Gene Myers at the University of Arizona in 1993. It is an interactive application for DNA and protein searching, finding exact and approximate matches of patterns ranging from simple strings to network expressions and spacers (Sections 6.7 and 4.3). This includes most patterns of interest in biosequence comparisons. The user specifies such patterns with a declarative, free-format, and strongly typed language called *A*.

How it works *Anrep* is described in [MM91] and its algorithmic principles can be found in [Mye96]. The language is very powerful, as is needed in biological applications, so simple algorithms cannot be used. *Anrep* is based on the algorithm mentioned in Section 6.7.2 for approximate searching with arbitrary costs of network expressions with spacers. This combines dynamic programming for matching network expressions allowing errors and an optimized backtracking procedure to determine which occurrences are at the correct distances from the others.

There is little point in comparing *Anrep* with the previous programs. *Anrep* is much slower because it can search for much more complex patterns.

Where to get it The **C** language source code of *Anrep* can be freely obtained at `http://www.cs.arizona.edu/people/gene/CODE/anrep.tar.Z`.

7.1.5 Other resources for computational biology

Apart from *Anrep*, there are lots of resources available for computational biology applications. We do not cover them all in detail because they focus less on string matching than on statistical problems related to determining relevant subsequences, and use very specific knowledge from computational biology. The algorithms are generally complex variants of approximate searching, with complicated cost functions, gap penalties, and so on. The searching is done with a combination of dynamic programming and filtering approaches (Chapter 6), plus heuristics for handling the gaps. We brielfly review two of the best known systems of this type.

BLAST is an acronym for *Basic Local Alignment Search Tool.* It was created in 1990 by Altschul et al. It consists of a set of similarity search programs for exploring sequence databases for protein or DNA queries. Its main aims are high speed with minimal sacrifice of sensitivity to detect interesting occurrences and a well-defined statistical interpretation of the matches reported. *BLAST* uses a heuristic algorithm that seeks local as opposed to global alignments and is therefore able to detect relationships among sequences that share only isolated regions of similarity. Its algorithmic principles are presented in [AGM+90]. Software executables for different architectures can be freely obtained from `http://www.ncbi.nlm.nih.gov/BLAST/`, where it is also possible to test the system on-line. Its current version is 2.0 (1997).

FASTA was created by Pearson and Lipman in 1988. *FASTA* is another system for searching sequence homology in biosequence databanks, finding optimal local alignment scores. It includes several programs that provide different speed/accuracy trade-offs. *FASTA* has similar aims as *BLAST*, their main differences being in the way they assign significance to the matches [Pea91]. The algorithmic principles behind *FASTA* are presented in [SW81, PL88]. The system sources can be freely obtained from `ftp://ftp.virginia.edu/pub/fasta/`. Its current version is 3.2 (1998).

7.2 Other books

7.2.1 Books on string matching

We present here all the books we are aware of that attempt to cover a reasonably wide area of string matching.

Handbook of algorithms and data structures by G. Gonnet and R. Baeza-Yates, Addison-Wesley, second edition, 1991

This book deals with algorithms in general, but it includes a chapter devoted to exact string matching. The book is organized as a set of recipes. For each algorithm it gives a short explanation of the main idea and then the code and analytical results.

The book is a good reference for somebody in a hurry to find an algorithm to solve a string matching problem, since one can look at the analysis and copy the code. But it probably is not enough for learning why and how an algorithm works. The other problem is that it lacks developments since 1992, as well as approximate search algorithms.

Indeed, there are many books on algorithms that devote one chapter to string matching, for example, [Knu73, AHU83, Meh84, Baa88, Sed88, Man89, CLR90], but in general they cover only **KMP** and **BM**. We chose this book because, among those dealing with general algorithms, it has the best coverage. Some books on compilers or formal languages, such as [ASU86, HU79], explain the classical DFA approach to regular expression searching.

Text algorithms by M. Crochemore and W. Rytter, Oxford University Press, 1994

This book is a good survey of the main techniques used in text searching algorithms. The focus of the book is definitely theoretical; for example, it

does not present any bit-parallel algorithms, and it presents many algorithms that we have omitted in this book because they are inefficient in practice. The book is mainly devoted to exact searching.

This book is a good choice for those interested in the theoretical and combinatorial aspects underlying string matching algorithms, but it is definitely not recommended if one needs a practical string matching algorithm and does not want to enter so deep into the field.

