

Accelerating Dynamic Programming

by

Oren Weimann

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

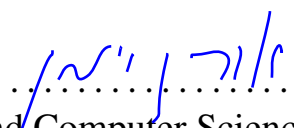
Doctor of Philosophy


at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

© Massachusetts Institute of Technology 2009. All rights reserved.

Author 
Department of Electrical Engineering and Computer Science
February 5, 2009

Certified by 
Erik D. Demaine
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by
Professor Terry P. Orlando
Chairman, Department Committee on Graduate Students

Accelerating Dynamic Programming

by

Oren Weimann

Submitted to the Department of Electrical Engineering and Computer Science
on February 5, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Dynamic Programming (DP) is a fundamental problem-solving technique that has been widely used for solving a broad range of search and optimization problems. While DP can be invoked when more specialized methods fail, this generality often incurs a cost in efficiency. We explore a unifying toolkit for speeding up DP, and algorithms that use DP as subroutines. Our methods and results can be summarized as follows.

- *Acceleration via Compression.* Compression is traditionally used to efficiently store data. We use compression in order to identify repeats in the table that imply a redundant computation. Utilizing these repeats requires a new DP, and often different DPs for different compression schemes. We present the first provable speedup of the celebrated Viterbi algorithm (1967) that is used for the decoding and training of Hidden Markov Models (HMMs). Our speedup relies on the compression of the HMM’s observable sequence.
- *Totally Monotone Matrices.* It is well known that a wide variety of DPs can be reduced to the problem of finding row minima in totally monotone matrices. We introduce this scheme in the context of planar graph problems. In particular, we show that planar graph problems such as shortest paths, feasible flow, bipartite perfect matching, and replacement paths can be accelerated by DPs that exploit a total-monotonicity property of the shortest paths.
- *Combining Compression and Total Monotonicity.* We introduce a method for accelerating string edit distance computation by combining compression and totally monotone matrices. In the heart of this method are algorithms for computing the edit distance between two straight-line programs. These enable us to exploit the compressibility of strings, even if each string is compressed using a different compression scheme.
- *Partial Tables.* In typical DP settings, a table is filled in its entirety, where each cell corresponds to some subproblem. In some cases, by changing the DP, it is possible to compute asymptotically less cells of the table. We show that $\Theta(n^3)$ subproblems are both necessary and sufficient for computing the similarity between two trees. This improves all known solutions and brings the idea of partial tables to its full extent.

- *Fractional Subproblems.* In some DPs, the solution to a subproblem is a data structure rather than a single value. The entire data structure of a subproblem is then processed and used to construct the data structure of larger subproblems. We suggest a method for reusing parts of a subproblem’s data structure. In some cases, such fractional parts remain unchanged when constructing the data structure of larger subproblems. In these cases, it is possible to copy this part of the data structure to the larger subproblem using only a constant number of pointer changes. We show how this idea can be used for finding the optimal tree searching strategy in linear time. This is a generalization of the well known binary search technique from arrays to trees.

Thesis Supervisor: Erik D. Demaine

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I am deeply indebted to my thesis advisor, Professor Erik Demaine. Erik's broad research interests and brilliant mind have made it possible to discuss and collaborate with him on practically any problem of algorithmic nature. His open-minded view of research, and his support in my research which was sometimes carried outside MIT, have provided me with the perfect research environment. I feel privileged to have learned so much from Erik, especially on the subject of data structures.

I owe much gratitude to Professor Gad Landau. His guidance, kindness, and friendship during my years in MIT and my years in Haifa have been indispensable. I am also thankful to the MADALGO research center and the Caesarea Rothschild Institute for providing me with additional working environments and financial support.

This thesis is based primarily on the following publications: [31] from ICALP'07, [12] from JCB, [80] from SODA'08, [63] from Algorithmica, [52] from SODA'09, and [45] from STACS'09. I would like to thank my collaborators in these publications. In particular, I am grateful for my ongoing collaboration with Shay Mozes and Danny Hermelin, who make it possible for me to conduct research together with close friends.

Finally, I wish to express my love and gratitude to Gabriela, my parents, my grandparents, and my sister. I am lucky to have been blessed with their constant support and endless love.

Contents

1	Introduction	13
1.1	Searching for DP Repeats	14
1.2	Totally Monotone Matrices	16
1.3	Partial Tables and Fractional Subproblems	18
2	Acceleration via Compression	21
2.1	Decoding and Training Hidden Markov Models	22
2.1.1	The Viterbi Algorithm	23
2.1.2	The Forward-Backward Algorithms	25
2.2	Exploiting Repeated Substrings	25
2.3	Five Different Implementations of the General Framework	27
2.3.1	Acceleration via the Four Russians Method	27
2.3.2	Acceleration via Run-length Encoding	28
2.3.3	Acceleration via LZ78 Parsing	29
2.3.4	Acceleration via Straight-Line Programs	32
2.3.5	Acceleration via Byte-Pair Encoding	34
2.4	Optimal state-path recovery	35
2.5	The Training Problem	36
2.5.1	Viterbi training	37
2.5.2	Baum-Welch training	37
2.6	Parallelization	40

3	Totally Monotone Matrices and Monge Matrices	43
3.1	Preliminaries	46
3.1.1	Monotonicity, Monge and Matrix Searching.	46
3.1.2	Jordan Separators for Embedded Planar Graphs.	48
3.2	A Bellman-Ford Variant for Planar Graphs	49
3.3	The Replacement-Paths Problem in Planar Graphs	54
4	Combining Compression and Total-Monotonicity	59
4.1	Accelerating String Edit Distance Computation	60
4.1.1	Straight-line programs	61
4.1.2	Our results	62
4.1.3	Related Work	63
4.2	The DIST Table and Total-Monotonicity	64
4.3	Acceleration via Straight-Line Programs	66
4.3.1	Constructing an xy -partition	69
4.4	Improvement for Rational Scoring Functions	71
4.5	Four-Russian Interpretation	72
5	Partial Tables	75
5.1	Tree Edit Distance	76
5.1.1	Shasha and Zhang’s Algorithm	79
5.1.2	Klein’s Algorithm	80
5.1.3	The Decomposition Strategy Framework	81
5.2	Our Tree Edit Distance Algorithm	83
5.3	A Space-Efficient DP formulation	88
5.4	A Tight Lower Bound for Decomposition Algorithms	95
5.5	Tree Edit Distance for RNA Comparison	99
5.5.1	RNA Alignment	101
5.5.2	Affine Gap Penalties	102

6 Fractional Subproblems	105
6.1 Finding an Optimal Tree Searching Strategy in Linear Time	106
6.2 Machinery for Solving Tree Searching Problems	109
6.3 Computing a Minimizing Extension	111
6.3.1 Algorithm Description.	111
6.4 Linear Time Solution via Fractional Subproblems	118
6.5 From a Strategy Function to a Decision Tree in Linear Time	125
Bibliography	136

List of Figures

2-1	An HMM and the optimal path of hidden states	23
2-2	The Viterbi algorithm	24
2-3	The Fibonacci straight-line program (SLP)	32
2-4	A “good” substring in the Baum-Welch training	39
3-1	The Bellman-Ford algorithm	49
3-2	Decomposing a planar graph using a Jordan curve	50
3-3	Decomposing a shortest path into external and internal paths	50
3-4	Pseudocode for the single-source boundary distances	51
3-5	Crossing paths between boundary nodes	53
3-6	Two crossing LL replacement paths	55
3-7	Two crossing LR replacement paths	55
3-8	The upper triangular fragment of the matrix $\widehat{\text{len}}_{d,d'}$	57
4-1	A subgraph of a Levenshtein distance DP graph	64
4-2	An xy -partition.	67
4-3	A closer look on the parse tree of an SLP \mathcal{A}	70
5-1	The three editing operations on a tree	77
5-2	A tree F with its heavy path and $\text{TopLight}(F)$ vertices	84
5-3	The intermediate left subforest enumeration	90
5-4	The indexing of subforests $G_{i,j}$	91
5-5	The two trees used to prove an $\Omega(m^2n)$ lower bound.	96

5-6	The two trees used to prove an $\Omega(m^2n \log \frac{n}{m})$ lower bound.	97
5-7	Three different ways of viewing an RNA sequence	100
5-8	Two example alignments for a pair of RNA sequences.	102
6-1	An optimal strategy for searching a tree	110
6-2	Algorithm for computing a minimizing extension	112

Chapter 1

Introduction

Dynamic Programming (DP) is a powerful problem-solving paradigm in which a problem is solved by breaking it down into smaller subproblems. These subproblems are then tackled one by one, so that the answers to small problems are used to solve the larger ones. The simplicity of the DP paradigm, as well as its broad applicability have made it a fundamental technique for solving various search and optimization problems.

The word “programming” in “dynamic programming” has actually very little to do with computer programming and writing code. The term was first coined by Richard Bellman in the 1950s, back when programming meant planning, and dynamic programming meant to optimally plan a solution process. Indeed, the challenge of devising a good solution process is in deciding what are the subproblems, and in what order they should be computed. Apart from the obvious requirement that an optimal solution to a problem can be obtained by optimal solutions to its subproblems, an efficient DP is one that induces only a “small” number of distinct subproblems. Each subproblem is then reused again and again for solving multiple larger problems.

This idea of reusing subproblems is the main advantage of the DP paradigm over recursion. Recursion is suited for design techniques such as divide-and-conquer, where a problem is reduced to subproblems that are substantially smaller, say half the size. In contrast, in a typical DP setting, a problem is reduced to subproblems that are only slightly smaller (e.g., smaller by only a constant factor). However, it is often convenient to write out a top-down recursive formula and then use

it to describe the corresponding bottom-up DP solution. In many cases, this transformation is not immediate since in the final DP we can often achieve a space complexity that is asymptotically smaller than the total number of subproblems. This is enhanced by the fact that we only need to store the answer of a subproblem until the larger subproblems depending on it have been solved.

The simplicity of the DP paradigm is what makes it appealing, both as a full problem solving method as well as a subroutine solver in more complicated algorithmic solutions. However, while DP is useful when more specialized methods fail, this generality often incurs a cost in efficiency. In this thesis, we explore a toolkit for speeding up straightforward DPs, and algorithms that use DPs as subroutines.

To illustrate some DP ideas, consider the *string edit distance* problem. This problem asks to compute the minimum number of character deletions, insertions, and replacements required to transform one string $A = a_1a_2 \cdots a_n$ into another string $B = b_1b_2 \cdots b_m$. Let $E(i, j)$ denote the edit distance between $a_1a_2 \cdots a_i$ and $b_1b_2 \cdots b_j$. Our goal is then to compute $E(n, m)$, and we can achieve this by using the following DP.

$$E(i, j) = \min\{E(i, j - 1) + 1, E(i - 1, j) + 1, E(i - 1, j - 1) + \text{diff}(i, j)\}$$

where $\text{diff}(i, j) = 0$ if $a_i = b_j$, and 1 otherwise. This standard textbook DP induces $O(nm)$ subproblems – one subproblem for each pair of prefixes of A and B . Since every subproblem requires the answers to three smaller subproblems, the time complexity is $O(nm)$ and the bottom-up DP can construct the table E row by row. Notice that at any stage during the computation, we only need to store the values of two consecutive rows, so this is a good example of a situation where the space complexity is smaller than the time complexity (the number of subproblems).

1.1 Searching for DP Repeats

As we already mentioned, a good DP solution is one that makes the most out of subproblem repeats. Indeed, if we look at a subproblem $E(i, j)$, it is used for solving the larger subproblem $E(i, j + 1)$, as well as $E(i + 1, j)$ and $E(i + 1, j + 1)$. However, does the above DP really utilize

repeats to their full extent? What if the string $A = A'A'$ (i.e., A is the concatenation of two equal strings of length $n/2$), and $B = B'B'$. The DP computation will then behave “similarly” on $E(i, j)$ and on $E(i + n/2, j + m/2)$ for every $i = 1, \dots, n/2$ and $j = 1, \dots, m/2$. In order for repeats in the strings to actually induce repeated subproblems we would have to modify the DP so that subproblems correspond to substrings rather than to prefixes of A and B .

The first paper to do this and break the quadratic time upper bound of the edit distance computation was the seminal paper of Masek and Paterson [72]. This was achieved by applying the so called “Four-Russians technique” – a method based on a paper by Arlazarov, Dinic, Kronrod, and Faradzev [10] for boolean matrix multiplication. The general idea is to first pre-compute the subproblems that correspond to all possible substrings (blocks) of logarithmic size, and then modify the edit distance DP so instead of advancing character by character it advances block by block. This gives a logarithmic speedup over the above $O(nm)$ solution.

The Masek and Paterson speedup is based on the fact that in strings over constant-sized alphabets, small enough substrings are guaranteed to repeat. But what about utilizing repeats of longer substrings? It turns out that we can use text compression to both find and utilize repeats. The approach of using compression to identify DP repeats, denoted “acceleration via compression”, has been successfully applied to many classical string problems such as exact string matching [5, 49, 61, 70, 91], approximate pattern matching [4, 48, 49, 82], and string edit distance [19, 8, 9, 29, 69, 29]. It is often the case that utilizing these repeats requires an entirely new (and more complicated) DP.

It is important to note, that all known improvements on the $O(nm)$ upper bound of the edit distance computation, apply acceleration via compression. In addition, apart from the naive compression of the Four-Russians technique, Run Length Encoding, and the LZW-LZ78 compression, we do not know how to compute the edit distance efficiently under other compression schemes.

Our string edit distance result. We introduce a general compression-based edit distance algorithm that can exploit the compressibility of two strings under *any* compression scheme, even if each string is compressed with a different compression. This is achieved by using Straight-line programs (a notion borrowed from the world of formal languages), together with an efficient

algorithm for finding row minima in totally monotone matrices. We describe the use of totally monotone matrices for accelerating DPs in Section 1.2.

Our HMM result. Hidden Markov Models (HMMs) are an extremely useful way of modeling processes in diverse areas such as error-correction in communication links [99], speech recognition [18], optical character recognition [1], computational linguistics [71], and bioinformatics [33]. The two most important computational problems involving HMMs are decoding and training. The problem of decoding an HMM asks to find the most probable sequence of hidden states to have generated some observable sequence, and the training problem asks to estimate the model given only the number of hidden states and an observable sequence.

In Chapter 2, we apply acceleration via compression to the DP algorithms used for decoding and training HMMs. We discuss the application of our method to Viterbi’s decoding and training algorithms [99], as well as to the forward-backward and Baum-Welch [13] algorithms. We obtain the first provable speedup of the celebrated Viterbi algorithm [99]. Compared to Viterbi’s algorithm, we achieve speedups of $\Theta(\log n)$ using the Four-Russians method, $\Omega(\frac{r}{\log r})$ using run-length encoding, $\Omega(\frac{\log n}{k})$ using Lempel-Ziv parsing, $\Omega(\frac{r}{k})$ using straight-line programs, and $\Omega(r)$ using byte-pair encoding, where k is the number of hidden states, n is the length of the observed sequence and r is its compression ratio (under each compression scheme).

1.2 Totally Monotone Matrices

One of the best known DP speedup techniques is the seminal algorithm of Aggarwal, Klawe, Moran, Shor, and Wilber [3] nicknamed SMAWK in the literature. This is a general speedup technique that can be invoked when the DP satisfies additional conditions of convexity or concavity that can be described by a totally monotone matrix. An $n \times m$ matrix $M = (M_{ij})$ is *totally monotone* if for every i, i', j, j' such that $i < i', j < j'$, and $M_{ij} \leq M_{i'j'}$, we have that $M_{i'j} \leq M_{ij'}$. Aggarwal et al. showed that a wide variety of problems in computational geometry can be reduced to the problem of finding row minima in totally monotone matrices. The SMAWK algorithm finds all row minima optimally in $O(n + m)$ time. Since then, many papers on applications which lead

to totally monotone matrices have been published (see [20] for a survey).

