Chapter 1

Data Structures and Algorithms on Words

In this book we choose one facet of the theory of algorithms, namely that of data structures and algorithms on words (strings, sequences) to illustrate probabilistic, combinatorial and analytic techniques of analysis. In this chapter, we briefly describe some data structures and algorithms on words that are used extensively throughout the book to illustrate the methods of analysis.

Data Structures and algorithms on words have experienced a new wave of interest due to a number of novel applications in computer science, communications, and biology. Among others, these include dynamic hashing, partial match retrieval of multidimensional data, searching and sorting, pattern matching, conflict resolution algorithms for broadcast communications, data compression, coding, security, genes searching, DNA sequencing, gnome maps, double digest problem, and so forth. To satisfy these diversified demands various data structures were proposed for these algorithms. Undoubtly, the most popular data structures in algorithms on words are digital trees [2, 157, 175] (e.g., tries, PATRICIA, digital search trees), and in particular suffix trees [2, 12, 45, 207, 210, 226, 227]. We discuss below various digital trees and introduce several parameters characterizing them that we shall study throughout the book.

The importance of digital trees stem from their abundance of applications in other problems such as data compression (cf. Section 1.2), pattern matching (cf. Section 1.3), and the shortest common superstring problem (cf. Section 1.4). These problems recently became very important due to the need for an efficient storage and transmission of multimedia, and possible applications to the DNA sequencing.

Graphs and directed acyclic graphs (DAG) also find several applications in problems on strings. In particular, we consider the *edit distance* problem and its variants (cf. Section 1.5). Finally, we close this chapter with a brief discussion of certain class of optimization problems on graphs that find applications for algorithms on words (cf. Section 1.6).

1.1 Digital Trees

We start our discussion with a brief review of the **digital trees**. The most basic digital tree known as a **trie** (the name comes from re*trie*val) is defined first, and then other digital trees (such as PATRICIA, digital search trees and suffix trees) are described in terms of the trie.

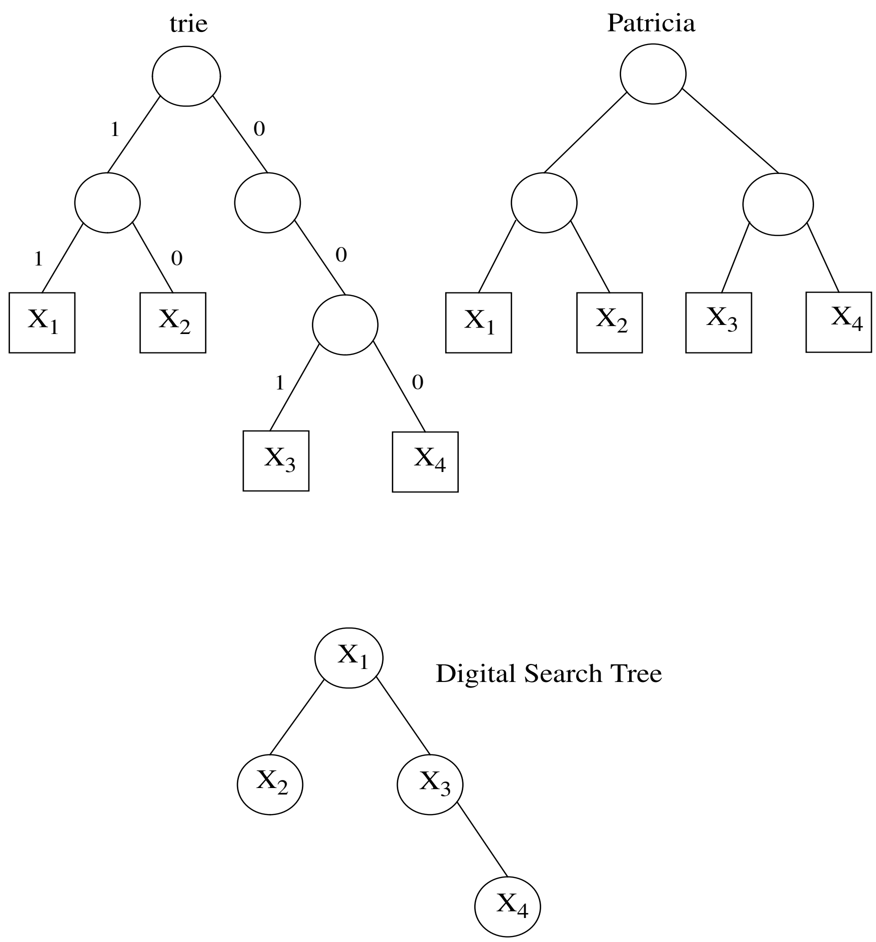

Figure 1.1: A trie, Patricia trie and a digital search tree (DST) built from the following four strings $X^1 = 11100\ldots$, $X^2 = 10111\ldots$, $X^3 = 00110\ldots$, and $X^4 = 00001\ldots$.

The primary purpose of a trie is to store a set $\mathcal{X}$ of strings (words, sequences), say $\mathcal{X} = \{X^1, \ldots, X^n\}$. Throughout the book, we use exchangeably the terms strings, words and sequences. Each string is a finite or infinite sequence of symbols taken from a finite alphabet $\Sigma = \{\omega_1, \ldots, \omega_V\}$ of size $V = |\Sigma|$. We use a generic notation $\mathcal{D}_n$ for all digital trees built over a set $\mathcal{X}$ of n strings. A string will be stored in a leaf of the trie. The trie over $\mathcal{X}$ is built recursively as follows: For $|\mathcal{X}| = 0$, the trie is, of course, empty. For $|\mathcal{X}| = 1$, $trie(\mathcal{X})$ is a single node. If $|\mathcal{X}| > 1$, $\mathcal{X}$ is split into V subsets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_V$ so that a string is in $\mathcal{X}_j$ if its first symbol is ω_j. The tries $trie(\mathcal{X}_1), trie(\mathcal{X}_2), \ldots, trie(\mathcal{X}_V)$ are constructed in the same way except that at the k-th step, the splitting of sets is based on the k-th symbol. They are then connected from their respective roots to a single node to create $trie(S)$. Figure 1.1

illustrates such a construction. Observe that all strings are stored in external nodes (shown as boxes in Figure 1.1) while internal nodes are branching nodes used to direct strings to their destination (i.e., external nodes).

There are many possible variations of the trie. One such variation is the **b-trie** in which a leaf is allowed to hold as many as b strings (cf. [79, 175, 226]). The b-trie is particularly useful in algorithms for extendible hashing in which the capacity of a page or other storage unit is b. A second variation of the trie, the **PATRICIA trie**, eliminates the waste of space caused by nodes having only one branch. This is done by collapsing one-way branches into a single node. In a **digital search tree** (in short: DST) strings are directly stored in nodes, and hence external nodes are eliminated. The branching policy is the same as in tries. As in the case of a trie, we can consider an extension of the above digital trees by allowing to store up to b strings. We denote such digital trees as $\mathcal{D}_n^{(b)}$, but we often drop upper index when $b = 1$ is discussed. Figure 1.1 illustrates these definitions for $b = 1$.