String searching algorithms by G. Stephen, World Scientific Press, 1994

This is a fairly complete book on exact and approximate string matching. For exact string matching, it covers more than the usual algorithms, paying special attention to the Boyer-Moore family. Yet it lacks coverage of the BDM family and of bit-parallel algorithms. Multiple and extended string matching are not covered. The coverage of approximate string matching algorithms is quite good, with a long chapter devoted to the different string similarity measures and another chapter with a very complete survey (for 1994) of approximate string matching algorithms. This particular area, however, has evolved a lot since then, so the fastest algorithms today are missing. The book also covers some data structures for indexed text searching, such as suffix trees.

String pattern matching strategies by J. Aoe (Editor), IEEE Computer Science Press, 1994

This book covers the most basic string searching algorithms for single, multiple, approximate, and multidimensional string matching. It lacks coverage of the newer algorithms, which are the fastest.

Pattern matching algorithms by A. Apostolico and Z. Galil (Editors), Oxford University Press, 1997

This book is a collection of chapters written by several researchers. The chapters are well chosen to cover a wide range of issues from on-line exact and approximate pattern matching, to parallel and indexed searching of strings, trees, and matrices. It is highly theoretical, and the same recommendations as for *Text algorithms* apply.

Modern information retrieval by R. Baeza-Yates and B. Ribeiro-Neto, Addison-Wesley, 1999

This book is mainly on information retrieval, but it is one of the few that pays attention to the algorithmic problems involved, and even includes a chapter devoted to on-line string matching. The chapter is intended to give a reader interested in information retrieval some insight into the string matching problems that lie behind, but it is not enough to solve a string matching problem.

7.2.2 Books on computational biology

A book that lies at the intersection of string matching and computational biology is the following.

Algorithms on strings, trees and sequences: Computer science and computational biology by D. Gusfield, Cambridge University Press, 1997

This book is a survey of the main algorithmic techniques used in computational biology when using data structures like sequences and trees, which actually represent a large part of the field. It gives a complete general view of these techniques, including a large section on indexing, and in particular on the suffix tree and the algorithms built on it.

There are many other books on computational biology that are less related to string matching, so we have chosen three that we consider representative.

Computational molecular biology: An algorithmic approach by P. A. Pevzner, MIT Press, 2000

This recent book presents the main topics in computational molecular biology that involve algorithmic developments. This includes computational gene hunting, restriction mapping, map assembly, sequencing, DNA arrays, sequence comparison, multiple alignment, finding signals in DNA, gene restrictions, genome rearrangements, and computational proteomics.

Introduction to computational biology by M. S. Waterman, Chapman & Hall, 1995

This book presents well-established topics in computational biology on

which much research has been performed. Many of those results are now considered as "classical."

Time warps, string edits, and macromolecules: The theory and practice of sequence comparison by D. Sankoff and J. B. Kruskal, Addison-Wesley, 1983

This was one of the first books published in computational biology. It is a collection of texts on different topics, most of them presenting a precise problem in computational biology and the algorithms to solve it. The algorithmic solutions presented are generally too old now to be of real interest, but the problems they solve are still of interest and their presentation is usually clear.

7.3 Other resources

7.3.1 Journals

Articles on pattern matching tend to appear sparsely in different journals. The most commonly chosen are; for tutorials: *ACM Computing Surveys*; for algorithms: *Algorithmica*, *Journal of the ACM*, *Journal of Algorithms*, *Communications of the ACM* (but not recently), *Information and Computation*, *Information and Control*, *Information Processing Letters*, *Information Science*, *Journal of Computer Systems Science*, *Nordic Journal of Computing*, *Random Structures and Algorithms*, *SIAM Journal on Computing*, *Theoretical Computer Science*, and the new *Journal of Discrete Algorithms*; for implementations: *Software Practice & Experience*, *IEEE Trans. on Software Engineering*, *Information Processing and Management*, and *ACM Journal of Experimental Algorithmics*. In this list we have not considered articles on combinatorial pattern matching, which is a wide area with deep theoretical roots. Indeed, string matching is one of the simplest branches of combinatorial pattern matching.

We also mention a few of the many journals on computational biology: *Bioinformatics* (and its former version, *CABIOS*), *Nucleic Acids Research*, *Journal of Computational Biology*, *Genome Research*, and *Journal of Molecular Biology*.