Our planar shortest paths result. In Chapter 3, we use totally monotone matrices in the context of planar graph problems. We take advantage of the fact that shortest paths in planar graphs exhibit a property similar to total-monotonicity. Namely, they can be described by an upper-triangular fragment of a totally monotone matrix. Our first result is an $O(n\alpha(n))$ time¹ DP for performing a variant of the Bellman-Ford shortest paths algorithm on the vertices of the planar separator. This DP serves as a subroutine in all single-source shortest paths algorithms for directed planar graphs with negative edge lengths. The previous best DP for this subroutine is that of Fakcharoenphol and Rao [35] and requires $O(n \log^2 n)$ time.

Fakcharoenphol and Rao used this subroutine (and much more) to obtain an $O(n \log^3 n)$ -time and $O(n \log n)$ -space solution for the single-source shortest paths problem in directed planar graphs with negative lengths. In [52], we used our new DP to improve this to $O(n \log^2 n)$ time and $O(n)$ space. This result is important not only for shortest paths, but also for solving bipartite perfect matching, feasible flow, and feasible circulation which are all equivalent in planar graphs to single-source shortest paths with negative lengths. Apart from the DP, we needed other non-DP ideas that are beyond the scope of this thesis. We therefore focus here only on the said Bellman-Ford variant.

Our planar replacement paths result. Our second result involving total-monotonicity in planar graphs concerns the *replacement-paths* problem. In this problem, we are given a directed graph with non-negative edge lengths and two nodes s and t , and we are required to compute, for every edge e in the shortest path between s and t , the length of an s -to- t shortest path that avoids e . By exploiting total-monotonicity, we show how to improve the previous best $O(n \log^3 n)$ -time solution of Emek et al. [34] for the replacement path problem in planar graphs to $O(n \log^2 n)$.

Finally, as we already mentioned, by combining acceleration via compression, and the totally monotone properties of the string edit distance DP, we show in Chapter 4 how to accelerate the string edit distance computation.

¹ $\alpha(n)$ denotes the slowly growing inverse Ackerman function.

1.3 Partial Tables and Fractional Subproblems

The main advantage of (top-down) recursion over (bottom-up) DP is that while DP usually solves every subproblem that could conceivably be needed, recursion only solves the ones that are actually used. We refer to these as *relevant subproblems*. To see this, consider the *Longest Common Subsequence* (LCS) problem that is very similar to string edit distance. The LCS of two strings $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$ is the longest string that can be obtained from both A and B by deleting characters. As in the edit distance case, we let $L(i, j)$ denote the length of the LCS between $a_1a_2 \cdots a_i$ and $b_1b_2 \cdots b_j$, and we compute $L(n, m)$ using the following DP.

$$L(i, j) = \begin{cases} L(i-1, j-1) + 1, & \text{if } a_i = b_j \\ \max\{L(i, j-1) + 1, L(i-1, j) + 1\}, & \text{otherwise} \end{cases}$$

Like the edit distance DP, the LCS DP will solve $O(nm)$ subproblems, one for each pair of prefixes of A and B . However, consider a simple scenario in which $A = B$. In this case, a recursive implementation will only compute $O(n)$ distinct (relevant) subproblems (one for each pair of prefixes of A and B of the same length), while DP will still require $\Omega(n^2)$. To avoid computing the same subproblem again and again, one could use *memoization* – a recursive implementation that remembers its previous invocations (using a hash table for example) and thereby avoids repeating them.

There are two important things to observe from the above discussion. First, that in the worst case (for general strings A and B), both the recursion (or memoization) and the DP compute $\Omega(n^2)$ distinct subproblems. Second, the memoization solution does not allow reducing the space complexity. That is, the space complexity using memoization is equal to the number of subproblems, while in DP it can be reduced to $O(n)$ even when the number of relevant subproblems is $\Omega(n^2)$.

Surprisingly, in some problems (unfortunately not in LCS), these two obstacles can be overcome. For such problems, it is possible to slightly change their original DP, so that it will fill only a partial portion of the DP table (i.e., only the subset of relevant subproblems). It can then be shown that even in the worst case, this partial portion is asymptotically smaller than the entire table. Furthermore, the space complexity can be reduced to be even smaller than the size of this

partial portion.

The main advantage of the partial table idea is that a slight change in the DP can make a big change in the number of subproblems that it computes. While the analysis can be complicated, the actual DP remains simple to describe and implement. This will be the topic of Chapter 5.

Our tree edit distance result. In Chapter 5, we will apply the above idea to the problem of *tree edit distance*. This problem occurs in various areas where the similarity between trees is sought. These include structured text databases like XML, computer vision, compiler optimization, natural language processing, and computational biology [15, 25, 54, 88, 95].

The well known DP solution of Shasha and Zhang [88] to the tree edit distance problem involves filling an $O(n^4)$ -sized table. Klein [53], showed that a small change to this DP induces only $O(n^3 \log n)$ relevant subproblems. We show that $\Theta(n^3)$ subproblems are both necessary and sufficient for computing the tree edit distance DP. This brings the idea of filling only a partial DP table to its full extent. We further show how to reduce the space complexity to $O(n^2)$.

After describing the idea of partial tables, the final chapter of this thesis deals with partial subproblems. Notice that in both DPs for edit distance and LCS that we have seen so far, the answer to a subproblem is a single value. In some DPs however, the solution to a subproblem is an entire data structure rather than only a value. In these cases, the DP solution processes the entire data structure computed for a certain subproblem in order to construct the data structure of larger subproblems. In Chapter 6, we discuss processing only parts of a subproblem's data structure. We show that in some cases, a part of a subproblem's data structure remains unchanged when computing the data structure of larger subproblems. In these cases, it is possible to copy this part of the data structure to the larger subproblem using only a constant number of pointer changes.

Our tree searching result. We show how this idea can be used for the problem of finding an optimal tree searching strategy. This problem is an extension of the binary search technique from sorted arrays to trees, with applications in file system synchronization and software testing [14, 57, 83]. As in the sorted array case, the goal is to minimize the number of queries required to find a target element in the worst case. However, while the optimal strategy for searching an array

is straightforward (always query the middle element), the optimal strategy for searching a tree is dependent on the tree's structure and is harder to compute. We give an $O(n)$ time DP solution that uses the idea of fractional subproblems and improves the previous best $O(n^3)$ -time algorithm of Onak and Parys [83].

Chapter 2

Acceleration via Compression

The main idea of DP is that subproblems that are encountered many times during the computation are solved once, and then used multiple times. In this chapter, we take this idea another step forward by considering entire regions of the DP table that appear more than once. This is achieved by first changing the DP in a way that regional repeats are guaranteed to occur, and then using text compression as a tool to find them.

The traditional aim of text compression is the efficient use of resources such as storage and bandwidth. The approach of using compression in DP to identify repeats, denoted “acceleration via compression”, has been successfully applied to many classical string problems. Various compression schemes, such as Lempel-Ziv [104, 103], Huffman coding, Byte-Pair Encoding [90], Run-Length Encoding, and Straight-Line Programs were employed to accelerate exact string matching [5, 49, 61, 70, 91], approximate pattern matching [4, 48, 49, 82], and string edit distance [8, 9, 19, 29, 69].

The acceleration via compression technique is designed for DPs that take strings as inputs. By compressing these input strings, it is possible to identify substring repeats that possibly imply regional repeats in the DP itself. In this chapter, we investigate this technique on a problem that is not considered a typical string problem as its input does not consist only of strings. We present a method to speed up the DPs used for solving Hidden Markov Model (HMM) decoding and training problems. Our approach is based on identifying repeated substrings in the observed input sequence.

We discuss the application of our method to Viterbi’s decoding and training algorithms [99], as well as to the forward-backward and Baum-Welch [13] algorithms.

In Section 2.1 we describe HMMs, and give a unified presentation of the HMM DPs. Then, in Section 2.2 we show how these DPs can be improved by identifying repeated substrings. Five different implementations of this general idea are presented in Section 2.3: Initially, we show how to exploit repetitions of all sufficiently small substrings (the Four-Russians method). Then, we describe four algorithms based alternatively on Run-Length Encoding (RLE), Lempel-Ziv (LZ78), Straight-Line Programs (SLP), and Byte-Pair Encoding (BPE). Compared to Viterbi’s algorithm, we achieve speedups of $\Theta(\log n)$ using the Four Russians method, $\Omega(\frac{r}{\log r})$ using RLE, $\Omega(\frac{\log n}{k})$ using LZ78, $\Omega(\frac{r}{k})$ using SLP, and $\Omega(r)$ using BPE, where k is the number of hidden states, n is the length of the observed sequence and r is its compression ratio (under each compression scheme). We discuss the recovery of the optimal state-path in Section 2.4, and the adaptation of our algorithms to the training problem in Section 2.5. Finally, in Section 2.6 we describe a parallel implementation of our algorithms.

2.1 Decoding and Training Hidden Markov Models

Over the last few decades HMMs proved to be an extremely useful framework for modeling processes in diverse areas such as error-correction in communication links [99], speech recognition [18], optical character recognition [1], computational linguistics [71], and bioinformatics [33].

Definition 2.1 (Hidden Markov Model) *Let Σ denote a finite alphabet and let $X \in \Sigma^n$, $X = x_1, x_2, \dots, x_n$ be a sequence of observed letters. A Markov model is a set of k states, along with emission probabilities $e_k(\sigma)$ - the probability to observe $\sigma \in \Sigma$ given that the state is k , and transition probabilities $P_{i,j}$ - the probability to make a transition to state i from state j .*

The core HMM-based applications fall in the domain of classification methods and are technically divided into two stages: a training stage and a decoding stage. During the *training* stage, the emission and transition probabilities of an HMM are estimated, based on an input set of observed sequences. This stage is usually executed once as a preprocessing stage and the generated

(“trained”) models are stored in a database. Then, a *decoding* stage is run, again and again, in order to classify input sequences. The objective of this stage is to find the most probable sequence of states to have generated each input sequence given each model, as illustrated in Fig. 2-1.

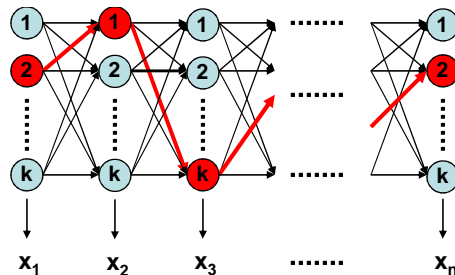


Figure 2-1: The HMM on the observed sequence $X = x_1, x_2, \dots, x_n$ and states $1, 2, \dots, k$. The highlighted path is a possible path of states that generate the observed sequence. VA finds the path with highest probability.

Obviously, the training problem is more difficult to solve than the decoding problem. However, the techniques used for decoding serve as basic ingredients in solving the training problem. The Viterbi algorithm (VA) [99] is the best known tool for solving the decoding problem. Following its invention in 1967, several other algorithms have been devised for the decoding and training problems, such as the forward-backward and Baum-Welch [13] algorithms. These algorithms are all based on DP and their running times depend linearly on the length of the observed sequence. The challenge of speeding up VA by utilizing HMM topology was posed in 1997 by Buchsbaum and Giancarlo [18] as a major open problem. We address this open problem by using text compression and present the first provable speedup of these algorithms.

2.1.1 The Viterbi Algorithm

The Viterbi algorithm (VA) finds the most probable sequence of hidden states given the model and the observed sequence, i.e., the sequence of states s_1, s_2, \dots, s_n which maximize

$$\prod_{i=1}^n e_{s_i}(x_i) P_{s_i, s_{i-1}} \quad (2.1)$$

The DP of VA calculates a vector $v_t[i]$ which is the probability of the most probable sequence of states emitting x_1, \dots, x_t and ending with the state i at time t . v_0 is usually taken to be the vector of uniform probabilities (i.e., $v_0[i] = \frac{1}{k}$). v_{t+1} is calculated from v_t according to

$$v_{t+1}[i] = e_i(x_{t+1}) \cdot \max_j \{P_{i,j} \cdot v_t[j]\} \quad (2.2)$$

Definition 2.2 (Viterbi Step) We call the computation of v_{t+1} from v_t a Viterbi step.

Clearly, each Viterbi step requires $O(k^2)$ time. Therefore, the total runtime required to compute the vector v_n is $O(nk^2)$. The probability of the most likely sequence of states is the maximal element in v_n . The actual sequence of states can then be reconstructed in linear time.

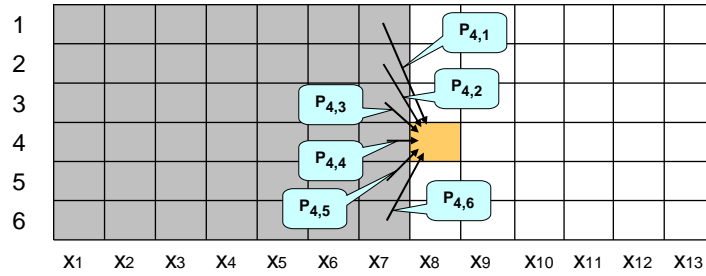


Figure 2-2: The VA dynamic programming table on sequence $X = x_1, x_2, \dots, x_{13}$ and states 1, 2, 3, 4, 5, 6. The marked cell corresponds to $v_8[4] = e_4(x_8) \cdot \max\{P_{4,1} \cdot v_7[1], P_{4,2} \cdot v_7[2], \dots, P_{4,6} \cdot v_7[6]\}$.

It is useful for our purposes to rewrite VA in a slightly different way. Let M^σ be a $k \times k$ matrix with elements $M_{i,j}^\sigma = e_i(\sigma) \cdot P_{i,j}$. We can now express v_n as:

$$v_n = M^{x_n} \odot M^{x_{n-1}} \odot \dots \odot M^{x_2} \odot M^{x_1} \odot v_0 \quad (2.3)$$

where $(A \odot B)_{i,j} = \max_k \{A_{i,k} \cdot B_{k,j}\}$ is the so called max-times matrix multiplication. Similar notation was already considered in the past. In [37], for example, writing VA as a linear vector recursion allowed the authors to employ parallel processing and pipelining techniques in the context of VLSI and systolic arrays.

VA computes v_n using (2.3) from right to left in $O(nk^2)$ time. Notice that if (2.3) is evaluated from left to right the computation would take $O(nk^3)$ time (matrix-vector multiplication vs. matrix-matrix multiplication). Throughout, we assume that the max-times matrix-matrix multiplications are done naïvely in $O(k^3)$. Faster methods for max-times matrix multiplication [22, 23] and standard matrix multiplication [28, 94] can be used to reduce the k^3 term. However, for small values of k this is not profitable.

2.1.2 The Forward-Backward Algorithms

The *forward-backward* algorithms are closely related to VA and are based on very similar DP. In contrast to VA, these algorithms apply standard matrix multiplication instead of max-times multiplication. The forward algorithm calculates $f_t[i]$, the probability to observe the sequence x_1, x_2, \dots, x_t requiring that $s_t = i$ as follows:

$$f_t = M^{x_t} \cdot M^{x_{t-1}} \cdot \dots \cdot M^{x_2} \cdot M^{x_1} \cdot f_0 \quad (2.4)$$

The backward algorithm calculates $b_t[i]$, the probability to observe the sequence $x_{t+1}, x_{t+2}, \dots, x_n$ given that $s_t = i$ as follows:

$$b_t = b_n \cdot M^{x_n} \cdot M^{x_{n-1}} \cdot \dots \cdot M^{x_{t+2}} \cdot M^{x_{t+1}} \quad (2.5)$$

Another algorithm which is used in the training stage and employs the forward-backward algorithm as a subroutine, is the Baum-Welch algorithm, to be further discussed in Section 2.5.

2.2 Exploiting Repeated Substrings

Consider a substring $W = w_1, w_2, \dots, w_\ell$ of X , and define

$$M(W) = M^{w_\ell} \odot M^{w_{\ell-1}} \odot \dots \odot M^{w_2} \odot M^{w_1} \quad (2.6)$$

Intuitively, $M_{i,j}(W)$ is the probability of the most likely path starting with state j , making a transition into some other state, emitting w_1 , then making a transition into yet another state and emitting w_2 and so on until making a final transition into state i and emitting w_ℓ .