One of the most important example of tries and PATRICIA trees are **suffix trees** and *compact suffix trees* (also called PAT). In suffix trees and compact suffix trees, the words stored in these trees are suffixes of a given string $X = x_1x_2\ldots$; that is, the word $X^j = x_jx_{j+1}x_{j+2}\cdots$ is the jth suffix of X which begins at the j-th position of X. Thus a suffix tree is a trie and a compact suffix tree is a PATRICIA trie in which the words are all suffixes of a *given* string. Clearly, in this case the strings X^j for $j = 1, \ldots, n$ depend strongly on each others while in a trie the strings of $\mathcal{X}$ might be completely independent. A suffix tree is illustrated on Figure 1.2.

Certain characteristics of tries and suffix trees are of primary importance. Hereafter, we assume that a digital tree is built from n strings or a suffix tree is constructed from a string of length n and is capable of storing up to b strings in a node.

Definition 1.1 (Digital Trees Parameters) *Let us consider a b-digital tree $\mathcal{D}_n^{(b)}$ built over n strings and capable of storing up to b strings in a (external) node.*

(i) *The m-th depth $D_n^{(b)}(m)$ is the length of a path from the root of the digital tree to the (external) node containing the mth string.*

(ii) *The typical depth $D_n^{(b)}$ is the depth of a randomly selected string, that is,*

$$\Pr\{D_n^{(b)} \le k\} = \frac{1}{n}\sum_{m=1}^{n} \Pr\{D_n^{(b)}(m) \le k\} \ . \tag{1.1}$$

(iii) *The (external) path length $L_n^{(b)}$ is the sum all depths, that is,*

$$L_n^{(b)} = \sum_{m=1}^{n} D_n^{(b)}(m). \tag{1.2}$$

(iv) *The height $H_n^{(b)}$ is the length of the longest depth, that is,*

$$H_n^{(b)} = \max_{1\le m\le n}\{D_n^{(b)}(m)\}. \tag{1.3}$$

$S_1 = 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0$
$S_2 = 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0$
$S_3 = 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0$
$S_4 = 1\ 1\ 0\ 1\ 1\ 1\ 0$
$S_5 = 1\ 0\ 1\ 1\ 1\ 0$

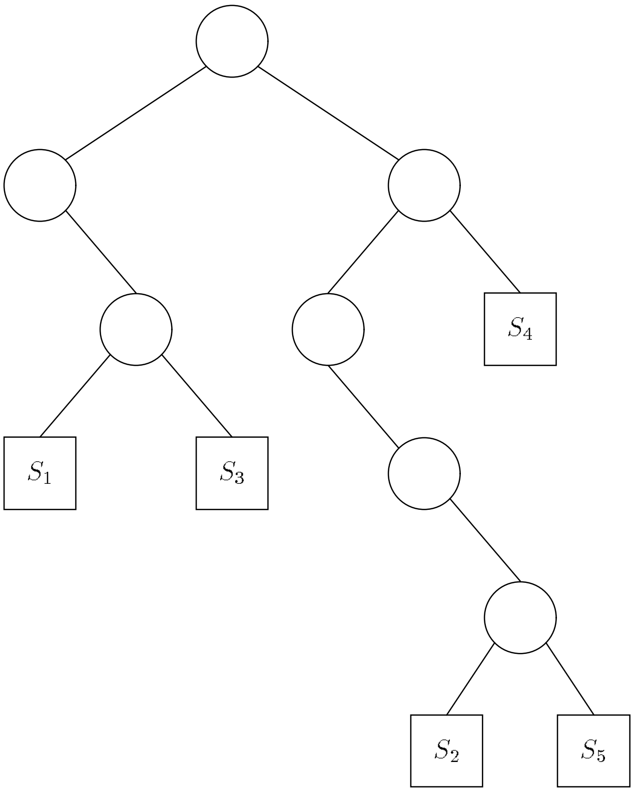

Figure 1.2: Suffix tree built from the first five suffixes $S_1, \ldots, S_5$ of $X = 0101101110\ldots$

(v) *The shortest feasible path* $s_n^{(b)}$ *is the length of the shortest path from the root to an available (feasible) node, that is, a node that is not in the tree* $\mathcal{D}_n^{(b)}$ *but whose predecessor node (either an internal node or an external node) is in* $\mathcal{D}_n^{(b)}$. *Observe that* $\mathcal{D}_n^{(b)}$ *is a complete tree up to level* $s_n^{(b)}$.

(vi) *The shortest depth* $\widetilde{s}_n^{(b)}$ *is the length of the shortest path from the root to an external node. (Clearly,* $s_n^{(b)} \leq \widetilde{s}_n^{(b)}$.*)*

(vii) *The size* $S_n^{(b)}$ *is the number of (internal) nodes in* $\mathcal{D}_n^{(b)}$.

These parameters can be conveniently represented in another way that reveals combinatorial relationships between strings stored in such digital trees. We start with the following definition.

Definition 1.2 ALIGNMENTS
For the set of strings $\mathcal{X} = \{X^1, X^2, \ldots, X^n\}$*, the alignment* $C_{i_1 \ldots i_{b+1}}$ *between* $b+1$ *strings* $X^{i_1}, \ldots, X^{i_{b+1}}$ *is the length of the longest common prefix of all these* $b+1$ *strings.*

To illustrate this definition and to show its usefulness to the analysis of digital trees, we consider the following example.

Example 1.1: *Illustration of the Self-Alignments*
Let us consider the suffix tree shown in Figure 1.2. Define $C = \{C_{ij}\}_{i,j=1}^5$ for $b = 1$ as the self-alignment matrix which is shown explicitly below:

$$C = \begin{bmatrix} \star & 0 & 2 & 0 & 0 \\ 0 & \star & 0 & 1 & 4 \\ 2 & 0 & \star & 0 & 0 \\ 0 & 1 & 0 & \star & 1 \\ 0 & 4 & 0 & 1 & \star \end{bmatrix}.$$

Observe that we can express the above parameters of the suffix tree in terms of the self-alignments C_{ij} as follows

$$\begin{aligned} D_n(1) &= \max_{2 \le j \le 5} \{C_{1j}\} + 1 = 3, \\ H_n &= \max_{1 \le i < j \le n} \{C_{ij}\} + 1 = 5, \\ \tilde{s}_n &= \min_{1 \le i \le n} \max_{1 \le j \le n} \{C_{ij}\} + 1 = 2 \end{aligned}$$

(since $b = 1$ we drop b from the above notations).

Certainly, similar relationships hold for tries, but not for PATRICIA tries and digital search trees. In the latter case, however, one can still express parameters of the trees in terms of the alignments matrix C. For example, the depth of the fourth string $D_4(4)$ can be expressed as follows

$$D_4(4) = \max\{\min\{C_{41}, D_4(1)\}, \min\{C_{42}, D_4(2)\}, \min\{C_{43}, D_4(3)\}\}.$$

This is a little bit too complicated to be of any usage. □

The above example suggests that there are relatively simple relationships between parameters of a trie and the alignments. We show it in the theorem below. The reader is asked to provide a formal prove in Exercise 1.