7.3.2 Conferences

There are a few conferences devoted to the field. Among the best are *Combinatorial Pattern Matching (CPM)*, *Computing and Combinatorics*

(COCOON), *ACM Computational Molecular Biology (RECOMB)*, *String Processing and Information Retrieval (SPIRE)*, and *Intelligent Systems for Molecular Biology (ISMB)*.

Other conferences that publish articles on pattern matching are *Data Compression Conference (DCC)*, *European Symposium on Algorithms (ESA)*, *IEEE Foundations on Computer Science (FOCS)*, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, *Automata, Languages and Programming (ICALP)*, *IFIP World Computer Congress*, *Algorithms and Computation (ISAAC)*, *Mathematical Foundations of Computer Science (MFCS)*, *Discrete Algorithms (SODA)*, *Theoretical Aspects of Computer Science (STACS)*, *ACM Theory of Computing (STOC)*, *Scandinavian Workshop on Algorithmic Theory (SWAT)*, *Workshop on Algorithm Engineering (WAE)*, and *Workshop on Algorithms and Data Structures (WADS)*.

7.3.3 On-line resources

Definitely one of the best Web pages on string matching is *Pattern Matching Pointers*, an invaluable directory for searching people, references, books, software, journals, news groups, and discussion boards related to pattern matching in general. The page is maintained by Stefano Lonardi at `http://www.cs.purdue.edu/homes/stelo/pattern.html`.

Other pages are *Pattern Matching and Data Mining Research*, maintained by Mika Klemettinen at `http://www.cs.helsinki.fi/research/pmdm`, and *The Bioinformatics Resources* at `http://hgmp.mrc.ac.uk/CCP11`. Some discussion boards on the subject are available at `http://www.purdue.cs.edu/homes/stelo/pmdb`. Related news groups are `comp.theory`, `comp.theory.info-retrieval`, `comp.text`, and `comp.infosystems`. Finally, relevant mailing lists are `theorynet`, `dbworld` and `dmanet`.

Beware that on-line references may change over time.

7.4 Related topics

We consider finally some topics related to the focus of our book. Any of the related topics cited below could be the subject of an entire volume. We give the main current references for each subject. This list is obviously not exhaustive.

7.4.1 Indexing

Our book is devoted to *on-line* searching in text and sequences, which means that we do not build any structure on the text. For a single search this is an optimal strategy, but for many search operations on the same text we can save time by first building a structure on the text, called an *index*, to speed up queries later.

Typical reasons for preferring on-line searching are (1) size of the text, that is, if the text is too small, an index is not worth maintaining; (2) volatility of the text with respect to the query frequency, that is, there is a cost to build and maintain the index, which to be amortized requires that changes to the text be much less frequent than queries made on it; (3) space unavailability, that is, an index needs extra space on top of the text, which may be too costly or not available. Even when indexes are used, on-line searching is of interest because many indexing techniques use some form of on-line searching inside.

7.4.1.1 General indexes

Indexes permit exact searching of a string of length m in a text of length n in $O(m)$ or $O(m \log n)$ time, after a construction that usually takes $O(n)$ but sometimes $O(n \log n)$ time, and $O(n)$ extra space, with a constant factor that may range from 2 to 30 times the text size. There exist many indexing structures depending on the type of search and the memory available. The most usual ones build on the concept of a *suffix trie* [AG85], which is a trie data structure (Chapter 3) built over all the suffixes of the text. Every text factor is found by descending in this trie following the characters of the pattern.

The most efficient data structures are compacted versions of the suffix trie: the compact suffix tree [AG85, Gus97]; the suffix automaton or DAWG (an automaton that recognizes all text suffixes [CR94]); the compact suffix automaton or CDAWG [CV97a, CV97b, IHS+01]; and the suffix array, an array storing all text suffixes in lexicographical order [MM93, GBYS92].

These structures can also be used for searching extended strings, regular expressions, and for approximate searching [MBY91, BYG96, NBY00]. The search time is either $O(mn^{\lambda})$ with $0 \leq \lambda \leq 1$ or exponential in m (or k for approximate searching with k errors), sometimes multiplied by an extra $O(\log n)$ factor, depending on the data structure.