In the core of our method stands the following observation, which is immediate from the associative nature of matrix multiplication.

Observation 2.3 *We may replace any occurrence of $M^{w_\ell} \odot M^{w_{\ell-1}} \odot \dots \odot M^{w_1}$ in eq. (2.3) with $M(W)$.*

The application of observation 1 to the computation of equation (2.3) saves $\ell - 1$ Viterbi steps *each* time W appears in X , but incurs the additional cost of computing $M(W)$ once.

An intuitive exercise. Let λ denote the number of times a given word W appears, in non-overlapping occurrences, in the input string X . Suppose we naively compute $M(W)$ using $(|W| - 1)$ max-times matrix multiplications, and then apply observation 1 to all occurrences of W before running VA. We gain some speedup in doing so if

$$\begin{aligned} (|W| - 1)k^3 + \lambda k^2 &< \lambda |W| k^2 \\ \lambda &> k \end{aligned} \tag{2.7}$$

Hence, if there are at least k non-overlapping occurrences of W in the input sequence, then it is worthwhile to naively precompute $M(W)$, regardless of its size $|W|$.

Definition 2.4 (Good Substring) *We call a substring W good if we decide to compute $M(W)$.*

We can now give a general four-step framework of our method:

- (I) *Dictionary Selection:* choose the set $D = \{W_i\}$ of good substrings.
- (II) *Encoding:* precompute the matrices $M(W_i)$ for every $W_i \in D$.
- (III) *Parsing:* partition the input sequence X into consecutive good substrings so that $X = W_{i_1} W_{i_2} \dots W_{i_{n''}}$ and let X' denote the compressed representation of this parsing of X , such that $X' = i_1 i_2 \dots i_{n''}$.

(IV) *Propagation*: run VA on X' , using the matrices $M(W_i)$.

The above framework introduces the challenge of how to select the set of good substrings (step I) and how to efficiently compute their matrices (step II). In the next section we show how the RLE, LZ78, SLP and BPE compression schemes can be applied to address this challenge, and how the above framework can be utilized to exploit repetitions of all sufficiently small substrings (this is similar to the Four Russians method). In practice, the choice of the appropriate compression scheme should be made according to the nature of the observed sequences. For example, genomic sequences tend to compress well with BPE [92] and binary images in facsimile or in optical character recognition are well compressed by RLE [1, 8, 9, 19, 69, 75]. LZ78 guarantees asymptotic compression for any sequence and is useful for applications in computational biology, where k is smaller than $\log n$ [17, 27, 33].

Another challenge is how to parse the sequence X (step III) in order to maximize acceleration. We show that, surprisingly, this optimal parsing may differ from the initial parsing induced by the selected compression scheme. To our knowledge, this feature was not applied by previous “acceleration by compression” algorithms.

We focus on computing path probabilities rather than the paths themselves. The actual paths can be reconstructed in linear time as described in Section 2.4.

2.3 Five Different Implementations of the General Framework

In this section, we present five different ways of implementing the general framework, based alternatively on the Four Russians method, and the RLE, LZ78, SLP, and BPE compression schemes. We show how each compression scheme can be used for both selecting the set of good substrings (step I) and for efficiently computing their matrices (step II).

2.3.1 Acceleration via the Four Russians Method

The most naïve approach is probably using all possible substrings of sufficiently small length ℓ as good ones. This approach is quite similar to the *Four Russians method* [11], and leads to a $\Theta(\log n)$

asymptotic speedup. The four-step framework described in Section 2.2 is applied as follows.

- (I) *Dictionary Selection*: all possible strings of length ℓ over alphabet $|\Sigma|$ are good substrings.
- (II) *Encoding*: For $i = 2 \dots \ell$, compute $M(W)$ for all strings W with length i by computing $M(W') \odot M(\sigma)$, where $W = W'\sigma$ for some previously computed string W' of length $i - 1$ and some letter $\sigma \in \Sigma$.
- (III) *Parsing*: X' is constructed by partitioning the input X into blocks of length ℓ .
- (IV) *Propagation*: run VA on X' , using the matrices $M(W_i)$ as described in Section 2.2.

Time and Space Complexity. The encoding step takes $O(2^{|\Sigma|} \ell k^3)$ time as we compute $O(2^{|\Sigma|} \ell)$ matrices and each matrix is computed in $O(k^3)$ time by a single max-times multiplication. The propagation step takes $O(\frac{nk^2}{\ell})$ time resulting in an overall running time of $O(2^{|\Sigma|} \ell k^3 + \frac{nk^2}{\ell})$. Choosing $\ell = \frac{1}{2} \log_{|\Sigma|}(n)$, the running time is $O(2\sqrt{n}k^3 + \frac{2nk^2}{\log_{|\Sigma|}(n)})$. This yields a speedup of $\Theta(\log n)$ compared to VA, assuming that $k < \frac{\sqrt{n}}{\log_{|\Sigma|}(n)}$. In fact, the optimal length is approximately $\ell = \frac{\log n - \log \log n - \log k - 1}{\log |\Sigma|}$, since then the preprocessing and the propagation times are roughly equal. This yields a $\Theta(\ell) = \Theta(\log n)$ speedup, provided that $\ell > 2$, or equivalently that $k < \frac{n}{2^{|\Sigma|^2} \log n}$. Note that for large k the speedup can be further improved using fast matrix multiplication [22, 28, 94].

2.3.2 Acceleration via Run-length Encoding

In this section we obtain an $\Omega(\frac{r}{\log r})$ speedup for decoding an observed sequence with run-length compression ratio r . A string S is *run-length encoded* if it is described as an ordered sequence of pairs (σ, i) , often denoted σ^i . Each pair corresponds to a *run* in S , consisting of i consecutive occurrences of the character σ . For example, the string $aaabbccccc$ is encoded as $a^3b^2c^6$. Run-length encoding serves as a popular image compression technique, since many classes of images (e.g., binary images in facsimile transmission or for use in optical character recognition) typically contain large patches of identically-valued pixels. The four-step framework described in Section 2.2 is applied as follows.

- (I) *Dictionary Selection*: for every $\sigma \in \Sigma$ and every $i = 1, 2, \dots, \log n$ we choose σ^{2^i} as a good substring.
- (II) *Encoding*: since $M(\sigma^{2^i}) = M(\sigma^{2^{i-1}}) \odot M(\sigma^{2^{i-1}})$, we can compute the matrices using repeated squaring.
- (III) *Parsing*: Let $W_1 W_2 \cdots W_{n'}$ be the RLE of X , where each W_i is a run of some $\sigma \in \Sigma$. X' is obtained by further parsing each W_i into at most $\log |W_i|$ good substrings of the form σ^{2^j} .
- (IV) *Propagation*: run VA on X' , as described in Section 2.2.

Time and Space Complexity. The offline preprocessing stage consists of steps I and II. The time complexity of step II is $O(|\Sigma|k^3 \log n)$ by applying max-times repeated squaring in $O(k^3)$ time per multiplication. The space complexity is $O(|\Sigma|k^2 \log n)$. This work is done offline once, during the training stage, in advance for all sequences to come. Furthermore, for typical applications, the $O(|\Sigma|k^3 \log n)$ term is much smaller than the $O(nk^2)$ term of VA.

Steps III and IV both apply one operation per occurrence of a good substring in X' : step III computes, in constant time, the index of the next parsing-comma, and step IV applies a single Viterbi step in k^2 time. Since $|X'| = \sum_{i=1}^{n'} \log |W_i|$, the complexity is

$$\sum_{i=1}^{n'} k^2 \log |W_i| = k^2 \log(|W_1| \cdot |W_2| \cdots |W_{n'}|) \leq k^2 \log((n/n')^{n'}) = O(n' k^2 \log \frac{n}{n'}).$$

Thus, the speedup compared to the $O(nk^2)$ time of VA is $\Omega(\frac{\frac{n}{n'}}{\log \frac{n}{n'}}) = \Omega(\frac{r}{\log r})$.

2.3.3 Acceleration via LZ78 Parsing

In this section we obtain an $\Omega(\frac{\log n}{k})$ speedup for decoding, and a constant speedup in the case where $k > \log n$. We show how to use the LZ78 [103] parsing to find good substrings and how to use the incremental nature of the LZ78 parse to compute $M(W)$ for a good substring W in $O(k^3)$ time.

LZ78 parses the string X into substrings (LZ78-words) in a single pass over X . Each LZ78-word is composed of the longest LZ78-word previously seen plus a single letter. More formally, LZ78 begins with an empty dictionary and parses according to the following rule: when parsing location i , look for the longest LZ78-word W starting at position i which already appears in the dictionary. Read one more letter σ and insert $W\sigma$ into the dictionary. Continue parsing from position $i + |W| + 1$. For example, the string “AACGACG” is parsed into four words: A, AC, G, ACG. Asymptotically, LZ78 parses a string of length n into $O(hn/\log n)$ words [103], where $0 \leq h \leq 1$ is the entropy of the string. The LZ78 parse is performed in linear time by maintaining the dictionary in a trie. Each node in the trie corresponds to an LZ78-word. The four-step framework described in Section 2.2 is applied as follows.

- (I) *Dictionary Selection*: the good substrings are all the LZ78-words in the LZ78-parse of X .
- (II) *Encoding*: construct the matrices incrementally, according to their order in the LZ78-trie, $M(W\sigma) = M(W) \odot M^\sigma$.
- (III) *Parsing*: X' is the LZ78-parsing of X .
- (IV) *Propagation*: run VA on X' , as described in Section 2.2.

Time and Space Complexity. Steps I and III were already conducted offline during the pre-processing compression of the input sequences (in any case LZ78 parsing is linear). In step II, computing $M(W\sigma) = M(W) \odot M^\sigma$, takes $O(k^3)$ time since $M(W)$ was already computed for the good substring W . Since there are $O(n/\log n)$ LZ78-words, calculating the matrices $M(W)$ for all W s takes $O(k^3n/\log n)$. Running VA on X' (step IV) takes just $O(k^2n/\log n)$ time. Therefore, the overall runtime is dominated by $O(k^3n/\log n)$. The space complexity is $O(k^2n/\log n)$.

The above algorithm is useful in many applications, where $k < \log n$. However, in those applications where $k > \log n$ such an algorithm may actually slow down VA. We next show an adaptive variant that is guaranteed to speed up VA, regardless of the values of n and k . This graceful degradation retains the asymptotic $\Omega(\frac{\log n}{k})$ acceleration when $k < \log n$.

An improved algorithm with LZ78 Parsing

Recall that given $M(W)$ for a good substring W , it takes k^3 time to calculate $M(W\sigma)$. This calculation saves k^2 operations each time $W\sigma$ occurs in X in comparison to the situation where only $M(W)$ is computed. Therefore, in step I we should include in D , as good substrings, only words that appear as a prefix of at least k LZ78-words. Finding these words can be done in a single traversal of the trie. The following observation is immediate from the prefix monotonicity of occurrence tries.

Observation 2.5 *Words that appear as a prefix of at least k LZ78-words are represented by trie nodes whose subtrees contain at least k nodes.*

In the previous case it was straightforward to transform X into X' , since each phrase p in the parsed sequence corresponded to a good substring. Now, however, X does not divide into just good substrings and it is unclear what is the optimal way to construct X' (in step III). Our approach for constructing X' is to first parse X into all LZ78-words and then apply the following greedy parsing to each LZ78-word W : using the trie, find the longest good substring $w' \in D$ that is a prefix of W , place a parsing comma immediately after w' and repeat the process for the remainder of W .

Time and Space Complexity. The improved algorithm utilizes substrings that guarantee acceleration (with respect to VA) so it is therefore faster than VA even when $k = \Omega(\log n)$. In addition, in spite of the fact that this algorithm re-parses the original LZ78 partition, the algorithm still guarantees an $\Omega(\frac{\log n}{k})$ speedup over VA as shown by the following lemma.

Lemma 2.6 *The running time of the above algorithm is bounded by $O(k^3 n / \log n)$.*

Proof. The running time of step II is at most $O(k^3 n / \log n)$. This is because the size of the entire LZ78-trie is $O(n / \log n)$ and we construct the matrices, in $O(k^3)$ time each, for just a subset of the trie nodes. The running time of step IV depends on the number of new phrases (commas) that result from the re-parsing of each LZ78-word W . We next prove that this number is at most k for each word.

Consider the first iteration of the greedy procedure on some LZ78-word W . Let w' be the longest prefix of W that is represented by a trie node with at least k descendants. Assume, contrary to fact, that $|W| - |w'| > k$. This means that w'' , the child of w' , satisfies $|W| - |w''| \geq k$, in contradiction to the definition of w' . We have established that $|W| - |w'| \leq k$ and therefore the number of re-parsed words is bounded by $k + 1$. The propagation step IV thus takes $O(k^3)$ time for each one of the $O(n/\log n)$ LZ78-words. So the total time complexity remains $O(k^3 n/\log n)$.

■

Based on Lemma 2.6, and assuming that steps I and III are pre-computed offline, the running time of the above algorithm is $O(nk^2/e)$ where $e = \Omega(\max(1, \frac{\log n}{k}))$. The space complexity is $O(k^2 n/\log n)$.

2.3.4 Acceleration via Straight-Line Programs

In this subsection we show that if an input sequence has a grammar representation with compression ratio r , then HMM decoding can be accelerated by a factor of $\Omega(\frac{r}{k})$.

Let us shortly recall the grammar-based approach to compression. A *straight-line program* (SLP) is a context-free grammar generating exactly one string. Moreover, only two types of productions are allowed: $X_i \rightarrow a$ and $X_i \rightarrow X_p X_q$ with $i > p, q$. The string represented by a given SLP is a unique text corresponding to the last nonterminal X_z . We say that the size of an SLP is equal to its number of productions. An example SLP is illustrated in Figure 2-3.

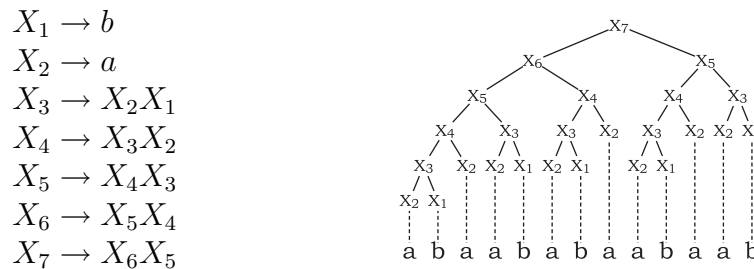


Figure 2-3: Consider the string *abaababaabaab*. It could be generated by the above SLP, also known as the *Fibonacci SLP*.

Rytter [85] proved that the resulting encoding of most popular compression schemes can be

transformed to straight-line programs quickly and without large expansion. In particular, consider an LZ77 encoding [104] with n'' blocks for a text of length n . Rytter's algorithm produces an SLP-representation with size $n' = O(n'' \log n)$ of the same text, in $O(n')$ time. Moreover, n' lies within a $\log n$ factor from the size of a *minimal* SLP describing the same text. Note also that any text compressed by the LZ78 encoding can be transformed directly into a straight-line program within a constant factor. However, here we focus our SLP example on LZ77 encoding since in certain cases LZ77 is exponentially shorter than LZ78, so even with the $\log n$ degradation associated with transforming LZ77 into an SLP, we may still get an exponential speedup over LZ78 from Section 2.3.3.

We next describe how to use SLP to achieve the speedup.

- (I) *Dictionary Selection*: let \mathcal{X} be an SLP representation of the input sequence. We choose all strings corresponding to nonterminals $X_1, \dots, X_{n'}$ as good substrings.
- (II) *Encoding*: compute $M(X_i)$ in the same order as in \mathcal{X} . Every concatenating rule requires just one max-times multiplication.
- (III) *Parsing*: Trivial (the input is represented by the single matrix representing $X_{n'}$).
- (IV) *Propagation*: $v_t = M(X_{n'}) \odot v_0$.