Theorem 1.3 *In a trie the following holds:*

$$D_n^{(b)}(i_{b+1}) = \max_{1\le i_1,\ldots,i_b\le n}\{C_{i_1\ldots i_{b+1}}\}+1, \tag{1.4}$$

$$H_n^{(b)} = \max_{1\le i_1,\ldots,i_{b+1}\le n}\{C_{i_1\ldots i_{b+1}}\}+1, \tag{1.5}$$

$$D_n^{(b)} = \max_{1\le i_1,\ldots,i_b\le n}\{C_{i_1\ldots i_b,n+1}\}+1, \tag{1.6}$$

$$\widetilde{s}_n^{(b)} = \min_{1\le i_{b+1}\le n}\{D_n^{(b)}(i_{b+1})\} = \min_{1\le i_{b+1}\le n}\max_{1\le i_1,\ldots,i_b\le n}\{C_{i_1\ldots i_{b+1}}\}+1 \tag{1.7}$$

for any $1\le i_1,\ldots,i_{b+1}\le n$*.*

In passing, we should mention that the above combinatorial relationships find applications in problems not directly related to digital tree. We shall meet them again when analyzing the shortest common superstring problem discussed in Section 1.4.

The digital trees are used very extensively in this book as illustrative examples. We analyze the height of tries, PATRICIA tries and suffix trees in Chapter 4. Recurrences arising in the analysis of digital trees are discussed in Chapter 7 while the typical depth of digital search trees is studied in Chapters 8–10.

1.2 Data Compression: Lempel-Ziv Algorithms

Source coding is an area of information theory (cf. Chapter 6) that deals with problems of optimal data compression. One of the most successful and known data compression schemes are due to Lempel and Ziv [259, 260]. Efficient implementation of these algorithms involves digital trees. We describe here some aspects of the Lempel-Ziv schemes and return to them in Chapter 6.

We start with some definitions (the reader can find in depth discussion of them in Chapter 6). Consider a sequence $\{X_k\}_{k=1}^{\infty}$ taking values in a finite alphabet $\mathcal{A}$ (e.g., for English text the cardinality $|\mathcal{A}|=26$ symbols, while for an image $|\mathcal{A}|=256$). We write X_m^n to denote $X_m, X_{m+1}\ldots X_n$. We encode X_1^n into a *binary* (compression) code $\mathscr{C}_n$, and the decoder produces the reproduction sequence $\hat{X}_1^n$ of X_1^n. More precisely, a code $\mathscr{C}_n$ is a function $\phi:\ \mathcal{A}^n\to\{0,1\}^*$. On the decoding side, the decoder function $\psi:\ \{0,1\}^*\to\mathcal{A}^n$ is applied to find $\hat{X}_1^n=\psi(\phi(X_1^n)$. Let $\ell(\mathscr{C}_n(X_1^n))$ be the length of the code $\mathscr{C}_n$ (in bits) representing X_1^n. Then the *bit rate* is defined as

$$r(X_1^n)=\frac{\ell(\mathscr{C}_n(X_1^n))}{n}.$$

For example, for text $r(X_1^n)$ is expressed in bits per symbol, while for image compression bits per pixel or in short bpp.

We shall discuss below two basic Lempel-Ziv schemes, namely the so called Lempel-Ziv'77 [259] (in short: LZ77) and Lempel-Ziv'78 [260] (in short: LZ78). Both schemes are examples of **lossless** compression, that is, the decoder can recover exactly the encoded sequence. In this book, a number of interesting problems will be generated by a lossy extension of the Lempel-Ziv schemes. In **lossy** data compression, discussed below, some information can be lost during the encoding.

1.2.1 Lempel-Ziv'77 Algorithm

The Lempel-Ziv algorithm partition or parsing a sequence into phrases that are similar in some sense. Depending on how such a parsing is encoded we have different versions of the algorithm. However, the basic idea is to find the longest prefix of yet uncompressed sequence that occurs in the already compressed sequence.

More specifically: let us assume that the first n symbols X_1^n have been parsed. This initial string is sometimes called the "database string" or the "training sequence". We search for the longest prefix $X_{n+1}^{n+\ell}$ of X_{n+1}^{∞} that is repeated in X_1^n, that is,

> Let I_n be the largest ℓ such that $X_{n+1}^{n+\ell} = X_m^{m-1+\ell}$ for some index m for some prescribed range of m and ℓ.

Depending on the range of m and ℓ, we can have different versions of LZ'77 scheme. In the original LZ'77 algorithm [259], m and ℓ were restricted to a window size W and "lookahead" buffer B, that is, $n-W+1 \le m \le n$ and $\ell \le B$. This implementation is sometimes called the *sliding window* LZ'77. In the *fixed database* (FDLZ) version [253, 254], one sets $1 \le m \le W$ and $m - 1 + \ell \le W$, that is, the database sequence is fixed and the parser always looks for matches inside such a fixed substring X_1^W. Such a scheme is sometimes called the Wyner-Ziv scheme [253]. Finally, in the *growing database* version the only restriction is that $1 \le m \le n$, i.e., the database consists of the last n symbols.

In general, the code built for LZ'77 consists of a triple of numbers: $(m, \ell, \texttt{char})$ where `char` is the symbol $X_{m+\ell}$. Since the pointer to m needs $\log_2 n$ bits, the length ℓ could be coded in $O(\log I_n)$ bits and `char` requires $\log|\mathcal{A}|$ bits, the code length of a *phrase* is $\log_2 n + O(\log I_n) + \log_2|\mathcal{A}|$ bits. In Exercise 5 we propose a formula for the code length of the FDLZ, while in Exercise 6 the reader is asked to find the code length for all other versions of the Lempel-Ziv schemes.

The heart of all versions of the Lempel-Ziv schemes is the algorithm that finds the longest prefix of length I_n that occurs in the database string of length n. It turns out that the suffix tree discussed in Section 1.1 can be used to find efficiently such a prefix. Indeed, let us consider a sequence $X = 0101101110\ldots$, and assume X_1^4 is the database string. The suffix tree built over X_1^4 is shown in Figure 1.3. Let us now look for I_4, that is, the longest prefix of X_5^{∞} that occurs (starts) in the database X_1^4. In the growing database implementation it is

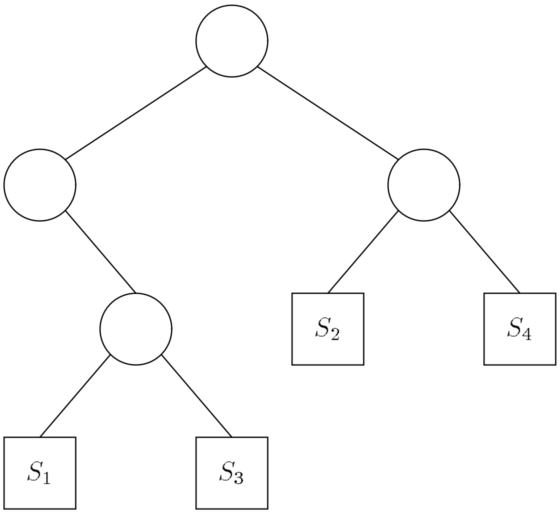

Figure 1.3: Suffix tree built from the first four suffixes of $X = 0101101110...$

X_5^8 since it is equal to X_2^5. This can be seen by inserting the fifth suffix of X into the suffix tree from Figure 1.3 – which actually leads to the suffix tree shown in Figure 1.2.