7.4.1.2 Indexes for natural language

When it comes to natural language texts, a very popular index is the inverted file or inverted index, which is normally just able to retrieve complete words

and phrases of the text, not any factor. Inverted indexes consist in general of the set of different words of the text (the vocabulary) and for each such word the list of positions where it appears in the text. This structure is also useful for information retrieval, which involves pattern matching but also concepts such as computing the relevance of a document with respect to a query. Some books dealing extensively with inverted indexes are [WMB99, BYRN99].

There are many variants on this structure, but if we consider the problem of finding words, the most important issue is the *addressing granularity* of the index. The indexes with the finest granularity store the exact positions of each word, and they need about 30% extra space over the text size. Others divide the text collection into documents and point just to the documents where each word appears, needing about 15% extra space. An interesting implementation capable of producing an index as small as 2%–4% over the text size is *Glimpse* [MW94] (`http://glimpse.cs.arizona.edu/`), which divides the text into equal size blocks and points to blocks instead of exact positions. This is called "block addressing." The search on these indexes has to be complemented with sequential searching. An analysis in [BYN00a] shows that the index size can be made sublinear with respect to the text size while keeping the search time sublinear as well. *Glimpse* also introduces techniques for searching extended strings, regular expressions, and approximate searching at the intraword level by scanning the vocabulary of the text, which is of sublinear size.

7.4.2 Searching compressed text

The problem of searching compressed text is that of finding the occurrences of a pattern in a compressed text without decompressing it. The subject has been an active area of research since 1992, motivated by the fact that CPU speed increases much faster than the speed of I/O devices and by the discovery that in some cases it is possible to search the compressed text *faster* than the uncompressed one.

7.4.2.1 Compression algorithms

Compression is a large and active area in computer science and of course we do not attempt to cover it here. Text compression is a subfield that deals with the best algorithms to compress text files. A good book on text compression is [BCW90]. More focused than text compression is the field of compressed text databases, which aims at text compression techniques that permit efficient searching of the compressed text. We briefly cover this area.

Compression formats for text databases must permit efficient decompres-

sion and random access to the text. Some of the most popular text compression formats for compressed text databases are Huffman [Huf51] (where each text symbol is replaced by a variable length code, trying to assign shorter codes to more frequent symbols), Ziv-Lempel (where the coder replaces text strings by pointers to previous occurrences already found in the text, without restriction in the LZ77 variant [ZL77] and only permitting a previous repetition plus an extra letter in the LZ78 variant [ZL78]), and Byte-Pair encoding or BPE [Gag94] (where pairs of characters are joined under a new unused code iteratively until no unused codes remain). An important feature of a compression method is the *compression ratio* achieved, which we define as the ratio of the compressed file size to the uncompressed file size. For example, on DNA, Huffman obtains about 25% compression, Ziv-Lempel 25%–30%, and BPE 30%, while on typical natural language text Huffman obtains about 60%, Ziv-Lempel 30%–40%, and BPE 70%.

7.4.2.2 On-line pattern matching in compressed text

The *compressed matching problem* was first defined in the work of Amir and Benson [AB92a] as the task of performing string matching in a compressed text without decompressing it. Given a text $T = t_1 \dots t_u$, a corresponding compressed string $Z = z_1 \dots z_n$, and a pattern $P = p_1 \dots p_m$, the compressed matching problem consists in finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(m + u)$. An optimal algorithm takes worst-case time $O(m + n)$.

The most practical methods for on-line pattern matching are based on the BPE algorithm and its variants [Man97, SMT+00, TSM+01]. They are able to search the compressed text faster than the original text. This may be a reason by itself to compress the text. Given the characteristics of the format, however, the compression ratio obtained is poor.

A large line of research is based on Ziv-Lempel compression, which obtains much better compression ratios. The first algorithm for exact searching was [ABF96]. They search LZ78 compressed text in $O(m^2 + n)$ time and space. One of the few techniques for the LZ77 format is [FT98], a randomized algorithm to determine in $O(m + n \log^2(u/n))$ time whether a pattern is present or not in the text.

Later practical improvements appeared in [NR99b, KTSA99, NT00]. Roughly speaking, it is possible to search the compressed text in about half the time necessary for decompressing and then searching it.