Time and Space Complexity. Let n' be the number of rules in the SLP constructed in the parsing step ($r = n/n'$ is the ratio of the *grammar-based compression*). The parsing step has an $O(n)$ time complexity and is computed offline. The number of max-times multiplications in the encoding step is n' . Therefore, the overall complexity of decoding the HMM is $n'k^3$, leading to a speedup factor of $\Omega(\frac{r}{k})$.

In the next section we give another example of a grammar-based compression scheme where the size of the uncompressed text may grow exponentially with respect to its description, and that furthermore allows, in practice, to shift the Encoding Step (III) to an off-line preprocessing stage, thus yielding a speedup factor of $\Omega(r)$.

2.3.5 Acceleration via Byte-Pair Encoding

In this section byte pair encoding is utilized to accelerate the Viterbi decoding computations by a factor of $\Omega(r)$, where n' is the number of characters in the BPE-compressed sequence X and $r = n/n'$ is the BPE compression ratio of the sequence. The corresponding pre-processing term for encoding is $O(|\Sigma'|k^3)$, where Σ' denotes the set of character codes in the extended alphabet.

Byte pair encoding [39, 90, 92] is a simple form of data compression in which the most common pair of consecutive bytes of data is replaced with a byte that does not occur within that data. This operation is repeated until either all new characters are used up or no pair of consecutive two characters appears frequently in the substituted text. For example, the input string $ABABCABCD$ could be BPE encoded to $XY YD$, by applying the following two substitution operations: First $AB \rightarrow X$, yielding $XXCXCD$, and then $XC \rightarrow Y$. A substitution table, which stores for each character code the replacement it represents, is required to rebuild the original data. The compression ratio of BPE has been shown to be about 30% for biological sequences [92].

The compression time of BPE is $O(|\Sigma'|n)$. Alternatively, one could follow the approach of Shibata et al. [90, 92], and construct the substitution-table offline, during system set-up, based on a sufficient set of representative sequences. Then, using this pre-constructed substitution table, the sequence parsing can be done in time linear in the total length of the original and the substituted text. Let $\sigma \in \Sigma'$ and let W_σ denote the word represented by σ in the BPE substitution table. The four-step framework described in Section 2.2 is applied as follows.

- (I) *Dictionary Selection*: all words appearing in the BPE substitution table are good substrings, *i.e.* $D = \{W_\sigma\}$ for all $\sigma \in \Sigma'$.
- (II) *Encoding*: if σ is a substring obtained via the substitution operation $AB \rightarrow \sigma$ then

$$M(W_\sigma) = M(W_A) \odot M(W_B).$$

So each matrix can be computed by multiplying two previously computed matrices.

- (III) *Parsing*: given an input sequence X , apply BPE parsing as described in [90, 92].

(IV) *Propagation*: run VA on X' , using the matrices $M(W_i)$ as described in Section 2.2.

Time and Space Complexity. Step I was already taken care of during the system-setup stage and therefore does not count in the analysis. Step II is implemented as an offline, preprocessing stage that is independent of the observed sequence X but dependent on the training model. It can therefore be conducted once in advance, for all sequences to come. The time complexity of this off-line stage is $O(|\Sigma'|k^3)$ since each matrix is computed by one max-times matrix multiplication in $O(k^3)$ time. The space complexity is $O(|\Sigma'|k^2)$. Since we assume Step III is conducted in pre-compression, the compressed decoding algorithm consists of just Step IV. In the propagation step (IV), given an input sequence of size n , compressed into its BPE-encoding of size n' (e.g. $n' = 4$ in the above example, where $X = ABABCABCD$ and $X' = XYYD$), we run VA using at most n' matrices. Since each VA step takes k^2 time, the time complexity of this step is $O(n'k^2)$. Thus, the time complexity of the BPE-compressed decoding is $O(n'k^2)$ and the speedup, compared to the $O(nk^2)$ time of VA, is $\Omega(r)$.

2.4 Optimal state-path recovery

In the previous section, we described our decoding algorithms assuming that we only want to compute the probability of the optimal path. In this section we show how to trace back the optimal path, within the same space complexity and in $O(n)$ time. To do the traceback, VA keeps, along with the vector v_t (see eq. (2.2)), a vector of the maximizing arguments of eq. (2.2), namely:

$$u_{t+1}[i] = \operatorname{argmax}_j \{P_{i,j} \cdot v_t[j]\} \quad (2.8)$$

It then traces the states of the most likely path in reverse order. The last state s_n is simply the largest element in v_n , $\operatorname{argmax}_j \{v_n[j]\}$. The rest of the states are obtained from the vectors u by $s_{t-1} = u_t[s_t]$. We use exactly the same mechanism in the propagation step (IV) of our algorithm. The problem is that in our case, this only retrieves the states on the boundaries of good substrings but not the states within each good substring. We solve this problem in a similar manner.

Note that in all of our decoding algorithms every good substring W is such that $W = W_A W_B$ where both W_A and W_B are either good substrings or single letters. In LZ78-accelerated decoding, W_B is a single letter, when using RLE $W_A = W_B = \sigma^{|W|/2}$, SLP consists just of production rules involving a single letter or exactly two non-terminals, and with BPE $W_A, W_B \in \Sigma'$. For this reason, we keep, along with the matrix $M(W)$, a matrix $R(W)$ whose elements are:

$$R(W)_{i,j} = \operatorname{argmax}_k \{M(W_A)_{i,k} \odot M(W_B)_{k,j}\} \quad (2.9)$$

Now, for each occurrence of a good substring $W = w_1, w_2, \dots, w_\ell$ we can reconstruct the most likely sequence of states s_1, s_2, \dots, s_ℓ as follows. From the partial traceback, using the vectors u , we know the two states s_0 and s_ℓ , such that s_0 is the most likely state immediately before w_1 was generated and s_ℓ is the most likely state when w_ℓ was generated. We find the intermediate states by recursive application of the computation $s_{|W_A|} = R(W)_{s_0, s_\ell}$.

Time and Space Complexity. In all compression schemes, the overall time required for tracing back the most likely path is $O(n)$. Storing the matrices R does not increase the basic space complexity, since we already stored the similar-sized matrices $M(W)$.

2.5 The Training Problem

In the training problem we are given as input the number of states in the HMM and an observed training sequence X . The aim is to find a set of model parameters θ (i.e., the emission and transition probabilities) that maximize the likelihood to observe the given sequence $P(X|\theta)$. The most commonly used training algorithms for HMMs are based on the concept of Expectation Maximization. This is an iterative process in which each iteration is composed of two steps. The first step solves the decoding problem given the current model parameters. The second step uses the results of the decoding process to update the model parameters. These iterative processes are guaranteed to converge to a local maximum. It is important to note that since the dictionary selection step (I) and the parsing step (III) of our algorithm are independent of the model parameters, we only

need run them once, and repeat just the encoding step (II) and the propagation step (IV) when the decoding process is performed in each iteration.

2.5.1 Viterbi training

The first step of Viterbi training [33] uses VA to find the most likely sequence of states given the current set of parameters (i.e., decoding). Let A_{ij} denote the number of times the state i follows the state j in the most likely sequence of states. Similarly, let $E_i(\sigma)$ denote the number of times the letter σ is emitted by the state i in the most likely sequence. The updated parameters are given by:

$$P_{ij} = \frac{A_{ij}}{\sum_{i'} A_{i'j}} \text{ and } e_i(\sigma) = \frac{E_i(\sigma)}{\sum_{\sigma'} E_i(\sigma')} \quad (2.10)$$

Note that the Viterbi training algorithm does not converge to the set of parameters that maximizes the likelihood to observe the given sequence $P(X|\theta)$, but rather the set of parameters that locally maximizes the contribution to the likelihood from the most probable sequence of states [33]. It is easy to see that the time complexity of each Viterbi training iteration is $O(k^2n + n) = O(k^2n)$ so it is dominated by the running time of VA. Therefore, we can immediately apply our compressed decoding algorithms from Section 2.3 to obtain a better running time per iteration.

2.5.2 Baum-Welch training

The Baum-Welch training algorithm converges to a set of parameters that maximizes the likelihood to observe the given sequence $P(X|\theta)$, and is the most commonly used method for model training. Recall the forward-backward matrices: $f_t[i]$ is the probability to observe the sequence x_1, x_2, \dots, x_t requiring that the t 'th state is i and that $b_t[i]$ is the probability to observe the sequence $x_{t+1}, x_{t+2}, \dots, x_n$ given that the t 'th state is i . The first step of Baum-Welch calculates $f_t[i]$ and $b_t[i]$ for every $1 \leq t \leq n$ and every $1 \leq i \leq k$. This is achieved by applying the forward and backward algorithms to the input data in $O(nk^2)$ time (see eqs. (2.4) and (2.5)). The second

step recalculates A and E according to

$$\begin{aligned} A_{i,j} &= \sum_t P(s_t = j, s_{t+1} = i | X, \theta) \\ E_i(\sigma) &= \sum_{t|x_t=\sigma} P(s_t = i | X, \theta) \end{aligned} \quad (2.11)$$

where $P(s_t = j, s_{t+1} = i | X, \theta)$ is the probability that a transition from state j to state i occurred in position t in the sequence X , and $P(s_t = i | X, \theta)$ is the probability for the t 'th state to be i in the sequence X . These probabilities are calculated as follows using the matrices $f_t[i]$ and $b_t[i]$ that were computed in the first step.

$P(s_t = j, s_{t+1} = i | X, \theta)$ is given by the product of the probabilities to be in state j after emitting x_1, x_2, \dots, x_t , to make a transition from state j to state i , to emit x_{t+1} at state i and to emit the rest of X given that the state is i :

$$P(s_t = j, s_{t+1} = i | X, \theta) = \frac{f_t[j] \cdot P_{i,j} \cdot e_i(x_{t+1}) \cdot b_{t+1}[i]}{P(X | \theta)} \quad (2.12)$$

where the division by $P(X | \theta) = \sum_i f_n[i]$ is a normalization by the probability to observe X given the current model parameters. Similarly

$$P(s_t = i | X, \theta) = \frac{f_t[i] \cdot b_t[i]}{P(X | \theta)}. \quad (2.13)$$

Finally, after the matrices A and E are recalculated, Baum-Welch updates the model parameters according to (2.10).

We next describe how to accelerate the Baum-Welch algorithm. It is important to notice that, in the first step of Baum-Welch, our algorithms to accelerate VA (sections 2.3.2 and 2.3.3) can be used to accelerate the forward-backward algorithms by simply replacing the max-times matrix multiplication with regular matrix multiplication. However, the accelerated forward-backward algorithms will only calculate f_t and b_t on the boundaries of good substrings. In what follows, we explain how to solve this problem and speed up the second step of Baum-Welch as well. We focus on updating the matrix A , updating the matrix E can be done in a similar fashion.

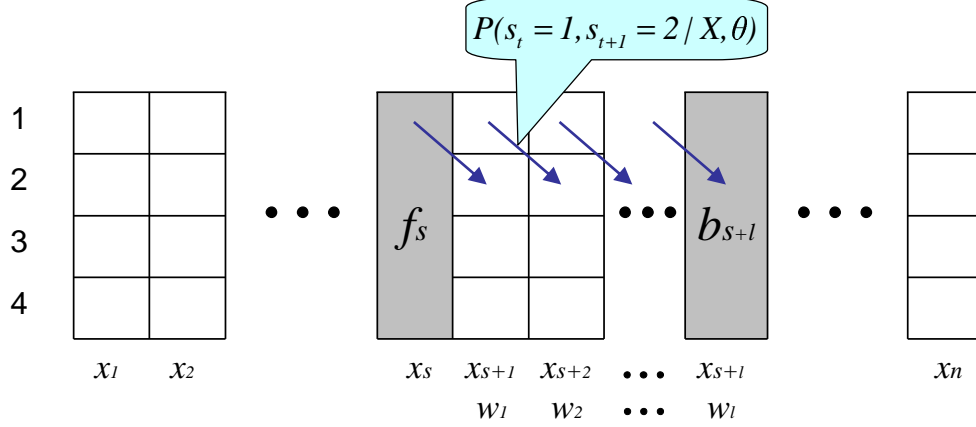


Figure 2-4: The contribution of an occurrence of the good string W to A_{12} , computed as the sum of the arrow probabilities.

We observe that when accumulating the contribution of some appearance of a good substring W to A , Baum-Welch performs $O(k^2|W|)$ operations, but updates at most k^2 entries (the size of A). Therefore, we may gain a speedup by precalculating the contribution of each good substring to A and E . More formally, let $W = w_1w_2 \cdots w_\ell$ be a substring of the observed sequence X starting s characters from the beginning (i.e., $W = x_{s+1}x_{s+2} \cdots x_{s+\ell}$ as illustrated in Fig. 2-4). According to (2.11) and (2.12), the contribution of this occurrence of W to A_{ij} is:

$$\begin{aligned} & \sum_{t=s}^{s+\ell-1} \frac{f_t[j] \cdot P_{i,j} \cdot e_i(x_{t+1}) \cdot b_{t+1}[i]}{P(X|\theta)} \\ &= \frac{1}{P(X|\theta)} \sum_{t=0}^{\ell-1} f_{t+s}[j] \cdot P_{i,j} \cdot e_i(w_{t+1}) \cdot b_{t+s+1}[i] \end{aligned}$$

However, by equation (2.4) we have

$$\begin{aligned} f_{t+s} &= M^{x_{t+s}} \cdot M^{x_{t+s-1}} \cdots M^{x_1} \cdot f_0 \\ &= M^{x_{t+s}} \cdot M^{x_{t+s-1}} \cdots M^{x_{s+1}} \cdot f_s \\ &= M^{w_t} \cdot M^{w_{t-1}} \cdots M^{w_1} \cdot f_s \end{aligned}$$

and similarly by equation (2.5) $b_{t+s+1} = b_{s+\ell} \cdot M^{w_\ell} \cdot M^{w_{\ell-1}} \cdots M^{w_{t+2}}$. The above sum thus equals

$$\begin{aligned}
& \frac{1}{P(X|\theta)} \sum_{t=0}^{\ell-1} (M^{w_t} M^{w_{t-1}} \dots M^{w_1} \cdot f_s)_j \cdot P_{ij} \cdot e_i(w_{t+1}) \cdot (b_{s+\ell} \cdot M^{w_\ell} M^{w_{\ell-1}} \dots M^{w_{t+2}})_i \\
&= \frac{1}{P(X|\theta)} \sum_{t=0}^{\ell-1} \sum_{\alpha=1}^k (M^{w_t} \dots M^{w_1})_{j,\alpha} \cdot f_s[\alpha] \cdot P_{ij} \cdot e_i(w_{t+1}) \cdot \sum_{\beta=1}^k b_{s+\ell}[\beta] \cdot (M^{w_\ell} \dots M^{w_{t+2}})_{\beta,i} \\
&= \sum_{\alpha=1}^k \sum_{\beta=1}^k f_s[\alpha] \cdot b_{s+\ell}[\beta] \cdot \underbrace{\frac{1}{P(X|\theta)} \sum_{t=0}^{\ell-1} (M^{w_t} \dots M^{w_1})_{j,\alpha} \cdot P_{ij} \cdot e_i(w_{t+1}) \cdot (M^{w_\ell} \dots M^{w_{t+2}})_{\beta,i}}_{R_{ij}^{\alpha\beta}} \\
&\equiv \sum_{\alpha=1}^k \sum_{\beta=1}^k f_s[\alpha] \cdot b_{s+\ell}[\beta] \cdot R_{ij}^{\alpha\beta} \tag{2.14}
\end{aligned}$$

Notice that the four dimensional array $R_{ij}^{\alpha\beta}$ can be computed in an encoding step (II) in $O(\ell k^4)$ time and is not dependant on the string context prior to X_s or following $X_{s+\ell}$. Furthermore, the vectors f_s and $b_{s+\ell}$ where already computed in the first step of Baum-Welch since they refer to boundaries of a good substring. Therefore, R can be used according to (2.14) to update A_{ij} for a single occurrence of W and for some specific i and j in $O(k^2)$ time. So R can be used to update A_{ij} for a single occurrence of W and for every i, j in $O(k^4)$ time. To get a speedup we need λ , the number of times the good substring W appears in X to satisfy:

$$\begin{aligned}
\ell k^4 + \lambda k^4 &< \lambda \ell k^2 \\
\lambda &> \frac{\ell k^2}{\ell - k^2} \tag{2.15}
\end{aligned}$$

This is reasonable if k is small. If $\ell = 2k^2$, for example, then we need λ to be greater than $2k^2$.