1.2.2 Lempel-Ziv'78 Algorithm

The Lempel-Ziv'78 (LZ'78) is a *dictionary-based* scheme that partitions a sequence into phrases (blocks) of variable sizes such that a new block is the shortest substring not seen in the past as a phrase. Every such a phrase is encoded by the index of its prefix appended by the symbol, thus LZ'78 code consists of pairs ($\texttt{pointer}, \texttt{symbol}$). A phrase containing only one symbol are coded with the index equal to zero.

Example 1.2: *The Lempel-Ziv'78 and Its Code*

Consider the string $X_1^{14} = ababbbabbaaaba$ over the alphabet $\Sigma = \{a, b\}$ which is parsed and coded as follows:

Phrase No:	1	2	3	4	5	6	7
Sequence:	(a)	(b)	(ab)	(bb)	(abb)	(aa)	(aba)
Code:	0a	0b	1b	2b	3b	1a	3a

Observe that we need $= \lceil \log_2 7 \rceil$ bits to code a phrase, two bits to code a symbol, so in total for 7 phrases we need 28 bits. □

The most time consuming part of the algorithm is finding the next phrase, that is, searching the dictionary. However, this can be speed up by using a digital search tree to build the dictionary. For example, the string 11001010001000100 that is parsed into

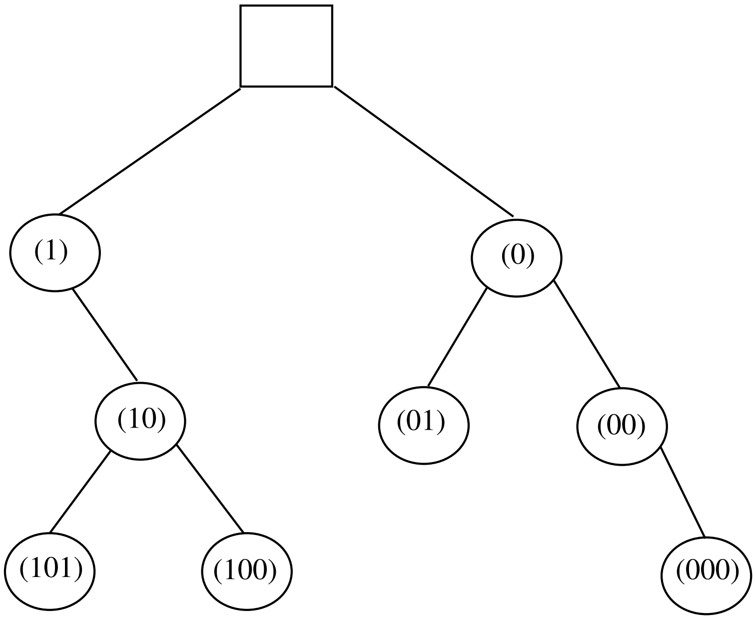

Figure 1.4: A digital tree representation of the Lempel-Ziv parsing for the string 11001010001000100

(1)(10)(0)(101)(00)(01)(000)(100) and this process is represented in Figure 1.4 using the digital search tree structure. Contrary to our previous digital search construction, we leave the In this case, however, we leave the root empty (or we put an empty phrase into it). To show that the root is different than other nodes we draw it in Figure 1.4 as a square. All other phrases of the Lempel-Ziv parsing algorithm are stored in internal nodes (represented in the figure as ovals). When a new phrase is created, the search starts at the root and proceeds down the tree as directed by the input symbols exactly in the same manner as in the digital tree construction, that is, symbol "0" in the input string means move to the right and "1" means proceed to the left. The search is completed when a branch is taken from an existing tree node to a new node that has not been visited before. Then, the edge and the new node are added to the tree. The phrases created in that way are stored directly into the nodes of the tree.

We should observe differences between digital search trees discussed in Section 1.1 and the one described above. For the Lempel-Ziv scheme we consider a word of *fixed length*, say n, while before we dealt with *fixed number of strings*, say m, resulting in a digital tree consisting of exactly m nodes. Looking at Figure 1.4, we conclude that the number of nodes in the associated digital tree is equal to the number of phrases generated by the Lempel-Ziv algorithm.

1.2.3 Extensions of Lempel-Ziv Schemes

Finally, we shall discuss two extensions of Lempel-Ziv schemes, namely: *generalized* Lempel-Ziv'78 and *lossy* Lempel-Ziv'77. Not only are these extensions useful from a practical point of view (cf. [173, 215, 201]), but they are also a source of interesting analytical problems. We often return to them in the course of this book.

Let us first consider the **generalized** Lempel-Ziv'78 scheme. It is known that the original Lempel–Ziv scheme does not cope very well with sequences containing a long string of repeated symbols. To somewhat remedy this situation, Louchard *et al.* [173] introduced a generalization of the Lempel–Ziv parsing scheme that works as follows: Fix an integer $b \geq 1$. The algorithm parses a sequence into phrases such that the next phrase is the *shortest* phrase seen in the past by *at most* $b-1$ phrases ($b = 1$ corresponds to the original Lempel–Ziv algorithm). It turns out that such an extension of the Lempel-Ziv algorithm protects against the propagation of errors in the dictionary (cf. [215, 201]).

Example 1.3: *Generalized Lempel-Ziv'78*

Consider the sequence $ababbababaaaaaaaaac$ over the alphabet $\Sigma = \{a, b, c\}$. For $b = 2$ it is parsed as follows: $(a)(b)(a)(b)(ba)(ba)(baa)(aa)(aa)(aaa)(c)$ which has seven *distinct* phrases and eleven phrases. The code for this new algorithm consists: (i) either of (`pointer`, `symbol`) when `pointer` refers to the *first* previous occurrence of the prefix of the phrase and `symbol` is the value of the last symbol of this phrase; (ii) or just (`pointer`) if the phrase has occurred previously (i.e., it is the second or the third or ... or the bth occurrence of this phrase). For example, the code for the previously parsed sequence is for $b = 2$: $0a0b122a33a1a55a0c$ (e.g., the phrase 2a occurs for the first time as a new phrase hence 2 refers to the second distinct phrase appended by a, while code 5 represents a phrase that has its second occurrence as the fifth distinct phrase). Observe that this code is of length 47 bits since there are eleven phrases each requiring up to $\lceil \log_2 7 \rceil = 3$ bits and seven symbols need 14 additional bits (i.e., $47 = 11 \cdot 3 + 7 \cdot 2 = 47$). The original LZ'78 code needs 54 bits. We saved 7 bits! But, the reader may verify that the same sequence requires only 46 bits for $b = 3$ (so only additional bit is saved), while for $b = 4$ the bit count increases again to 52. □

The above example suggests that $b = 3$ is (at least local) optimum for the above sequence. Can one draw similar conclusions "on average" for a typical sequence (i.e., generated randomly)? This book is intended to provide tools to analyze such problems.