Some extensions of the search problem have been pursued for the Ziv-Lempel format. An extension of [ABF96] to multipattern searching was

presented in [KTS+98], where they achieved $O(m^2 + n)$ time and space, where m is the total length of all the patterns. Approximate string matching on compressed text was an open problem advocated in [AB92b], and the first theoretical [KNU00, MKT+00] and practical [NKT+01] algorithms for handling it have appeared only recently. All are for the LZ78 format. Regular expression searching over LZ78 was considered in [Nav01c].

A useful search-oriented abstraction of the compression formats used for pattern matching, called "collage systems," was proposed in [KTSA99]. Algorithms designed for collage systems can be implemented for many different formats. In the same paper they design a **KMP** algorithm on collage systems. Later papers of the same group (referenced in the previous paragraphs) develop this concept.

Finally, it is interesting to mention that there are very efficient algorithms able to find complete words and phrases in natural language texts. In [MNZBY00], a word-oriented Huffman coding where the symbols are the text words and separators, not the characters, is used as the basis for very fast algorithms that are able of exact and approximate searching for simple and extended strings. The compression ratio is very good, about 25%–30% on English texts of at least 10 megabytes, and the search on compressed text is as fast as on uncompressed text for simple searching, while it is up to eight times faster when searching for complex patterns and for approximate searching.

7.4.2.3 Indexed pattern matching in compressed text

Building compressed indexes over compressed text is a natural goal on large text databases. Compressed data structures for text searching have been sought for some time [KU96, Kär99, KS98, GV00], but they always used the text in uncompressed form as an integral part of the data structure.

Recently, some very promising structures have appeared which compress the text together with the structure. These data structures are compressed versions of the suffix array [Sad99, Sad00, FM00, FM01], and in some cases they are able to represent index and text in less space than that of the original uncompressed text.

For inverted indexes for word retrieval in natural language text, the text and the index are compressed separately in general. Index compression takes advantage of the fact that the list of occurrences of each word is increasing. Differences are encoded with a coding method that favors small numbers. The larger the addressing granularity, the more effective the compression of the lists of occurrences. A system combining block addressing with index compression on word-based Huffman text compression is described in

[NMN+00]. With respect to file-addressing indexes, the book [WMB99] describes extensively the *MG* system, a compressed inverted index over compressed text (freely available at `ftp://munnari.oz.au/pub/mg`).

7.4.3 Repeats and repetitions

Much research has been untertaken to study and search for repetitions in texts or sequences, since many of them have a biological role. There exist many definitions of a repeat or a repetition. Moreover, it is not clear how to define and take into account the approximate repetitions that are needed in computational biology. Here is a short summary.

7.4.3.1 Exact repetitions

The first notion of a repetition is simply a factor u that is contiguously repeated more than twice, that is, $u^k v$, where $k \geq 2$ and v is a prefix of u. Clearly, a text may contain a quadratic number of repetitions, for example, $T = a^n$. A first form of repetition that has been extensively studied is the square u^2. Many algorithms exist for finding all the square locations (see [CR94] or [KK99] for a survey). If we consider complete repetitions, the notion of a maximal repetition (sometimes called a *run* or a *maximal periodicity*) represents them all it in a compact way. A repetition is maximal if it cannot be extended in the text to the left or to the right without breaking it. There are at most $O(n)$ factors of the text that can be maximal repetitions, and they can be found in $O(n)$ time [KK99].

A second notion is used when considering noncontiguous repetitions. A *repeat* is a factor of the text that occurs at least twice. A *maximal repeat* is a repeat that cannot be extended to the left or to the right without breaking it. There exist at most a linear number of maximal repeats, and they can be enumerated in $O(n)$ time [Gus97]. However, these definitions do not take into account the relative positions of the repeats. A *pair* u is an occurrence of uvu in the text, and a *maximal pair* cannot be extended, similarly to a maximal repeat or a maximal repetition. The most interesting pairs are usually those such that the two occurrences of u are not too close or not too far away, that is, $|v|$ is bounded between $\delta_1 \leq |v| \leq \delta_2$. An $O(n \log n + nocc)$ time algorithm has been proposed to enumerate such a pair (maximal or not) in a text, where *nocc* is the number of resulting occurrences [BLPS99]. If the upper bound δ_2 is removed, the time reduces to $O(n+nocc)$ [BLPS99].