2.6 Parallelization

We conclude this chapter with a discussion on a parallel implementation of our algorithms. We first note that the classical formulation of the decoding algorithms by the basic recursion in Eq. (2.2) imposes a constraint on a parallel implementation. It is possible to compute the maximum in

(2.2) in parallel, and eliminate the linear dependency on k , the number of states. However, the dependency of v_{t+1} on v_t makes it difficult to avoid the dependency of the running time on n , the length of the input.

Once VA is cast into the form of Eq. (2.3), it is easy to achieve full parallelization, both with respect to the number of states in the model and with respect to the length of the input. Similar ideas were previously considered in [64, 37] in the context of VLSI architectures for Viterbi decoders in data communications. We describe and analyze the general idea in the CREW (Concurrent Read, Exclusive Write) PRAM model for the Four-Russians variant of our method. The same approach applies to the other algorithms described above with the exception of the SLP-based algorithm, assuming that the parsing step (III) is performed in advance.

Our algorithms, as well as VA in the form of (2.3), are essentially a sequence of matrix multiplications (either max-times or regular matrix multiplication) which may be evaluated in any order. The product of two k -by- k matrices can be computed in parallel in $O(\log k)$ time using $O(k^3)$ processors. The product of x such matrices can therefore be calculated in parallel in $O(\log x \log k)$ time using $O(xk^3)$ processors. Therefore, VA in the form of (2.3) can be performed in parallel. The maximal number of processors used concurrently is $O(nk^3)$, and the running time is $O(\log n \log k)$. For our Four-Russians algorithm, we first compute in parallel all possible matrices for words of length $\frac{1}{2} \log_{|\Sigma|}(n)$. This corresponds to step (II) of the algorithm. Next, we perform step (IV) by computing, in parallel, the product of $\frac{2n}{\log_{|\Sigma|}(n)}$ matrices. The maximal number of processors used concurrently along this computation is $O(\frac{nk^3}{\log n})$, and the running time is $O(\log \frac{n}{\log n} \log k) = O(\log n \log k)$. As can be seen, this does not improve the asymptotic running time, but does decrease the required number of processors. It should be noted that if the number of processors is bounded, then exploiting repetitions does improve the asymptotic running time. Tracing the optimal path can also be done in parallel using $O(n)$ processors in $O(\log n)$ time in both cases.

Chapter 3

Totally Monotone Matrices and Monge Matrices

Probably the best known DP speedup technique is the seminal algorithm of Aggarwal, Klawe, Moran, Shor, and Wilber [3] (nicknamed SMAWK in the literature) for finding the row minima of a totally monotone $n \times m$ matrix in $O(n + m)$ time. A matrix $M = (M_{ij})$ is *totally monotone* if for every i, i', j, j' such that $i < i', j < j'$, and $M_{ij} \leq M_{ij'}$, we have that $M_{i'j} \leq M_{i'j'}$. Aggarwal et al. showed that a wide variety of DPs in computational geometry can be reduced to the problem of finding row minima in totally monotone or in Monge matrices.

A matrix $M = (M_{ij})$ is *Monge* if for every i, i', j, j' such that $i < i', j < j'$, we have $M_{ij} + M_{i'j'} \geq M_{i'j} + M_{ij'}$. It is immediate that if M is Monge then it is totally monotone. Monge matrices were discovered back in 1781 by the French engineer and mathematician Gaspard Monge [77] in the context of transportation problems. In 1961, Hoffman [46] rediscovered these matrices and coined the term *Monge property*. Since SMAWK was introduced, many papers on applications which lead to Monge matrices have been published (see [20] for a survey).

In this chapter, we present speedups for DPs used for solving planar graphs problems such as single-source shortest paths, replacement paths, bipartite perfect matching, feasible flow, and feasible circulation. The speedups are based on the fact that shortest paths in planar graphs exhibit a property similar to Monge. Namely, they can be described by an upper-triangular fragment of a

Monge matrix.

Shortest paths with negative lengths. The problem of *directed shortest paths with negative lengths* is as follows: Given a directed graph G with positive and negative arc-lengths containing no negative cycles,¹ and given a source node s , find the distances from s to all the nodes in the graph.

For general graphs, the Bellman-Ford algorithm solves the problem in $O(mn)$ time, where m is the number of arcs and n is the number of nodes. For integer lengths whose absolute values are bounded by N , the algorithm of Gabow and Tarjan [38] takes $O(\sqrt{nm} \log(nN))$. For integer lengths exceeding $-N$, the algorithm of Goldberg [40] takes $O(\sqrt{nm} \log N)$ time.

For planar graphs, there has been a series of results yielding progressively better bounds. The first algorithm that exploited planarity was due to Lipton, Rose, and Tarjan [68], who gave an $O(n^{3/2})$ algorithm. Henzinger et al. [44] gave an $O(n^{4/3} \log^{2/3} D)$ algorithm where D is the sum of the absolute values of the lengths. Fakcharoenphol and Rao [35] gave an algorithm requiring $O(n \log^3 n)$ time and $O(n \log n)$ space.

These planarity-exploiting shortest paths algorithms, make use of planar separators [67, 73]. Given an n -node planar embedded directed graph G with arc-lengths, a planar separator is a Jordan curve C that passes through $O(\sqrt{n})$ nodes (and no arcs) such that between $n/3$ and $2n/3$ nodes are enclosed by C . A node through which C passes is called a *boundary node*. Cutting the planar embedding along C and duplicating boundary nodes yields two subgraphs G_0 and G_1 . Refer to Fig. 3-2 for an illustration.

All planarity-exploiting shortest paths algorithms use DP for a procedure that can be seen as a Bellman-Ford step. In this step, we assume that we have already computed the shortest paths distances in G_0 and in G_1 between every pair of boundary vertices, and stored them in two $\sqrt{n} \times \sqrt{n}$ matrices δ_0 and δ_1 . The DP is then used to compute the shortest paths distances in G from an arbitrary boundary vertex r to all boundary vertices using δ_0 and δ_1 . Lipton, Rose and Tarjan [68] showed how to do this in $O(n^{3/2})$ time, Fakcharoenphol and Rao [35] improved to $O(n \log^2 n)$, and Mozes [79] to $O(n \log n)$. We show (in Section 3.2) how this DP can be preformed in $O(n\alpha(n))$

¹Algorithms for this problem can also be used to detect negative cycles.

time, where $\alpha(n)$ is the inverse Ackerman function. Our speedup is based on the fact that the upper triangular fragment of δ_0 and δ_1 is Monge. We have recently used this speedup in [52] to obtain the following theorem. In this thesis, we only describe the DP part of this result (i.e. the Bellman-Ford step).

Theorem 3.1 *There is an $O(n \log^2 n)$ -time, linear-space algorithm to find single-source shortest paths in planar directed graphs with negative lengths.*

In addition to being a fundamental problem in combinatorial optimization, shortest paths in planar graphs with negative lengths arises in solving other problems. Miller and Naor [74] show that, by using planar duality, the following problem can be reduced to shortest paths in a planar directed graph:

Feasible circulation: Given a directed planar graph with upper and lower arc-capacities, find an assignment of flow to the arcs so that each arc's flow is between the arc's lower and upper capacities, and, for each node, the flow into the node equals the flow out.

They further show that the following problem can in turn be reduced to feasible circulation:

Feasible flow: Given a directed planar graph with arc-capacities and node-demands, find an assignment of flow that respects the arc-capacities and such that, for each node, the flow into the node minus the flow out equals the node's demand.

For integer-valued capacities and demands, the solutions obtained to the above problems are integer-valued. Consequently, as Miller and Naor point out, the problem of finding a *perfect matching* in a bipartite planar graph can be solved using an algorithm for feasible flow. Our speedup thus accelerates algorithms for bipartite planar perfect matching, feasible flow, and feasible circulation.

Replacement paths with nonnegative lengths. The *replacement-paths problem* is defined as follows: we are given a directed graph with non-negative arc lengths and two nodes s and t . We are required to compute, for every arc e in the shortest path between s and t , the length of an s -to- t shortest path that avoids e .

Emek et al. [34] gave an $O(n \log^3 n)$ -time algorithm for solving the replacement-paths problem in a directed planar graph. This was achieved by reducing the problem to that of computing the following n values associated with the diagonal of a certain $n \times n$ matrix $\widehat{\text{len}}$. The value of the diagonal cell (i, i) is the minimal element of the rectangular portion of $\widehat{\text{len}}$ defined by rows 1 to i and columns i to n . The matrix $\widehat{\text{len}}$, defined in Section 3.3, is designed to capture all possible replacement paths. By exploiting a Monge property of $\widehat{\text{len}}$, we obtain the following logarithmic improvement for finding these n values.

Theorem 3.2 *There is an $O(n \log^2 n)$ -time algorithm for solving the replacement-paths problem in a directed planar graph.*

In the following section we describe some preliminary definitions. Then, in Section 3.2, we describe our speedup for the Bellman-Ford procedure and in Section 3.3 we describe the replacement paths algorithm.

3.1 Preliminaries

We now describe some necessary definitions regarding Monge matrices and planar graphs.

3.1.1 Monotonicity, Monge and Matrix Searching.

A matrix $M = (M_{ij})$ is *totally monotone* if for every i, i', j, j' such that $i < i', j < j'$ and $M_{ij} \leq M_{i'j'}$, we also have $M_{i'j} \leq M_{ij'}$. The SMAWK algorithm of Aggarwal et al. [3] finds all row-maxima of a totally monotone $n \times m$ matrix M in just $O(n + m)$ time. It is easy to see that by negating each element of M and reversing the order of its columns, SMAWK can be used to find the row minima of M as well.

We will not give a full description of SMAWK, but let us outline an intuitive explanation of the basis of SMAWK. Consider an $n \times n$ totally monotone matrix M . We want to find the maximum element in every row of M using only $O(n)$ comparisons. In the beginning, every element of M is a possible candidate for being its row maximum. Assume for example, that we

then compare $M_{\frac{n}{2}, \frac{n}{2}}$ with its adjacent cell $M_{\frac{n}{2}, \frac{n}{2}+1}$. Notice that if $M_{\frac{n}{2}, \frac{n}{2}} \leq M_{\frac{n}{2}, \frac{n}{2}+1}$ then by total monotonicity, $M_{i, \frac{n}{2}} \leq M_{i, \frac{n}{2}+1}$ for every $i \geq \frac{n}{2}$. In other words, by performing one comparison, we have eliminated $\frac{n}{2}$ candidates from being the maximum of their row. If on the other hand $M_{\frac{n}{2}, \frac{n}{2}} \geq M_{\frac{n}{2}, \frac{n}{2}+1}$ then by total monotonicity, $M_{i, \frac{n}{2}} \geq M_{i, \frac{n}{2}+1}$ for every $i \leq \frac{n}{2}$, and again our one comparison eliminated $\frac{n}{2}$ candidates.

Another matrix property related to total-monotonicity is the *Monge* property. A matrix M is *convex Monge* (resp. *concave Monge*) if for every i, i', j, j' such that $i < i', j < j'$, we have $M_{ij} + M_{i'j'} \geq M_{ij'} + M_{i'j}$ (resp. $M_{ij} + M_{i'j'} \leq M_{ij'} + M_{i'j}$). It is immediate that if M is convex Monge then it is totally monotone. It is also easy to see that the matrix obtained by transposing M is also totally monotone. Thus SMAWK can also be used to find the *column* minima and maxima of a convex Monge matrix.

In [51] Klawe and Kleitman define a *falling staircase matrix* to be a lower triangular fragment of a totally monotone matrix. More precisely, $(M, \{f(i)\}_{0 \leq i \leq n+1})$ is an $n \times m$ falling staircase matrix if

1. for $i = 0, \dots, n+1$, $f(i)$ is an integer with $0 = f(0) < f(1) \leq f(2) \leq \dots \leq f(n) < f(n+1) = m+1$.
2. M_{ij} , is a real number if and only if $1 \leq i \leq n$ and $1 \leq j \leq f(i)$. Otherwise, M_{ij} is blank.
3. (total monotonicity) for $i < k$ and $j < l \leq f(i)$, and $M_{ij} \leq M_{il}$, we have $M_{kj} \leq M_{kl}$.

Finding the row maxima in a falling staircase matrix can be easily done using SMAWK in $O(n+m)$ time after replacing the blanks with sufficiently small numbers so that the resulting matrix is totally monotone. However, this trick does not work for finding the row minima. Aggarwal and Klawe [2] give an $O(m \log \log n)$ time algorithm for finding row-minima in falling staircase matrices of size $n \times m$. Klawe and Kleitman give in [51] a more complicated algorithm that computes the row-minima of an $n \times m$ falling staircase matrix in $O(m\alpha(n)+n)$ time, where $\alpha(n)$ is the inverse Ackerman function. If M satisfies the above conditions with total monotonicity replaced by the convex Monge property then M and the matrix obtained by transposing M and reversing

the order of rows and of columns are falling staircase. In this case both algorithms can be used to find the column-minima as well as the row-minima.

3.1.2 Jordan Separators for Embedded Planar Graphs.

A *planar embedding* of a graph assigns each node to a distinct point on the plane, and assigns each edge to a simple arc between the points corresponding to its endpoints, with the property that no arc-arc or arc-point intersections occur except for those corresponding to edge-node incidence in the graph. A graph is planar if it has a planar embedding. Consider the set of points on the sphere that are not assigned to any node or edge; each connected component of this set is a *face* of the embedding.

Miller [73] gave a linear-time algorithm that, given a triangulated two-connected n -node planar embedded graph, finds a simple cycle separator consisting of at most $2\sqrt{2}\sqrt{n}$ nodes, such that at most $2n/3$ nodes are strictly enclosed by the cycle, and at most $2n/3$ nodes are not enclosed.

For an n -node planar embedded graph G that is not necessarily triangulated or two-connected, we define a *Jordan separator* to be a Jordan curve C that intersects the embedding of the graph only at nodes such that at most $2n/3$ nodes are strictly enclosed by the curve and at most $2n/3$ nodes are not enclosed. The nodes intersected by the curve are called *boundary nodes* and denoted V_c . To find a Jordan separator with at most $2\sqrt{2}\sqrt{n}$ boundary nodes, add artificial edges with sufficiently large lengths to triangulate the graph and make it two-connected without changing the distances in the graph. Now apply Miller's algorithm.

The *internal part of G with respect to C* is the embedded subgraph consisting of the nodes and edges enclosed by C , i.e. including the nodes intersected by C . Similarly, the *external part of G with respect to C* is the subgraph consisting of the nodes and edges not strictly enclosed by C , i.e. again including the nodes intersected by C .

Let $G_1(G_0)$ denote the internal (external) part of G with respect to C . Since C is a Jordan curve, the set of points of the plane strictly exterior to C form a connected region. Furthermore, it contains no point or arc corresponding to a node or edge of G_1 . Therefore, the region remains connected when these points and arcs are removed, so the region is a subset of some face of

G_1 . Since every boundary node is intersected by C , it follows that all boundary nodes lie on the boundary of a single face of G_1 . Similarly, in G_0 , all boundary nodes lie on a single face.