As for the original Lempel-Ziv algorithm, the most time consuming part of the construction is to generate a new phrase. An efficient way of accomplishing this is by means of generalized digital search trees introduced in Section 1.1, namely: b-digital search tree (in short: b-DST). We recall that in such a digital tree one is allowed to store up to b strings in a node. In Figure 1.5 we show the 2-DST constructed from the sequence 1100101000100010011.

We now discuss another extension, namely: a **lossy** Lempel-Ziv'77 scheme. In such a scheme in the process of coding some information might be lost. To control this loss, one

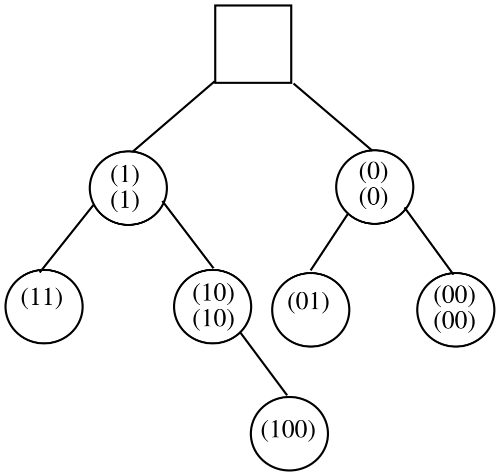

Figure 1.5: A 2-digital search tree representation of the generalized Lempel–Ziv parsing for the string 1100101000100010011

needs a measure of fidelity $d(\cdot,\cdot)$ between two sequences. For example, the Hamming distance is defined as

$$d_n(X_1^n, \hat{X}_1^n) = \frac{1}{n}\sum_{i=1}^{n} d_1(X_i, \hat{X}_i)$$

where $d_1(X_i, \hat{X}_i) = 0$ for $X_i = \hat{X}_i$ and 1 otherwise In the square error distortion we set $d(X_i, \hat{X}_i) = (X_i - \hat{X}_i)^2$. We explore this topic in depth in Chapter 6.

Let us now fix $D > 0$. In the lossy LZ'77, we consider the longest prefix of the uncompressed file that approximately (within distance D) occurs in the database sequence. More precisely, the quantity I_n defined in Section 1.2.1 becomes in this case:

> Let I_n be the largest K such that a prefix of X_{n+1}^{∞} of length K is within distance D from X_i^{i-1+K} for some $1 \le i \le n-K+1$, that is, $d(X_i^{i-1+K}, X_{n+1}^{n+K}) \le D$.

Not surprisingly, the bit rate of such a compression scheme depends on the probabilistic behavior of I_n. We shall analyze it in Chapter 6.

1.3 Pattern Matching

There are various kinds of patterns occurring in strings that are important to locate. These include squares, palindromes, and a specific pattern. For example, in computer security one wants to know if a certain pattern (i.e., a substring or even better a subsequence) appears (too) frequently in an audit file (text) since this may indicate an intrusion. In general, pattern matching involves a pattern H and a text string T. pattern matching algorithms are

H	0 1 0 1 1 0
T	1 0 1 1 0 1 0 0 1 1
	(a) first attempt
H	0 1 0 1 1 0
T	1 0 1 1 0 1 0 0 1 1
	(b) second attempt
H	0 1 0 1 1 0
T	1 0 1 1 0 1 0 0 1 1
	(c) third attempt

Figure 1.6: Pattern matching

to determine the existence of H within T, the first occurrence of H, the number of occurrences or the location of all occurrences of H.

Two well-known pattern matching algorithms are the Knuth-Morris-Pratt (KMP) algorithm and the Boyer-Moore (BM) algorithm [2, 45]. (In this section we focus our attention on KMP algorithm, and the reader is referred to [32, 45] for a detailed description of other pattern matching algorithms.) As in other matching algorithms, the efficiency of these algorithms depends greatly on determining the location of the next matching attempt, given that the previous attempt was unsuccessful. The key observation here is that following a mismatch at the k-th position of the pattern, the preceding $k-1$ symbols of the pattern and their structure that give insight as to where the next matching attempt should begin. This idea is used in the KMP pattern matching algorithm and is illustrated in the following example and Figure 1.6.

Example 1.4: *Comparisons Made by KMP Algorithm in Fig. 1.6*

Let $H_1^6 = 010110$ and the text string $T_1^{10} = 1011010011$. When attempting to match P with T, proceed from left to right, comparing each symbol. No match is made with the first symbol of each, so the pattern H is moved one position to the right. On the second attempt, the third symbol of H does not match the text, so this attempt is halted and the pattern H is shifted to the right. Notice that it is not fruitful to begin matching at either the third or fourth position of T since it was discovered in the previous matching attempt that neither is a 0. Thus, the next matching attempt begins at the fifth symbol of T. □

Knowing how far to shift the pattern H is the key to both the KMP and the BM algorithms. In each algorithm, the pattern H is preprocessed to determine the shift for a failure at the i-th symbol of H for each i. To be more precise, we consider a class of Morris-Pratt like algorithms, defined below, and define the shift for them.

It is known that in Morris-Pratt like algorithms after a mismatch occurs when comparing T_l with H_k, some alignment positions in $l+1,\ldots,l+k-1$ can be disregarded without further text-pattern comparisons. Namely, the ones that satisfy $T_{l+i}^{l+k-1} \neq H_1^{k-i}$ or equivalently $H_{1+i}^{k} \neq H_1^{k-i}$. The set of such i can be known by a preprocessing of H. Other i define the "surviving candidates". Different choices lead to different variants of the classic Morris-Pratt algorithm [158]. They differ by the use of the information obtained from the mismatching position. We formally define three main variants, and provide an example. One defines a shift function S to be used after any mismatch as:

Morris-Pratt variant:

$$S = \min\{k-1;\ \min\{s>0:\ H_{1+s}^{k-1} = H_1^{k-1-s}\}\}\ ;$$

Knuth-Morris-Pratt variant:

$$S = \min\{k;\ \min\{s:\ H_{1+s}^{k-1} = H_1^{k-1-s}\textbf{and}\ H_k^k \neq H_{k-s}^{k-s}\}\}\ ;$$

Simon variant: Let K be the largest k such that H_k is compared to a symbol in the text. Then

$$\begin{aligned} B &= \{s:\ p_{1+s}^{K-1} = p_1^{K-1-s}\textbf{and}\ 0 \le s \le K-k\}, \\ S &= \min\{d>0: H_{1+d}^{k-1} = H_1^{k-1-d}\ \textbf{and}\ (H_{k-d}^{k-d} \neq p_{K-s}^{K-s}, s \in B)\}. \end{aligned}$$

There are several parameters of pattern matching algorithms that either determine their performance or shed some light on their behaviors. For example, efficiency of an algorithm is characterized by its complexity defined below.