7.4.3.2 Approximate repetitions

Approximate repetitions are required, for instance, in computational biology, when a sequence rarely matches exactly, and also in musicology, where the

https://doi.org/10.1017/CBO9781316135228.007 Published online by Cambridge University Press

problem is to retrieve repeating themes. The concepts are however more fuzzy since the notions used for exact repetitions can be extended in various ways, depending on the approximate relation we want between the repeated parts.

The approximate concept of *repetition* is usually called *tandem repeat*. Originally, this expression was used for two continuous repetitions uv, where v matches approximately u. The algorithmic problem is to find all these factors in a text. An algorithm taking $O(n^2 \log n)$ time and $O(n^2)$ space exists [Sch98]. If the maximal number of errors is bounded by k, then all the nonoverlapping tandem repeats can be found in $O(kn \log k \log(n/k))$ time [KM93]. These algorithms are mainly of theoretical interest, since $O(n^2)$ is generally too large to be of real use on genomic sequences. Moreover, searching for only two repetitions is a strong limitation, for the computational biologist usually looks for more than two continuous parts. One algorithm for this problem permits finding small satellites [SM98], with a more flexible notion of repetitions (some parts can be missing), but with a strong length limitation (less than 40 bases). The idea is to filter the text, and, with an efficient verification algorithm [FLSS92], this idea leads in [Ben98] to a very fast algorithm/software, called *Tandem repeats finder* [Ben99]. The executable files are available on-line for many operating systems (the software is source protected) at `http://c3.biomath.mssm.edu/trf.html`.

The approximate concept of a *pair* is usually called a *nontandem repeat*. When we are interested in nonoverlapping ones, the same algorithm as for tandem repeats can be used [Sch98], but the real problem becomes managing the huge number of occurrences.

Approximate repetitions in computational biology is a recent and moving topic that evolves rapidly. One of the sofware products most used currently is *RepeatMasker* (part of the *Phrap* package, `http://www.phrap.org/`), which masks some repetitive regions of DNA sequences.

7.4.4 Pattern matching in two and more dimensions

Pattern matching in two-dimensional texts, for instance, in images, is a direct extension of string matching. Many of the most efficient algorithms are extensions of those we presented for one-dimensional text. However, many problems are specific to this field. Books partially covering this issue are [Aoe94, AG97].

In this area we speak of a text of $O(n^2)$ size (i.e., $n \times n$ cells), where a pattern of $O(m^2)$ size ($m \times m$ cells) is sought. This is done for simplicity.

Many algorithms can handle general rectangular and even nonrectangular texts and patterns.

7.4.4.1 Two-dimensional pattern matching

Two-dimensional exact string matching was first considered by Bird and Baker [Bir77, Bak78], who obtained $O(n^2)$ worst-case time. Good average-case results are presented by Zhu and Takaoka [ZT89] and Baeza-Yates and Régnier [BYR93]. Karkkäinen and Ukkonen [KU94] achieved $O(n^2 \log_\sigma m / m^2)$ average-case time, which is optimal.

Two-dimensional approximate string matching usually considers only substitutions for rectangular patterns, which is much simpler than the general case with insertions and deletions, because in this case rows and/or columns of the pattern can match pieces of the text of different length.

If we consider matching the pattern with at most k substitutions, one of the best results for the worst case, due to Amir and Landau [AL91], is $O((k + \log \sigma)n^2)$ time using $O(n^2)$ space. A similar algorithm is presented by Crochemore and Rytter [CR94]. Ranka and Heywood [RH91], on the other hand, solve the problem in $O((k + m)n^2)$ time and $O(kn)$ space. Amir and Landau also present a different algorithm running in $O(n^2 \log n \log \log n \log m)$ time. On average, the best algorithm is due to Karkkäinen and Ukkonen [KU94, Par96]. The expected time is $O(n^2 k \log_\sigma m / m^2)$ for $k \leq m^2/(4 \log_\sigma m)$, using $O(m^2)$ space ($O(k)$ space on average). This expected complexity is optimal.