3.2 A Bellman-Ford Variant for Planar Graphs

The classical Bellman-Ford algorithm computes the shortest paths from a given source r to all vertices in a general directed graph with positive and negative lengths. Let $e_j[v]$ be the length of a shortest path from r to v that is composed of at most j edges. Bellman-Ford computes $e_{n-1}[v]$ in $O(mn)$ time for all vertices v , where m is the number of edges and n is the number of vertices. The Bellman-Ford pseudocode is given in Figure 3-1. We denote $\delta[w, v]$ as the length of the edge (w, v) , and ∞ if no such edge exists.

```

1:  $e_0[v] = \infty$  for all  $v \in V$ 
2:  $e_0[r] = 0$ 
3: for  $j = 1, 2, \dots, n - 1$ 
4:    $e_j[v] = \min_{w \in V} \{e_{j-1}[w] + \delta[w, v]\}, \forall v \in V$ 

```

Figure 3-1: Pseudocode for the the Bellman-Ford algorithm that computes the shortest-path distances from a source vertex r to all vertices of a general directed graph $G = (V, E)$ with positive and negative edge lengths $\delta(\cdot)$.

Since the number of edges m in a planar graph G is $O(n)$, the Bellman-Ford algorithm runs in $O(n^2)$ time on planar graphs. Faster planarity-exploiting shortest paths algorithms use a Bellman-Ford variant as a procedure for computing the distances from r to all boundary nodes (i.e., the nodes of V_c). This is done by using the recursively computed δ_0 and δ_1 tables, where $\delta_i[w, v]$ is the w -to- v distance in G_i for every pair w, v of boundary nodes. The Bellman-Ford variant of [68] requires $O(n^{3/2})$ time, the one of [35] runs in $O(n \log^2 n)$, and that of [79] in $O(n \log n)$. In this section, we prove the following theorem that gives an $O(n\alpha(n))$ Bellman-Ford variant.

Theorem 3.3 *Let G be a directed planar graph with arbitrary arc-lengths. Let C be a separator in G and let G_0 and G_1 be the external and internal parts of G with respect to C . Let δ_0 and δ_1 be the all-pairs distances between nodes in V_c in G_0 and in G_1 respectively. Let $r \in V_c$ be an*

arbitrary node on the boundary. There exists an algorithm that, given δ_0 and δ_1 , computes the from- r distances in G to all nodes in V_c in $O(|V_c|^2\alpha(|V_c|))$ time and $O(|V_c|)$ space.

Summary of the Algorithm. A shortest path in G passes back and forth between G_0 and G_1 . Refer to Fig. 3-2 and Fig. 3-3 for an illustration. We use a variant of Bellman-Ford to compute the distances in G from r to all the boundary nodes. Alternating iterations use the all-boundary-distances δ_0 and δ_1 . Because the distances have a Monge property, each iteration can be implemented by two executions of the Klawe-Kleitman algorithm [51] for finding row minima in a lower triangular fragment of a Monge matrix. Each iteration is performed in $O(\sqrt{n}\alpha(n))$. The number of iterations is $O(\sqrt{n})$, so the overall time is $O(n\alpha(n))$.

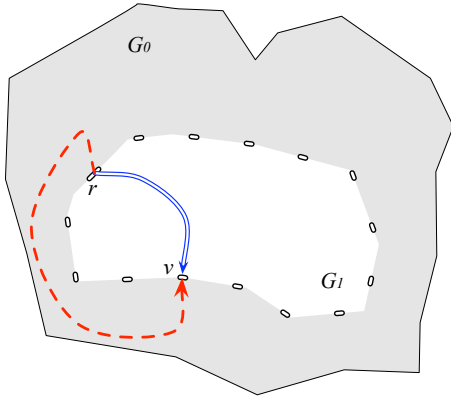


Figure 3-2: A graph G and a decomposition using a Jordan curve into an external subgraph G_0 (in gray) and an internal subgraph G_1 (in white). Only boundary nodes are shown. r and v are boundary nodes. The double-lined blue path is an r -to- v shortest path in G_1 . The dashed red path is an r -to- v shortest path in G_0 .

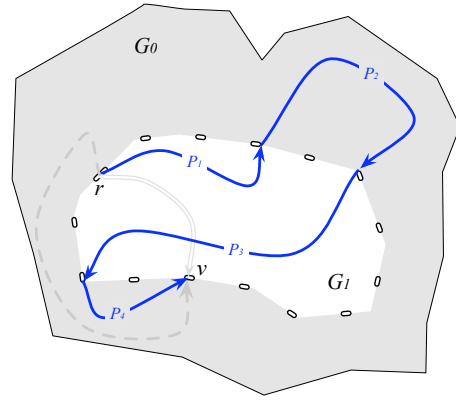


Figure 3-3: The solid blue path is an r -to- v shortest path in G . It can be decomposed into four subpaths. The subpaths P_1 and P_3 (P_2 and P_4) are shortest paths in G_1 (G_0) between boundary nodes. The r -to- v shortest paths in G_0 and G_1 are shown in gray in the background.

The rest of this section describes the algorithm, thus proving Theorem 3.3. The following structural lemma stands in the core of the computation. The same lemma has been implicitly used by previous planarity-exploiting algorithms.

Lemma 3.4 *Let P be a simple r -to- v shortest path in G , where $v \in V_c$. Then P can be decomposed into at most $|V_c|$ subpaths $P = P_1P_2P_3 \dots$, where the endpoints of each subpath P_i are boundary nodes, and P_i is a shortest path in $G_{i \bmod 2}$.*

Proof. Consider a decomposition of $P = P_1P_2P_3 \dots$ into maximal subpaths such that the subpath P_i consists of nodes of $G_{i \bmod 2}$. Since r and v are boundary nodes, and since the boundary nodes are the only nodes common to both G_0 and G_1 , each subpath P_i starts and ends on a boundary node. If P_i were not a shortest path in $G_{i \bmod 2}$ between its endpoints, replacing P_i in P with a shorter path would yield a shorter r -to- v path, a contradiction.

It remains to show that there are at most $|V_c|$ subpaths in the decomposition of P . Since P is simple, each node, and in particular each boundary node appears in P at most once. Hence there can be at most $|V_c| - 1$ non-empty subpaths in the decomposition of P . Note, however, that if P starts with an arc of G_0 then P_1 is a trivial empty path from r to r . Hence, P can be decomposed into at most $|V_c|$ subpaths. ■

Lemma 3.4 gives rise to a DP solution for calculating the from- r distances to nodes of C , which resembles the Bellman-Ford algorithm. The pseudocode is given in Fig. 3.2. Note that, at this level of abstraction, there is nothing novel about this DP, and it is very similar to the one in Figure 3-1. Our contribution is in an efficient implementation of Step 4.

<ol style="list-style-type: none"> 1: $e_0[v] = \infty$ for all $v \in V_c$ 2: $e_0[r] = 0$ 3: for $j = 1, 2, 3, \dots, V_c$ 4: $e_j[v] = \left\{ \begin{array}{l} \min_{w \in V_c} \{e_{j-1}[w] + \delta_1[w, v]\}, \text{ if } j \text{ is odd} \\ \min_{w \in V_c} \{e_{j-1}[w] + \delta_0[w, v]\}, \text{ if } j \text{ is even} \end{array} \right\}, \forall v \in V_c$
--

Figure 3-4: Pseudocode for calculating the from- r distances in G to all nodes in V_c using just δ_0 and δ_1 .

The algorithm consists of $|V_c|$ iterations. On odd iterations, it uses the boundary-to-boundary distances in G_1 , and on even iterations it uses the boundary-to-boundary distances in G_0 .

Lemma 3.5 *After the table e_j is updated by the algorithm, $e_j[v]$ is the length of a shortest path in G from r to v that can be decomposed into at most j subpaths $P = P_1P_2P_3 \dots P_j$, where the endpoints of each subpath P_i are boundary nodes, and P_i is a shortest path in $G_{i \bmod 2}$.*

Proof. By induction on j . For the base case, e_0 is initialized to be infinity for all nodes other than r , trivially satisfying the lemma. For $j > 0$, assume that the lemma holds for $j - 1$, and let

P be a shortest path in G that can be decomposed into $P_1P_2 \dots P_j$ as above. Consider the prefix P' , $P' = P_1P_2 \dots P_{j-1}$. P' is a shortest r -to- w path in G that can be decomposed into at most $j - 1$ subpaths as above for some boundary node w . Hence, by the inductive hypothesis, when e_j is updated in Line 4, $e_{j-1}[w]$ already stores the length of P' . Thus $e_j[v]$ is updated in line 4 to be at most $e_{j-1}[w] + \delta_{j \bmod 2}[w, v]$. Since, by definition, $\delta_{j \bmod 2}[w, v]$ is the length of the shortest path in $G_{j \bmod 2}$ from w to v , it follows that $e_j[v]$ is at most the length of P . For the opposite direction, since for any boundary node w , $e_{j-1}[w]$ is the length of some path that can be decomposed into at most $j - 1$ subpaths as above, $e_j[v]$ is updated in Line 4 to the length of some path that can be decomposed into at most j subpaths as above. Hence, since P is the shortest such path, $e_j[v]$ is at least the length of P . ■

From Lemma 3.4 and Lemma 3.5, it immediately follows that the table $e_{|V_c|}$ stores the from- r shortest path distances in G . We now show how to perform all the minimizations in the j^{th} iteration of Line 4 in $O(|V_c|\alpha(|V_c|))$ time. Let $i = j \bmod 2$, so this iteration uses distances in G_i . Since all boundary nodes lie on the boundary of a single face of G_i , there is a natural cyclic clockwise order $v_1, v_2, \dots, v_{|V_c|}$ on the nodes in V_c . Define a $|V_c| \times |V_c|$ matrix A with elements $A_{k\ell} = e_{j-1}(v_k) + \delta_i(v_k, v_\ell)$. Note that computing all minima in Line 4 is equivalent to finding the column-minima of A . We define the *upper triangle* of A to be the elements of A on or above the main diagonal. More precisely, the upper triangle of A is the portion $\{A_{k\ell} : k \leq \ell\}$ of A . Similarly, the lower triangle of A consists of all the elements on or below the main diagonal of A .

Lemma 3.6 *For any four indices k, k', ℓ, ℓ' such that either $A_{k\ell}, A_{k\ell'}, A_{k'\ell}$ and $A_{k'\ell'}$ are all in A 's upper triangle, or are all in A 's lower triangle (i.e., either $1 \leq k \leq k' \leq \ell \leq \ell' \leq |V_c|$ or $1 \leq \ell \leq \ell' \leq k \leq k' \leq |V_c|$), the convex Monge property holds:*

$$A_{k\ell} + A_{k'\ell'} \geq A_{k\ell'} + A_{k'\ell}.$$

Proof. Consider the case $1 \leq k \leq k' \leq \ell \leq \ell' \leq |V_c|$, as in Fig. 3-5. Since G_i is planar, any pair of paths in G_i from k to ℓ and from k' to ℓ' must cross at some node w of G_i . Let $b_k = e_{j-1}(v_k)$ and let $b_{k'} = e_{j-1}(v_{k'})$. Let $\Delta(u, v)$ denote the u -to- v distance in G_i for any nodes u, v of G_i . Note

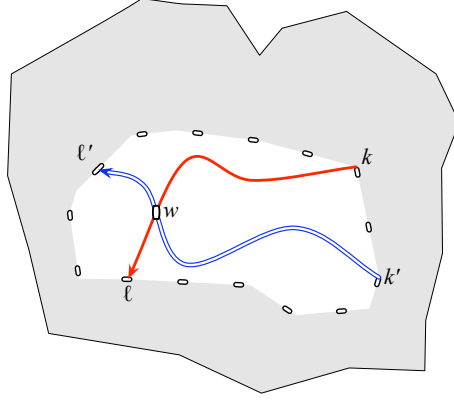


Figure 3-5: Nodes $k < k' < l < l'$ in clockwise order on the boundary nodes. Paths from k to l and from k' to l' must cross at some node w . This is true both in the internal and the external subgraphs of G

that $\Delta(u, v) = \delta_i(u, v)$ for $u, v \in V_c$. We have

$$\begin{aligned}
 A_{k,\ell} + A_{k',\ell'} &= (b_k + \Delta(v_k, w) + \Delta(w, v_\ell)) \\
 &\quad + (b_{k'} + \Delta(v_{k'}, w) + \Delta(w, v_{\ell'})) \\
 &= (b_k + \Delta(v_k, w) + \Delta(w, v_{\ell'})) \\
 &\quad + (b_{k'} + \Delta(v_{k'}, w) + \Delta(w, v_\ell)) \\
 &\geq (b_k + \Delta(v_k, v_{\ell'})) + (b_{k'} + \Delta(v_{k'}, v_\ell)) \\
 &= (b_k + \delta_i(v_k, v_{\ell'})) + (b_{k'} + \delta_i(v_{k'}, v_\ell)) \\
 &= A_{k,\ell'} + A_{k',\ell}
 \end{aligned}$$

The case $(1 \leq \ell \leq \ell' \leq k \leq k' \leq |V_c|)$ is similar. ■

Lemma 3.7 *A single iteration of the DP in Fig. 3-4 can be computed in $O(|V_c|\alpha(|V_c|))$ time.*

Proof. We need to show how to find the column-minima of the matrix A . We compute the column-minima of A 's lower and upper triangles separately, and obtain A 's column-minima by comparing the two values obtained for each column.

It follows directly from Lemma 3.6 that replacing the upper triangle of A with blanks yields a falling staircase matrix. By [51], the column-minima of this falling staircase matrix can be computed in $O(|V_c|\alpha(|V_c|))$ time. Another consequence of Lemma 3.6 is that the column-minima of the upper triangle of A may also be computed using the algorithm in [51]. To see this consider a counterclockwise ordering of the nodes of $|V_c|$ $v'_1, v'_2, \dots, v'_{|V_c|}$ such that $v'_k = v_{|V_c|+1-k}$. This reverses the order of both the rows and the columns of A , thus turning its upper triangle into a lower triangle. Again, replacing the upper triangle of this matrix with blanks yields a falling staircase matrix.

We thus conclude that A 's column-minima can be computed in $O(2|V_c| \cdot \alpha(|V_c|) + |V_c|) = O(|V_c| \cdot \alpha(|V_c|))$ time. Note that we never actually compute and store the entire matrix A as this would take $O(|V_c|^2)$ time. We compute the entries necessary for the computation on the fly in $O(1)$ time per element. ■

Lemma 3.7 shows that the time it takes the DP described in Fig. 3-4 to compute the distances between r and all nodes of V_c is $O(|V_c|^2 \cdot \alpha(|V_c|))$. We have thus proved Theorem 3.3. The choice of separator ensures $|V_c| = O(\sqrt{n})$, so this computation is performed in $O(n\alpha(n))$ time.

3.3 The Replacement-Paths Problem in Planar Graphs

Recall that given a directed planar graph with non-negative arc lengths and two nodes s and t , the *replacement-paths problem* asks to compute, for every arc e in the shortest path between s and t , the length of an s -to- t shortest path that avoids e .

In this section we show how to modify the algorithm of Emek et al. [34] to obtain an $O(n \log^2 n)$ running time for the replacement-paths problem. This is another example of using a Monge property for finding minima in a matrix. Similarly to Section 3.2, we deal with a matrix whose upper triangle satisfies a Monge property. However, the minima search problem is restricted to rectangular portions of that upper triangle. Hence, each such rectangular portion is entirely Monge (rather than falling staircase) so the SMAWK algorithm of Aggarwal et al. [3] can be used (rather than Klawe and Kleitman's algorithm [51]).