Definition 1.4 (i) *For any pattern matching algorithm that runs on a given text T and a given pattern H, let $M(l,k)=1$ if the l-th symbol T_l of the text is compared by the algorithm to the k-th symbol H_k of the pattern, and $M(l,k)=0$ otherwise.*

(ii) *For a given pattern matching algorithm partial complexity function $C_{r,n}$ is defined as*

$$C_{r,s}(H_1^m, T_1^n) = \sum_{l\in[r,s],k\in[1,m]} M[l,k]$$

where $1 \le r < s \le n$. For $r=1$ and $s=n$ the function $C_{1,n} := C_n$ is simply called the **complexity** *of the algorithm.*

We illustrate some of the notions just introduced in the following example

Example 1.5 *Illustration to Definition 1.4.*

Let $H = abacabacabab$ and $T = abacabacabaaa$. The first mismatch occurs for $M(12,12)$. The comparisons performed from that point are:

1. **Morris-Pratt variant:**

$$(12,12); (12,8); (12,4); (12,2); (12,1); (13,2); (13,1) \quad ,$$

where the text character is compared in turn with pattern characters (b,c,c,b,a,b,a) with the alignment positions $(1,5,9,11,12,12,13)$.

2. **Knuth-Morris-Pratt variant:**

$$(12,12); (12,8); (12,2); (12,1); (13,2); (13,1) \quad ,$$

where the text character is compared in turn with pattern characters (b,c,b,a,b,a) with the alignment positions $(1,5,11,12,12,13)$.

3. **Simon variant:**

$$(12,12); (12,8); (12,1); (13,2); (13,1) \quad ,$$

where the text character is compared in turn with pattern characters (b,c,a,b,a) with the alignment positions $(1,5,12,12,13)$.

□

Some observations are in sequel: Morris-Pratt variant considers one alignment position at a time, while the optimal sequential algorithm, that of Simon, considers several alignment positions at the same time, and may disregard several of them simultaneously (e.g., in Example 3 positions 1 and 9 at the first step and 5 and 11 at the second step). It is interesting to observe that the subset $\{1,5,12\}$ appears in all variants. We will see that they share a common property of "unavoidability" explored in Section 5.5.2.

1.4 Shortest Common Superstring

Various versions of the **shortest common superstring** (in short: SCS) problem play important roles in data compression and DNA sequencing. In fact, in laboratories DNA sequencing (cf. [162, 247]) is routinely done by sequencing large numbers of relatively short fragments, and then heuristically finding a short common superstring. The problem can be formulated as follows: given a collection of strings, say $X^1, X^2, \ldots, X^n$ over an alphabet Σ, find the shortest string Z such that each of X^i appears as a substring (a consecutive block) of Z. In DNA sequencing, another formulation of the problem may be of even greater interest. We call it an *approximate* SCS and one asks for a superstring that contains *approximately* (e.g., in the Hamming distance sense) the original strings $X^1, X^2, \ldots, X^n$ as substrings.

More precisely, suppose $X = x_1 x_2 \ldots x_r$ and $Y = y_1 y_2 \ldots y_s$ are strings over the same finite alphabet Σ. We also write $|X|$ for the length of X. We define their *overlap* $o(X, Y)$ by

$$o(X, Y) = \max\{j : \; y_i = x_{r-j+i}, \; 1 \le i \le j\}.$$

If $X \ne Y$ and $k = o(X, Y)$, then

$$X \cdot Y = x_1 x_2 \ldots x_r y_{k+1} y_{k+2} \ldots y_s.$$

is the superstring of X and Y, where $\cdot$ is the concatenation operation. Let $\mathcal{S}$ be a set of all superstrings built over the strings $X^1, \ldots, X^n$. Then,

$$O_n^{\text{opt}} = \sum_{i=1}^{n} |X_i| - \min_{Z \in \mathcal{S}} |Z|$$

is the *optimal overlap* in the shortest common superstring.

Example 1.6: *Common Superstring and Its Graph Representation*

Let us consider the following five strings: $X^1 = abaaab$, $X^2 = aabaaaa$, $X^3 = aababb$, $X^4 = bbaaba$, and $X^5 = bbbb$. We first find C_{ij} that represents the length of the longest suffix of X^i that is also equal to a prefix of X^j. In our case

$$C = \begin{bmatrix} \star & 3 & 3 & 1 & 1 \\ 1 & \star & 2 & 0 & 0 \\ 0 & 0 & \star & 2 & 2 \\ 3 & 4 & 4 & \star & 0 \\ 0 & 0 & 0 & 2 & \star \end{bmatrix}$$

Let now $\mathcal{G}(C)$ be a weighted di-graph built on the set of strings $\{X^1, \ldots, X^5\}$ with weights C_{ij}. This di-graph is shown on Figure 1.7. Observe that the optimal (maximum) Hamiltonian path in $\mathcal{G}(C)$ determines the maximum overlap between strings $X^1, \ldots, X^5$. Hence, one can construct the shortest common superstring, which in our case is: $Z = X^4 \cdot X^1 \cdot X^2 \cdot X^3 \cdot X^5$. $\square$

From the above example, we should conclude that computing the shortest common superstring is NP-hard. Thus constructing a good approximation to SCS is of prime interest. It has been shown recently, that a greedy algorithm can compute in $O(n \log n)$ time a superstring that in the worst case is only β times (where $2 \le \beta \le 4$) longer than the shortest common superstring [31, 240]. Often, one is interested in maximizing total overlap of SCS using a greedy heuristic and to show that such a heuristic produces an overlap O_n^{gr} that approximates well the optimal overlap O_n^{opt} where n is the number of strings.

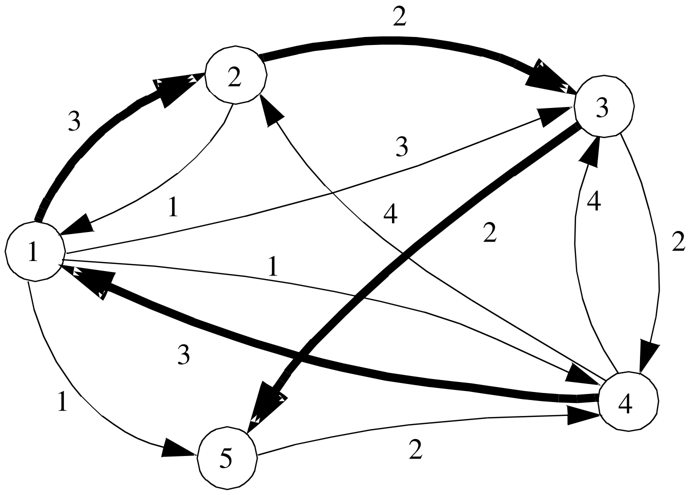

Figure 1.7: The di-graph $\mathcal{G}(C)$ from Example 6. Optimal Hamiltonian path (starting at node 4 is shown in bold.

A generic greedy algorithm for the SCS problem can be described as follows (cf. [31, 90, 240]: Its input is the n strings $X^1, X^2, \ldots, X^n$ over Σ. It outputs a string Z which is a superstring of the input.