The extension of the classic notion of edit distance is difficult. Krithivasan and Sitalakshmi [KS87] defined the edit distance in two dimensions as the sum of the edit distances of the corresponding row images. Let us call it the KS model. Using this notion they obtain $O(m^2 n^2)$ search time. Krithivasan [Kri87] presents for the same model an $O(m(k + \log m)n^2)$ time algorithm that uses $O(mn)$ space. Amir and Landau [AL91] give an $O(k^2 n^2)$ worst-case time algorithm using $O(n^2)$ space (note that k can be larger than m, so this is not necessarily better than the previous algorithms). Amir and Farach [AF91] also considered nonrectangular patterns, achieving $O(k(k + \sqrt{m \log m}\sqrt{k \log k})n^2)$ time. Finally, Navarro and Baeza-Yates obtain $O(n^2 k \log_\sigma m / m^2)$ average case time for the KS model, for $k < m(m+1)/(5 \log_\sigma m)$, using $O(m^2)$ space.

For many two-dimensional matching problems, the KS distance does not reflect well simple cases of approximate matching in different settings. New distances and search algorithms have been introduced recently [BYN00b]. In the new models the errors can occur along rows or columns.

7.4.4.2 Other multidimensional matching problems

Other problems related to comparing images are searching allowing rotations [FNU01] and scaling [ABL00].

There are other approaches to matching images that are very different from those cited above (which belong to what is called combinatorial pattern matching). Among them we can mention techniques used in pattern matching related to artificial intelligence, for example, image processing and neural networks, and techniques used in databases, for example, extracting features of the image such as color histograms.

All the previous problems can be generalized to more than two dimensions, and many algorithms running on two-dimensional texts can be extended to run in more dimensions [AL91, KU94, GG97, FU98, BYN00b].

7.4.5 Tree pattern matching

Tree pattern matching is another extension of pattern matching in text. The problem arises in computational biology when we need to compare RNA structures. It also arises when a program is represented during compilation as an instruction tree in which we want to find special patterns, usually to perform some optimizations. Hierarchically structured text databases also require this form of matching.

For exact tree pattern matching there are two ordered trees called the *text* and the *pattern* by extension of string pattern matching, and the problem is to find all the occurrences of the pattern in the text, that is, all nodes of the text rooting a subtree that matches the pattern. We denote by n and m the sizes (in number of nodes) of the text and the pattern. The naive algorithm, which consists of verifying each text node, runs in $O(mn)$ worst-case time.

Several algorithms exist that improve the worst-case bound, but they are mainly theoretical. An $O(nm^{0.75} \log m)$ time algorithm has been proposed in [Kos89], using a connection between this problem and that of searching strings with "don't care" symbols. This in turn has been improved in [DGM94] to $O(nm^{0.5} \log m)$, using the periodicities of the paths of the pattern tree. A major improvement over this has been obtained in several articles by the same authors, leading to an $O(n \log^3 n)$ time algorithm in [CHI99]. A survey on this topic can be found in [ZS97].

The approximate tree pattern matching problem arises in computational biology when comparing and aligning trees. It implies a notion of distance on trees [ZSW94, SZ97]. Many algorithms exist, but this topic is still in development since the notion has to fit exacly the biological properties of

the objects that the trees represent, which are generally secondary structures [ZWM99].

A related notion dealing with trees and matching is the maximum subtree agreement problem. Given two rooted trees whose leaves are taken from the same set of items, which for instance represent two phylogenetic trees, the problem is to find the largest subset so that the portions of the two trees restricted to these items are isomorphic. When the two trees are binary, which is usually the case in practice, an $O(n \log n)$ time algorithm exists [CFCH+01].

Finally, structured text databases introduce a concept of tree pattern matching similar to the "extended strings" considered in this book. In this case the pattern is a tree, but each pattern edge can match an arbitrary path in the text tree. It is also possible to force the occurrence to honor the intersibling ordering given in the pattern. Depending on these options, the search complexity goes from polynomial time to NP-complete [Kil92, KM92].

7.4.6 Sequence comparison

Sequence comparison is about determining similarities and correspondences between two or more strings. It is related to approximate searching (Chapter 6) and has many applications in computational biology, speech recognition, computer science, coding theory, chromatography, and so on. These applications look for similarities between sequences of symbols. The general goal is to perform basic operations over the strings until they become equal. Those basic operations have an associated cost, and we seek the minimum-cost sequence of operations that achieves the goal. The reason for preferring the minimum cost is different in each application, but the general idea is that the sequences differ by a series of alterations on one or both of them, and the cheapest series are those of maximum likelihood.