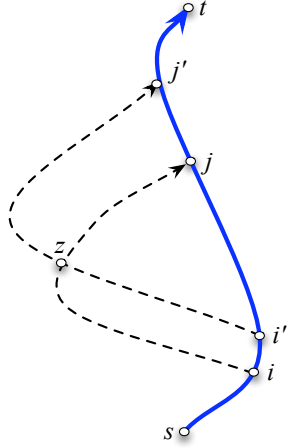


Figure 3-6: The s - t shortest path P is shown in solid blue. Paths of type Q_2 (dashed black) do not cross P . Two LL paths (i.e., leaving and entering P from the left) are shown. For $i < i' < j < j'$, the ij path and the $i'j'$ path must cross at some node z .

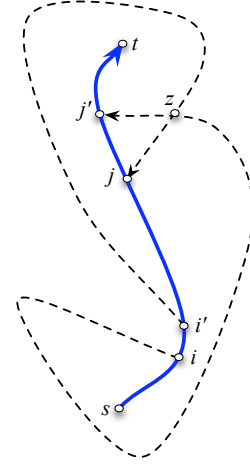


Figure 3-7: The s - t shortest path P is shown in solid blue. Paths of type Q_2 (dashed black) do not cross P . Two LR paths (i.e., leaving P from the left and entering P from the right) are shown. For $i < i' < j < j'$, the ij' path and the $i'j$ path must cross at some node z .

Let $P = (u_1, u_2, \dots, u_{p+1})$ be the shortest path from $s = u_1$ to $t = u_{p+1}$ in the graph G . Consider the replacement s -to- t path Q that avoids the arc e in P . Q can be decomposed as $Q_1Q_2Q_3$ where Q_1 is a prefix of P , Q_3 is a suffix of P , and Q_2 is a subpath from some u_i to some u_j that avoids any other vertex in P . If in a clockwise traversal of the arcs incident to some node u_i of P , starting from the arc (u_i, u_{i+1}) we encounter an arc e before we encounter the arc (u_{i-1}, u_i) , then we say that e is *to the right* of P . Otherwise, e is *to the left* of P . The first arc of Q_2 can be left or right of P and the last arc of Q_2 can be left or right of P . In all four cases Q_2 never crosses P (see Fig. 3-6 and Fig. 3-7).

For nodes x, y , the x -to- y distance is denoted by $\delta_G(x, y)$. The distances $\delta_G(s, u_i)$ and $\delta_G(u_i, t)$ for $i = 1, \dots, p + 1$ are computed from P in $O(p)$ time, and stored in a table.

The $p \times p$ matrix $\widehat{\text{len}}_{d,d'}$ is defined in [34] as follows: for any $1 \leq i \leq p$ and $1 \leq j \leq p$, let $\widehat{\text{len}}_{d,d'}(i, j)$ be the length of the shortest s -to- t path of the form $Q_1Q_2Q_3$ described above where Q_2 starts at u_i via a left-going arc if $d = L$ or a right-going arc if $d = R$, and Q_2 ends at u_{j+1} via a left-going arc if $d' = L$ and a right-going arc if $d' = R$. The length of Q_2 is denoted $\text{PAD-query}_{G,d,d'}(i, j)$. It can be computed in $O(\log n)$ time by a single query to a data structure

that Emek et al. call PADO (Path Avoiding Distance Oracle). Thus, we can write

$$\widehat{\text{len}}_{d,d'}(i, j) = \delta_G(s, u_i) + \text{PAD-query}_{G,d,d'}(i, j) + \delta_G(u_{j+1}, t),$$

and query any entry of $\widehat{\text{len}}_{d,d'}$ in $O(\log n)$ time.

The $O(n \log^3 n)$ time-complexity of Emek et al. arises from the recursive calls to the `District` procedure. We next give an alternative description of their algorithm. This new description is slightly simpler and makes it easy to explain the use of SMAWK.

Let $\text{range}(i)$ denote the rectangular portion of the matrix $\widehat{\text{len}}_{d,d'}$ defined by rows $1, \dots, i$ and columns i, \dots, p . With this definition the length of the replacement s -to- t path that avoids the edge (u_i, u_{i+1}) is equal to the minimal element in $\text{range}(i)$. Since $d \in \{L, R\}$ and $d' \in \{L, R\}$, we need to take the minimum among these four rectangular portions corresponding to the four possible $\widehat{\text{len}}$ matrices. The replacement-paths problem thus reduces to computing the minimum element in $\text{range}(i)$ for every $i = 1, 2, \dots, p$, and every $d, d' \in \{L, R\}$.

Given some $1 \leq a < b \leq p$ and some $d, d' \in \{L, R\}$, `District`(a, b) computes the row and column-minima of the rectangular portion of the matrix $\widehat{\text{len}}_{d,d'}$ defined by rows a to $\lfloor (a+b)/2 \rfloor$ and columns $\lfloor (a+b)/2 \rfloor$ to b . Initially, `District` is called with $a = 1$ and $b = p$. This, in particular, computes the minimum of $\text{range}(\lfloor p/2 \rfloor)$. Then, `District`($a, \lfloor (a+b)/2 \rfloor - 1$) and `District`($\lfloor (a+b)/2 \rfloor + 1, b$) are called recursively. Notice that the previous call to `District`(a, b), together with the current call to `District`($a, \lfloor (a+b)/2 \rfloor - 1$) suffice for computing all row and column-minima of $\text{range}(\lfloor p/4 \rfloor)$ (and hence also the global minimum of $\text{range}(\lfloor p/4 \rfloor)$), as illustrated in Fig. 3-8. Similarly, `District`(a, b), together with `District`($\lfloor (a+b)/2 \rfloor + 1, b$) suffice for computing all row and column-minima of $\text{range}(\lfloor 3p/4 \rfloor)$. The recursion stops when $b - a \leq 1$. Therefore, the depth of the recursion for `District`($1, p$) is $O(\log p)$, and it computes the minimum of $\text{range}(i)$ for all $1 \leq i \leq p$.

Emek et al. show how to compute `District`(a, b) in $O((b-a) \log^2(b-a) \log n)$ time, leading to a total of $O(n \log^3 n)$ time for computing `District`($1, p$). They use a divide and conquer technique to compute the row and column-minima in each of the rectangular areas encountered along the computation. Our contribution is in showing that instead of divide-and-conquer one can

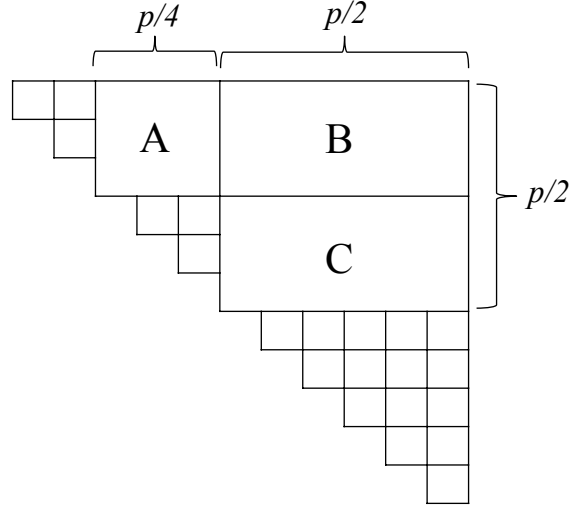


Figure 3-8: The upper triangular fragment of the matrix $\widehat{\text{len}}_{d,d'}$. The procedure $\text{District}(1, p)$ computes the row and column-minima of the area $B \cup C$, and therefore also the minimum element in $B \cup C$ (i.e. in $\text{range}(p/2)$). Then, $\text{District}(1, p/2)$ computes the row and column-minima of the area A . Together, they enable finding the row and column-minima of $A \cup B$ (i.e. in $\text{range}(p/4)$), and in particular the minimum element in $\text{range}(p/4)$.

use SMAWK to find those minima. This enables computing $\text{District}(a, b)$ in $O((b-a) \log(b-a) \log n)$ time, which leads to a total of $O(n \log^2 n)$ time for $\text{District}(1, p)$, as shown by the following lemmas.

Lemma 3.8 *The upper triangle of $\widehat{\text{len}}_{d,d'}$ satisfies a Monge property.*

Proof. Note that adding $\delta_G(s, u_i)$ to all of the elements in the i^{th} row or $\delta_G(u_{j+1}, t)$ to all elements in the j^{th} column preserves the Monge property. Therefore, it suffices to show that the upper triangle of $\text{PAD-query}_{G,d,d'}$ satisfies a Monge property.

When $d = d'$, the proof is essentially the same as that of Lemma 3.6 because the Q_2 paths have the same crossing property as the paths in Lemma 3.6. This is illustrated in Fig. 3-6. We thus establish that the convex Monge property holds.

When $d \neq d'$, Lemma 3.6 applies but with the convex Monge property replaced with the concave Monge property. To see this, consider the crossing paths in Fig. 3-7. In contrast to Fig. 3-6, this time the crossing paths are i -to- j' and i' -to- j . ■

Lemma 3.9 *Procedure `District(a, b)` can be computed in $O((b - a) \log(b - a) \log n)$ time.*

Proof. Recall that, for every pair d, d' , `District(a, b)` first computes the row and column-minima of the rectangular submatrix of $\widehat{\text{len}}_{d,d'}$ defined by rows a to $\lfloor (a + b)/2 \rfloor$ and columns $\lfloor (a + b)/2 \rfloor$ to b . By Lemma 3.8, this entire submatrix has a Monge property. In the case of the convex Monge property, we can use SMAWK to find all row and column-minima of the submatrix. In the case of the concave Monge property, we cannot directly apply SMAWK. By negating all the elements we get a convex Monge matrix but we are now looking for its row and column *maxima*. As discussed in Section 3.1.1, SMAWK can be used to find row and column-maxima of a convex Monge matrix. Thus, in both cases, we find the row and column-minima of the submatrix by querying only $O(b - a)$ entries each in $O(\log n)$ time for a total of $O((b - a) \log n)$ time. Therefore, $T(a, b)$, the time it takes to compute `District(a, b)` is given by

$$T(a, b) = T(a, (a + b)/2) + T((a + b)/2, b) + O((b - a) \log n) = O((b - a) \log(b - a) \log n).$$

■

Chapter 4

Combining Compression and Total-Monotonicity

In Chapter 2 we discussed the acceleration via compression technique, and in Chapter 3 we saw speedups based on totally monotone and Monge matrices. In this chapter we show how to combine these two ideas in order to accelerate the *string edit distance* computation.

Recall the edit distance between two strings over a fixed alphabet Σ is the minimum cost of transforming one string into the other via a sequence of character deletion, insertion, and replacement operations [100]. The cost of these elementary editing operations is given by some scoring function which induces a metric on strings over Σ . The simplest and most common scoring function is the Levenshtein distance [60] which assigns a uniform score of 1 for every operation (this is the scoring function we considered in Section 1).

Let us describe again the standard DP solution for computing the edit distance between a pair of strings $A = a_1a_2 \cdots a_N$ and $B = b_1b_2 \cdots b_N$. The $(N + 1) \times (N + 1)$ DP table T is defined so that $T[i, j]$ stores the edit distance between $a_1a_2 \cdots a_i$ and $b_1b_2 \cdots b_j$. The computation is done according to the base-case rules given by $T[0, 0] = 0$, $T[i, 0] = T[i - 1, 0] +$ the cost of deleting

a_i , and $T[0, j] = T[0, j - 1] +$ the cost of inserting b_j , and according to the following DP:

$$T[i, j] = \min \begin{cases} T[i - 1, j] + \text{the cost of deleting } a_i \\ T[i, j - 1] + \text{the cost of inserting } b_j \\ T[i - 1, j - 1] + \text{the cost of replacing } a_i \text{ with } b_j \end{cases} \quad (4.1)$$

Note that as T has $(N + 1)^2$ entries, the time-complexity of the algorithm above is $O(N^2)$.

To this date, this quadratic upper-bound has never been substantially improved for general strings. However, there are known (acceleration via compression) techniques for breaking this bound in case the strings are known to compress well under a particular compression scheme. It is therefore natural to ask whether there is a single edit-distance algorithm that can exploit the compressibility properties of strings under any compression method, even if each string is compressed using a different compression. In this chapter we answer this question by using *straight-line programs* (SLPs). These provide a generic platform for representing many popular compression schemes including the LZ-family, Run-Length Encoding, Byte-Pair Encoding, and dictionary methods. Our speedup combines acceleration via SLP compression with the speedup technique of the previous section, that takes advantage of totally monotone matrices,

4.1 Accelerating String Edit Distance Computation

In Chapter 2, we have already mentioned that the “acceleration via compression” approach has been successfully applied to many classical problems on strings. Regarding edit-distance computation, Bunke and Csirik presented a simple algorithm for computing the edit-distance of strings that compress well under *Run Length Encoding* (RLE) [19]. This algorithm was later improved in a sequence of papers [8, 9, 29, 69] to an algorithm running in time $O(nN)$, for strings of total length N that encode into run-length strings of total length n . In [29], an algorithm with the same time complexity was given for strings that are compressed under LZW-LZ78, where n again is the length of the compressed strings. Note that this algorithm is also $O(N^2 / \lg N)$ in the worst-case for any strings over constant-size alphabets.

The first paper to break the quadratic time-barrier of edit-distance computation was the seminal paper of Masek and Paterson [72], who applied the "Four-Russians technique" to obtain a running-time of $O(N^2/\lg N)$ for any pair of strings, and of $O(N^2/\lg^2 N)$ assuming a unit-cost RAM model. Their algorithm essentially exploits repetitions in the strings to obtain the speed-up, and so in many ways it can also be viewed as compression-based. In fact, one can say that their algorithm works on the "naive compression" that all strings over constant-sized alphabets have. A drawback of the Masek and Paterson algorithm is that it can only be applied when the given scoring function is rational. That is, when all costs of editing operations are rational numbers. Note that this restriction is indeed a limitation in biological applications [29, 72]. For this reason, the algorithm in [29] mentioned above was designed specifically to work for arbitrary scoring functions. We mentioned also Bille and Farach-Colton [16] who extend the Masek and Paterson algorithm to general alphabets.

Notice that all known techniques for improving on the $O(N^2)$ time bound of edit-distance computation, essentially apply acceleration via compression. In addition, apart from RLE, LZW-LZ78, and the naive compression of the Four-Russians technique, we do not know how to efficiently compute edit-distance under other compression schemes. For example, no algorithm is known which substantially improves $O(N^2)$ on strings which compress well under LZ77. Such an algorithm would be interesting since there are various types of strings that compress much better under LZ77 than under RLE or LZW-LZ78.

In this chapter, we introduce a general compression-based edit-distance algorithm based on straight-line programs (SLPs) and the SMAWK algorithm for finding row minima in totally monotone matrices. Our algorithm can be used on strings which compress well under all the above compression schemes, including LZ77, and even if each string is compressed using a different compression scheme. This is due to the ability of SLPs to capture these various compressions.

4.1.1 Straight-line programs

Recall the definition of a *straight-line program* (SLP) from Section 2.3.4. An SLP is a context-free grammar generating exactly one string. Moreover, only two types of productions are allowed:

$X_i \rightarrow a$ where a is a unique terminal, and $X_i \rightarrow X_p X_q$ with $i > p, q$ where X_1, \dots, X_n are the grammar variables. Each variable appears exactly once on the left hand side of a production. The string represented by a given SLP is a unique string corresponding to the last nonterminal X_n .

We define the size of an SLP to be n , the number of variables (or productions) it has. The length of the strings that is generated by the SLP is denoted by N . It is important to observe that SLPs can be exponentially smaller than the string they generate.

Rytter [85] proved that the resulting encoding of most compression schemes including the LZ-family, RLE, Byte-Pair Encoding, and dictionary methods, can be transformed to straight-line programs quickly and without large expansion¹. In particular, consider an LZ77 encoding [104] with n' blocks for a string of length N . Rytter's algorithm produces an SLP-representation with size $n = O(n' \log N)$ of the same string, in $O(n)$ time. Moreover, n lies within a $\log N$ factor from the size of a *minimal* SLP describing the same string. This gives us an efficient logarithmic approximation of minimal SLPs, since computing the LZ77 encoding of a string can be done in linear-time. Note also that any string compressed by the LZ78-LZW encoding can be transformed directly into a straight-line program within a constant factor.