Generic greedy algorithm

1. $I \leftarrow \{X^1, X^2, X^3, \ldots, X^n\}$; $O_n^{\text{gr}} \leftarrow 0$;
2. **repeat**
3. choose $X, Y \in I$; $Z = X \oplus Y$;
4. $I \leftarrow (I \setminus \{X, Y\}) \cup \{Z\}$;
5. $O_n^{\text{gr}} \leftarrow O_n^{\text{gr}} + o(X, Y)$;
6. **until** $|I| = 1$

Different variants of the above generic algorithm can be envisioned by interpreting appropriately the "choose" statement in Step 3 above:

RGREEDY: In Step 3, X is the string Z produced in the previous iteration, while Y is chosen in order to maximize $o(X, Y) = o(Z, Y)$. Our initial choice for X is X^1. Thus, in RGREEDY we have one "long" string Z which grows by addition of strings at the *right hand* *end.* In terms of the di-graph $\mathcal{G}(C)$, one starts from the vertex X_1 and follow the out-going edge with the maximum weight $C_{1,j}$. This process continues until all vertices of $\mathcal{G}(C)$ are visited disregarding cycles of length smaller than n.

GREEDY: Sort the edges of $\mathcal{G}_n(C)$ into $e_1, e_2, \dots, e_N$, $N = n^2$ so that $C(e_i) \geq C(e_{i+1})$ where $C(e_i)$ is the weight assigned to edge e_i; $S_G \leftarrow \emptyset$;

For $i = 1$ **to** N **do**:
if $S_G \cup \{e_i\}$ contains in $\mathcal{G}$ neither
(i) a vertex of outdegree or in–degree at least 2 in S_G,
(ii) a directed cycle,
then $S_G \leftarrow S_G \cup \{e_i\}$.

On termination S_G contains the $n-1$ edges of a Hamilton path of $\mathcal{G}$ and corresponds to a superstring of $X^1, X^2, \dots, X^n$. The selection of an edge weight (X^i, X^j) corresponds to overlapping X^i to the left of X^j.

MGREEDY: In Step 3 choose X, Y in order to maximize $o(X, Y)$. If $X \neq Y$ proceed as in GREEDY. If $X = Y$, then $I \leftarrow I \setminus \{X\}$, O_n^{gr} is not incremented, and $\mathcal{C} \leftarrow \mathcal{C} \cup \{X\}$ where the set $\mathcal{C}$ is initially empty. On termination we add the final string left in I to $\mathcal{C}$.

MGREEDY: sort the edges $\mathcal{G}$ into $e_1, e_2, \dots, e_N$, $N = n^2$ so that $w(e_i) \geq w(e_{i+1})$; $S_{MG}, C \leftarrow \emptyset$;
For $i = 1$ **to** N **do**:
if $S_{MG} \cup \{e_i\}$ contains no vertex of outdegree or indegree at least 2 in S_{MG},
then $S_{MG} \leftarrow S_{MG} \cup \{e_i\}$.
If e_i closes a cycle, **then** $C \leftarrow C \cup \{e_i\}$.

On termination the edges of S_{MG} form a collection of vertex disjoint cycles $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_t$, $t = |\mathcal{C}|$ which cover $[n]$. Each C_j contains one edge f_j which is a member of $\mathcal{C}$ and f_j is a lowest weight edge of $\mathcal{C}_j$. Let $\mathcal{P}_j = C_j - f_j$. The catenation of paths $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_t$ define a superstring of the input.

We shall analyze the shortest common superstring in Chapter 4 and Chapter 5.

1.5 String Editing Problem

The string editing problem arises in many applications, notably in text editing, speech recognition, machine vision and molecular sequence comparison (cf. [247]). Algorithmic aspects of this problem have been studied rather extensively in the past (cf. [18, 180, 207, 247]). In fact, many important problems on words are special cases of string editing, including the *longest common subsequence* problem (cf. [45, 38, 207]) and the problem of *approximate pattern matching* (cf. [45]). In the following, we review the string editing problem and its relationship to the longest path problem in a grid graph.

Let Y be a string consisting of ℓ symbols over the alphabet Σ of size V. There are three operations that can be performed on a string, namely *deletion* of a symbol, *insertion* of a symbol, and *substitution* of one symbol for another symbol in Σ. With each operation is associated a *weight* function. We denote by $W_I(y_i)$, $W_D(y_i)$ and $W_Q(x_i, y_j)$ the weight of insertion and deletion of the symbol $y_i \in \Sigma$, and substitution of x_i by $y_j \in \Sigma$, respectively. An *edit script* on Y is any sequence of edit operations, and the total weight is the sum of weights of the edit operations.

The **string editing** problem deals with two strings, say Y of length ℓ (for ℓong) and X of length s (for $s$hort), and consists of finding an edit script of minimum (maximum) total weight that transforms X into Y. The maximum (minimum) weight is called the *edit distance from X to Y*, and its is also known as the Levenshtein distance. In molecular biology, the Levenshtein distance is used to measure similarity (homogeneity) of two molecular sequences, say DNA sequences (cf. [207]).

The string edit problem can be solved by the standard dynamic programming method. Let $C_{\max}(i,j)$ denote the maximum weight of transforming the prefix of Y of size i into the prefix of X of size j. Then (cf. [18, 180, 247])

$$\begin{aligned} C_{\max}(i,j) &= \max\{C_{\max}(i-1,j-1) + W_Q(x_i,y_j), C_{\max}(i-1,j) \\ &+ W_D(x_i), C_{\max}(i,j-1) + W_I(y_j)\} \end{aligned}$$

for all $1 \le i \le \ell$ and $1 \le j \le s$. We compute $C_{\max}(i,j)$ row by row to obtain finally the total cost $C_{\max} = C_{\max}(\ell, s)$ of the maximum edit script. A similar procedure works for the minimum edit distance.

The edit distance can be conveniently represented on a grid graph. The key observation is to note that interdependency among the partial optimal weights $C_{\max}(i,j)$ induce an $\ell \times s$ grid-like directed acyclic graph, called further a *grid graph.* In such a graph vertices are points in the grid and edges go only from (i,j) point to the neighboring points, namely $(i,j+1)$, $(i+1,j)$ and $(i+1,j+1)$. A horizontal edge from $(i-1,j)$ to (i,j) carries the weight $W_I(y_j)$; a vertical edge from $(i,j-1)$ to (i,j) has weight $W_D(x_i)$; and finally a diagonal edge from $(i-1,j-1)$ to (i,j) is weighted according to $W_Q(x_i,y_j)$. Figure 1.8 shows an example of such an edit graph. The edit distance is the longest (shortest) path from the point $O = (0,0)$ to $E = (\ell, s)$.

Finally, we should mention that by selecting properly the distributions of W_I, W_D and W_Q we can model several variations of the string editing problem. For example, in the standard setting the deletion and insertion weights are identical, and usually constant, while the substitution weight takes two values, one (high) when matching between a letter of X and a letter of Y occurs, and another value (low) in the case of a mismatch (e.g., in the *Longest Common Substring* problem [38, 207], one sets $W_I = W_D = 0$, and $W_Q = 1$ when a matching occurs, and $W_Q = -\infty$ for a mismatch).