A concept of "distance" between two strings can be defined according to the minimum cost of making them equal. The basic operations considered depend on the application, but the most typical are supressing characters, inserting characters, substituting characters by others, swapping adjacent or nonadjacent characters, reversing substrings, moving substrings to another place of the string, compressing a run of equal characters, expanding a single character to a run of them, and so on. Each operation has a rationale in the model where it is used. The application also gives a rationale for assigning costs to the different operations, for example, the most likely operations cost less. A simple case is to assign a cost that is the logarithm of the probability of this operation occurring in the process that made the strings

differ. Hence the sum of the costs corresponds to the logarithm of the product of the probabilities of the operations, which is a good model if they are independent.

Two of the most popular similarity measures are the Levenshtein distance and the "indel" distance. The first one permits character insertions, deletions, and substitutions, all at cost 1. The second one permits only insertions and deletions and is related to the longest common subsequence (LCS) between the two strings. The popularity of these models lies in their simplicity, in the efficiency of the algorithms that handle them, and in the simplicity of their mathematical properties. We devoted Chapter 6 almost entirely to the Levenshtein distance.

In many applications it is also interesting to know how the two sequences differ. In general we speak about *aligning* the two sequences (recall Section 6.2.1). There are several ways to express an alignment. One is to put the strings one on top of the other and space their characters so that similar characters are in the same column. The amount of spacing needed gives an idea of how different the strings are. Another method is to draw a set of *traces*, where lines connect the aligned characters in both strings (Figure 6.1). Yet a third way, less popular but very useful mathematically, is to list the operations made on the strings.

In some cases it is useful to measure the degree of similarity rather than of dissimilarity (i.e., a distance). One example is the LCS, a heavily studied measure. Other examples are the shortest common supersequence (SCS), longest common substring (LCG, different from the LCS because the common string has to be a contiguous substring of both sequences), and shortest common superstring (SCG), as well as their versions on more than two strings.

Most algorithms for sequence comparison rely on dynamic programming, since it is useful to have all previous results precomputed in order to use them. However, backtracking has also been used. The simplest distances such as Levenshtein or indel, even with arbitrary costs for the operations, can be computed in $O(n^2)$ time for two strings of length n. The same holds for computing the LCS or SCS of two strings. For N strings the cost raises to $O(n^N)$ and is NP-complete for arbitrary N. The cost for LCG and SCG is $O(n)$ for two strings. There exist complicated algorithms that improve over the quadratic complexity under diverse assumptions. On the other hand, computing the distance when block moves are involved is NP-complete in some cases. This shows that the nature of the problem depends on the type of distance used.

Another issue in sequence comparison is statistics. How significant is it

that the LCS between two binary strings is 80% of their length? Does it mean that they are close, or could it happen perfectly well to two random sequences? There has been much research on the expected length of the LCS between two random strings. It is known that it grows linearly with n, but the exact constant is not yet known; only tight upper and lower bounds exist.

We refer the reader to good books on the subject [SK83] or to the section of books of computational biology (7.2.2).

7.4.7 Meaningful string occurrences

The problem of finding factors that are "unusual" in sequences is a topic that has led to many studies. There are four main underlying points.

First, the sequences have to be considered under a specific probability model, for instance, under a Markov model, to get a comparison point for what should be considered normal.

Next, a formula to get the "normal" occurrence probability or other interesting parameters, for instance, the expected distance between two occurrences, has to be obtained. The expected probability of the approximate occurrences of a string has recently been obtained [RS97]. An algorithm to compute the expected frequency of occurrence of a string, a set of strings, or a regular expression under both the Bernoulli or Markov models exists [NSF01], but evaluating the formula is complicated and computationally time-consuming (the same is true for [RS97]). A *Maple* package implementing the evaluation, called *Algolib*, is available at `http://algo.inria.fr/libraries/software.html`.

Third, given a certain model and a way to compute the interesting parameters of a given factor in that model, one has to decide which factor of the text should be considered as unusual.

Finally, when the three previous problems have been solved for a certain type of sequence, model, and pattern, the algorithmic problem is to find and visualize the unusual factors in an efficient and usable way (for that there could be $O(n^2)$ such factors).

A short survey on these questions can be found in [Pev00], but, to our knowledge, there is no complete survey grouping these questions together. A sofware product called *Verbumculus* [ABLX00] permits visualizing unusual factors under a restricted definition of what a usual factor is. The binaries are available at `http://www.cs.purdue.edu/homes/stelo/Verbumculus`.