We shall analyze pattern matching problems and the edit distance problem in Chapter 5 and Chapter 7.

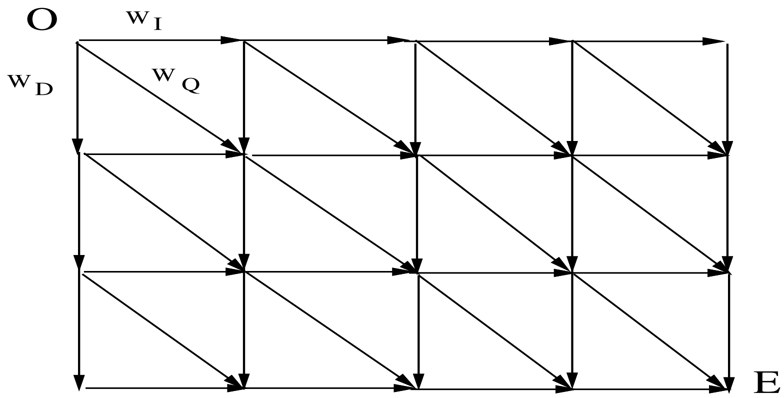

Figure 1.8: Example of a grid graph of size $\ell = 4$ and $s = 3$.

1.6 Optimization Problems

The last two sections shown that combinatorial optimization problems on graphs arise often in the design and analysis of algorithms and data structures on strings. Undoubtly, these problems are also a source of exciting and interesting probabilistic problems. Therefore, in this last section we present a fairly general description of optimization problems.

We consider a class of optimization problems that can be formulated as follows: Let n be an integer (e.g., number of vertices in a graph, size of a matrix, number of strings in a digital tree, etc.), and $\mathcal{S}_n$ a set of objects (e.g., set of vertices, elements of a matrix, strings, etc). We shall investigate the behavior of the optimal values $Z_{\max}(\mathcal{S}_n)$ and $Z_{\min}(\mathcal{S}_n)$ defined as follows

$$Z_{\max}(\mathcal{S}_n) = \max_{\alpha \in \mathcal{B}_n} \left\{ \sum_{i \in \mathcal{S}_n(\alpha)} w_i(\alpha) \right\} \quad , \quad Z_{\min}(\mathcal{S}_n) = \min_{\alpha \in \mathcal{B}_n} \left\{ \sum_{i \in \mathcal{S}_n(\alpha)} w_i(\alpha) \right\} , \tag{1.8}$$

and

$$Z_{\max}(\mathcal{S}_n) = \max_{\alpha \in \mathcal{B}_n} \left\{ \min_{i \in \mathcal{S}_n(\alpha)} w_i(\alpha) \right\} \quad , \quad Z_{\min}(\mathcal{S}_n) = \min_{\alpha \in \mathcal{B}_n} \left\{ \max_{i \in \mathcal{S}_n(\alpha)} w_i(\alpha) \right\} \tag{1.9}$$

where $\mathcal{B}_n$ is a set of all feasible solutions, $\mathcal{S}_n(\alpha)$ is a set of objects from $\mathcal{S}_n$ belonging to the α-th feasible solution, and $w_i(\alpha)$ is the weight assigned to the i-th object in the α-th feasible solution. We often write $Z_{\max}$ and $Z_{\min}$ instead of $Z_{\max}(\mathcal{S}_n)$ and $Z_{\min}(\mathcal{S}_n)$, respectively.

For example, in the traveling salesman problem, $\mathcal{B}_n$ represents the set of all Hamiltonian paths in a graph built over n vertices, $\mathcal{S}_n(\alpha)$ is the set of edges belonging to the α-th Hamiltonian path, and $w_i(\alpha)$ is the length (weight) of the i-th edge. Traditionally, formulation (1.8) is called the *optimization problem with sum-objective function*, while (1.9) is either known as the *capacity optimization problem* ($Z_{\max}$) or the *bottleneck optimization problem* ($Z_{\min}$).

Certainly, combinatorial optimization problems arise in many areas of science and engineering. Among others we mention here: the (capacity and bottleneck) assignment problem, the (bottleneck and capacity) quadratic assignment problem, the minimum spanning tree, the minimum weighted k-clique problem, geometric location problems, and some others not directly related to optimization (but of interest to us) such as the height and depth of digital trees, the maximum queue length and hashing with lazy deletion [4], pattern matching [19], shortest common superstring, longest common subsequence, and so forth.

We return to these optimization problems in Chapter 5.

1.7 Exercises

1.1 Prove (1.4)–(1.7) of Theorem 1.3.

1.2 Establish combinatorial relationships between PATRICIA and digital trees parameters and the alignments $\{C_{ij}\}_{ij=1}^{n}$, as we did in Theorem 1.3 for tries.

1.3 Parse and build Lempel-Ziv'77 and Lempel-Ziv'78 codes for the following sequence: $X = ababbbaabbbabbbbababbb$. How the parsing and the code will change when generalized LZ'78 is used with $b = 4$.

1.4 Fix an integer $b \geq 1$. Generalize Lempel-Ziv'77 is a similar manner as we generalized Lempel-Ziv'78 in Section 1.2.3.

1.5 We consider here the FDLZ, that is, the fixed database version of the Lempel-Ziv scheme. Let the database sequence be denoted as $\hat{X}_1^n$ while the source sequence be X_1^M. The source sequence is partitions into phrases of length $I_1, I_2, \ldots, I_K$ where K is the total number of phrases (i.e., $I_1 + \cdots I_K = M$). Show that the compression ratio $r_n(X_1^M)$ can be expressed as

$$r_n(X_1^M) = \frac{1}{M}\sum_{i=1}^{K} \log_2 n + O(\log I_i).$$

1.6 Given a sequence of length n, derive a closed-form formulæ for the code lengths of the LZ'77, LZ'78 and the generalized-LZ'78 schemes in terms of the number of phrase M_n, size of the alphabet V, and the parameter b.

1.7 Construct the compression code for $X = ababbbaabbbabbbbababbb$ using the lossy LZ'77 with $D = 0.3$ assuming Hamming distance.

1.8 Compute the complexity C_n introduced in Def. 1.4 for $H = abab$ and $T = abbbbabababbbab$ when KMP algorithm is used to find occurrences of H in T.

1.9 Construct the optimal and the three greedy heuristics RGREEDY, GREEDY and MGREEDY for the common superstring of $X^1 = ababababa$, $X^2 = bbbbbbbb$, $X^3 = aaaaaaaaa$, $X^4 = babbbaaa$ and $X^5 = bbaabbaabbaa$.

1.10 Find the longest common subsequence of $X = abcccabbbbbabab$ and $Y = cabbbabababb$ over $\Sigma = \{a, b, c\}$.

1.11 Determine the feasible set $\mathcal{B}_n$ and set of objects $\mathcal{S}_n$ for the shortest common superstring problem and the longest common subsequence problem.