4.1.2 Our results

Due to Rytter's results, SLPs are perfect candidates for achieving our goal of generalizing compression-based edit-distance algorithms. Our main result is the following.

Theorem 4.1 *Let \mathcal{A} and \mathcal{B} be two SLPs of total size n that respectively generate two string A and B of total size N . Then, given \mathcal{A} and \mathcal{B} , one can compute the edit-distance between A and B in $O(n^{1.4} N^{1.2})$ time for any rational scoring function.*

We can remove the dependency of rational scoring schemes in Theorem 4.1, recalling that arbitrary scoring schemes are important for biological applications. We obtain the following secondary result for arbitrary scoring functions:

¹Important exceptions of this list are statistical compressors such as Huffman or arithmetic coding, as well as compressions that are applied after a Burrows-Wheeler transformation.

Theorem 4.2 *Let \mathcal{A} and \mathcal{B} be two SLPs of total size n that respectively generate two strings A and B of total size N . Then, given \mathcal{A} and \mathcal{B} , one can compute the edit-distance between A and B in $O(n^{1.34}N^{1.34})$ time for any arbitrary scoring function.*

Finally, we explain how the Four-Russians technique can also be incorporated into our SLP edit-distance scheme. We obtain a very simple algorithm that matches the performance of [29] in the worst-case. That is, we obtain a Four-Russians like algorithm with an $\Omega(\lg N)$ speed-up which can handle arbitrary scoring functions, unlike the Masek and Paterson algorithm which works only for rational functions.

4.1.3 Related Work

Rytter et al. [50] was the first to consider SLPs in the context of pattern matching, and other subsequent papers also followed this line [59, 76]. In [85] and [61] Rytter and Lifshits took this work one step further by proposing SLPs as a general framework for dealing with pattern matching algorithms that are accelerated via compression. However, the focus of Lifshits was on determining whether or not these problems are polynomial in n or not. In particular, he gave an $O(n^3)$ -time algorithm to determine equality of SLPs [61], and he established hardness for the edit distance [62], and even for the hamming distance problems [61]. Nevertheless, Lifshits posed as an open problem the question of whether or not there is an $O(nN)$ edit-distance algorithm for SLPs. Here, our focus is on algorithms which break the quadratic $O(N^2)$ time-barrier, and therefore all algorithms with running-times between $O(nN)$ and $O(N^2)$ are interesting for us.

Recently, Tiskin [97] gave an $O(nN^{1.5})$ algorithm for computing the longest common subsequence between two SLPs, an algorithm which can be extended at constant-factor cost to compute the edit-distance between the SLPs under any rational scoring function. Observe that our algorithm for arbitrary scoring functions in Theorem 4.2 is already faster than Tiskin's algorithm for most values of N and n . Also, it has the advantage of being much more simpler to implement. As for our main algorithm of Theorem 4.1, our faster running-time is achieved also by utilizing some of the techniques used by Tiskin in a more elaborate way.

4.2 The DIST Table and Total-Monotonicity

The central DP tool we use in our algorithms is the *DIST* table, a simple and handy data-structure which was originally introduced by Apostolico et al. [6], and then further developed by others in [29, 86]. In the following section we briefly review basic facts about this tool, mostly its total-monotonicity property. We begin with the so-called DP grid, a graph representation of edit-distance computation on which *DIST* tables are defined.

Consider the standard DP formulation (4.1) for computing the edit-distance between two strings $A = a_1a_2 \cdots a_N$ and $B = b_1b_2 \cdots b_N$. The *DP grid* associated with this program, is an acyclic-directed graph which has a vertex for each entry of T (see Figure 4-1). The vertex corresponding to $T[i, j]$ is associated with a_i and b_j , and has incoming edges according to (4.1) – an edge from $T[i - 1, j]$ whose weight is the cost of deleting a_i , an edge from $T[i, j - 1]$ whose weight is the cost of inserting b_j , and an edge from $T[i - 1, j - 1]$ whose weight is the cost of replacing a_i with b_j . The *value* at the vertex corresponding to $T[i, j]$ is the value stored in $T[i, j]$, *i.e.* the edit-distance between the length i prefix of A and the length j prefix of B . Using the DP grid G , we reduce the problem of computing the edit-distance between A and B to the problem of computing the weight of the lightest path from the upper-left corner to bottom-right corner in G .

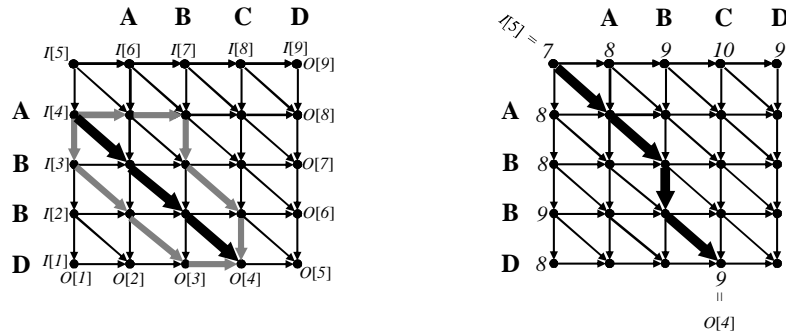


Figure 4-1: A subgraph of a Levenshtein distance DP graph. On the left, $DIST[4, 4]$ (in bold) gives the minimum-weight path from $I[4]$ to $O[4]$. On the right, the value 9 of $O[4]$ is computed by $\min_i I[i] + DIST[i, 4]$.

We will work with sub-grids of the DP grid that will be referred to as *blocks*. The *input vertices* of a block are all vertices in the first row and column of the block, while its *output vertices* are

all vertices in the last row and column. Together, the input and output vertices are referred to as the *boundary* of the block. The substrings of A and B associated with the block are defined in the straightforward manner according to its first row and column. Also, for convenience purposes, we will order the input and output vertices, with both orderings starting from the vertex in bottom-leftmost corner of the block, and ending at the vertex in the upper-rightmost corner. The i th input vertex and j th output vertex are the i th and j th vertices in these orderings. We next give the definition of *DIST* tables, defined over blocks of G .

Definition 4.3 (*DIST* [6]) *Let G' be a block in G with x input vertices and x output vertices. The *DIST* table corresponding to G' is an $x \times x$ matrix, with $DIST[i, j]$ storing the weight of the minimum-weight path from the i th input to the j th output in G , and otherwise ∞ if no such paths exists.*

It is important to notice that the values at the output vertices of a block are completely determined by that values at its input and its corresponding *DIST* table. In particular, if $I[i]$ and $O[j]$ are the values at the i th input vertex and j th output vertex of a block G' of G , then

$$O[j] = \min_{1 \leq i \leq x} I[i] + DIST[i, j]. \quad (4.2)$$

Equation 4.2 implies not only the input-output relation of the DP values of a block, but also that the values at the output vertices can be computed in linear time from the values at the input vertices. Indeed, by (4.2), the values at the output vertices of G' are given by the column minima of the matrix $I + DIST$. Furthermore, by a simple modification of all ∞ values in $I + DIST$, we get a matrix with the *concave Monge* property [29] (See Chapter 3 Section 3.1.1 for the definition of Monge matrices). Recall, the SMAWK algorithm of Aggarwal et al. [3] that computes all column minima of an $x \times x$ Monge matrix by querying only $O(x)$ elements of the matrix. It follows that using SMAWK we can compute the output values of G' in $O(x)$ time.

We now describe how to efficiently construct the *DIST* table corresponding to a block in G . Observe that this can be done quite easily in $O(x^3)$ time, for blocks with boundary size $O(x)$, by computing the standard DP table between every prefix of A against B and every prefix of B

against A . Each of these DP tables contains all values of a particular row in the $DIST$ table. In [6], Apostolico et al. show an elegant way to reduce the time-complexity of this construction to $O(x^2 \lg x)$. In the case of rational scoring functions, the complexity can be further reduced to $O(x^2)$ as shown by Schmidt [86].

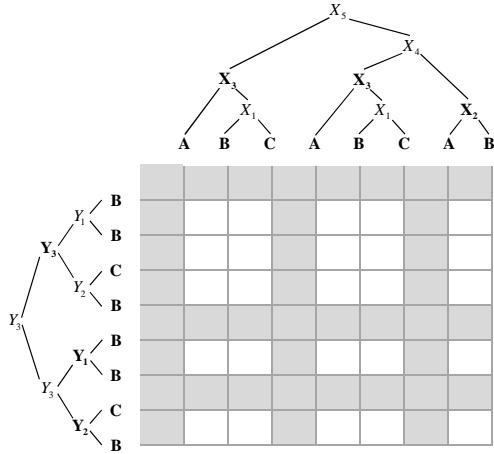
4.3 Acceleration via Straight-Line Programs

In the following section we describe a generic framework for accelerating the edit distance computation of two strings which are given by their SLP representation. This framework will later be used for explaining all our algorithms. We will refer to this framework as the *block edit-distance* procedure.

Let \mathcal{A} and \mathcal{B} be two SLP representations of a pair of strings A and B , and for ease of presentation assume that $|\mathcal{A}| = |\mathcal{B}| = n$ and $|A| = |B| = N$. Recall the definition in Section 4.2 for the DP grid corresponding to A and B . The general idea behind the block edit-distance procedure is to partition this grid into disjoint blocks, and then to compute the edit-distance between A and B at the cost of computing the values at the boundary vertices of each block. This is achieved by building in advance a repository containing all $DIST$ tables corresponding to blocks in the partition. To efficiently construct this repository, we show how to partition the grid in a way which induces many block repeats. This is possible by utilizing substring repeats in A and B that are captured in \mathcal{A} and \mathcal{B} , and imply block repeats in the partitioning of G . The edit-distance of A and B is then computed by propagating the DP values at the boundary vertices of the blocks using the $DIST$ tables in the repository and SMAWK. Before giving a complete description of this algorithm, we need to introduce the notion of xy -partition.

Definition 4.4 (xy -partition) *An xy -partition is a partitioning of G into disjoint blocks such that every block has boundary of size $O(x)$, and there are $O(y)$ blocks in each row and column. In addition, we require each pair of substrings of A and B associated with a block to be generated by a pair of SLP variables in \mathcal{A} and \mathcal{B} .*

An xy -partition of G is a partition with a specific structure, but more importantly, one where



An xy -partition of an edit distance graph for two SLPs generating the strings “ABCABCAB” and “BBCBBBCB”. The white blocks are the ones of the partition and their corresponding SLP variables are marked in bold. Notice that there are nine blocks in the partition but only six of them are distinct.

Figure 4-2: An xy -partition.

each substring is generated by a unique SLP variable of \mathcal{A} and \mathcal{B} . This latter requirement allows us to exploit the repetitions of A and B captured by their SLPs. We next give a complete description of the block edit distance procedure. It assumes an xy -partition of G has already been constructed.

Block Edit Distance

1. Construct a repository with the $DIST$ tables corresponding to each block in the xy -partition.
2. Fill-in the first row and column of G using the standard base-case rules.
3. In top-to-bottom and left-to-right manner, identify the next block in the partition of G and use its input and the repository to compute its output using (4.2).
4. Use the outputs in order to compute the inputs of the next blocks using (4.1).
5. The value in the bottom-rightmost cell is the edit distance of A and B .

Apart from the repository construction in step 1, all details necessary for implementing the block edit-distance procedure are by now clear. Indeed, steps 2 and 5 are trivial, and step 4 is done via the standard DP formulation of (4.1). Furthermore, the SMAWK computation of output values of a block, given its input values plus its corresponding $DIST$ table (step 3), is explained in Section 4.2. We next show that, as we are working with xy -partitions where each block is

associated with an SLP variable, we can compute a repository containing all *DIST* necessary which is rather small.

The first crucial observation for this, is that any two blocks associated with the same pair of substrings A' and B' have the same *DIST* table. This is immediate since any such pair of blocks have identical edge-weights.

Observation 4.5 *A pair of substrings A', B' uniquely identify the *DIST* table of a block.*

Since we required each substring in the xy -partition of G to be generated by some SLP variable, the above observation actually suggests that the number of different *DIST* tables is bounded by the number of variable pairs $X \in \mathcal{A}$ and $Y \in \mathcal{B}$:

Observation 4.6 *The number of different *DIST* tables corresponding to any xy -partition is $O(n^2)$.*

Therefore, combining the two observations above, we know that a repository containing a *DIST* tables for each SLP variable pair $X \in \mathcal{A}$ and $Y \in \mathcal{B}$ will not be too large, and that it will contain a table corresponding to each block in our given xy -partition at hand. We can therefore state the following lemma:

Lemma 4.7 *The block edit-distance procedure runs in $O(n^2x^2 \lg x + Ny)$ time.*

Proof. We analyze the time complexity of each step in the block edit-distance procedure separately. Step 1 can be performed in $O(n^2x^2 \lg x)$ time, as we can construct every *DIST* table in $O(x^2 \lg x)$ time (see Section 4.2), and the total number of such distinct matrices is $O(n^2)$. Step 2 can be done trivially in $O(N)$ time. Then, step 3 takes $O(x)$ time per block by using the SMAWK algorithm as explained in Section 4.2. Step 4 also takes $O(x)$ time per block as it only computes the values in the $O(x)$ vertices adjacent to the output vertices. The total time complexity of steps 3 and 4 is thus equal to the total number of boundary vertices in the xy -partition of G , and therefore to $O(Ny)$. Accounting for all steps together, this gives us the time complexity stated in the lemma.

■

4.3.1 Constructing an xy -partition

We now discuss the missing component of the block edit-distance procedure, namely the construction of xy -partitions. In particular, we complete the proof of Theorem 4.2 by showing how to efficiently construct an xy -partition where $y = O(nN/x)$ for every $x \leq N$. Together with Lemma 4.7, this implies an $O(n^{\frac{4}{3}}N^{\frac{4}{3}}\lg^{\frac{1}{3}}N) = O(n^{1.34}N^{1.34})$ time algorithm for arbitrary scoring functions by considering $x = N^{\frac{2}{3}}/(n\lg N)^{\frac{1}{3}}$. In the remainder of this section we prove the following lemma.

Lemma 4.8 *For every $x \leq N$ there exists an xy -partition with $y = O(nN/x)$. Moreover, this partition can be found in $O(N)$ time.*

To prove the lemma, we show that for every SLP \mathcal{A} generating a string A and every $x \leq N$, one can partition A into $O(nN/x)$ disjoint substrings, each of length $O(x)$, such that every substring is generated by some variable in \mathcal{A} . This defines a subset of variables in both input SLPs which together defined our desired xy -partition. To partition A , we first identify $O(N/x)$ grammar variables in \mathcal{A} each generating a disjoint substring of length between x and $2x$. We use these variables to partition A . We then show that the substrings of A that are still not associated with a variable can each be generated by $O(n)$ additional variables. Furthermore, these $O(n)$ variables each generate a string of length bounded by x . We add all such variables to our partition of A for a total of $O(nN/x)$ variables.

Consider the parse tree of \mathcal{A} . We want to identify $O(nN/x)$ *key-vertices* such that every key-vertex generates a substring of length $O(x)$, and A is a concatenation of substrings generated by key-vertices. We start by marking every vertex v that generates a substring of length greater than x as a key-vertex iff both children of v generate substrings of length smaller than x . This gives us $\ell \leq N/x$ key-vertices so far, each generating a substring of length $\Theta(x)$ (see Figure 4-3). But we are still not guaranteed that these vertices cover A entirely.

To fix this, consider the ordering v_1, v_2, \dots, v_ℓ on the current key-vertices induced by a left-to-right postorder traversal of the parse tree. This way, v_{i+1} is “to the right of” v_i . If every v_i generates the substring A_i then $A = A'_1A_1A'_2A_2 \cdots A'_\ell A_\ell A'_{\ell+1}$, where every A_i is of length $\Theta(x)$, and every

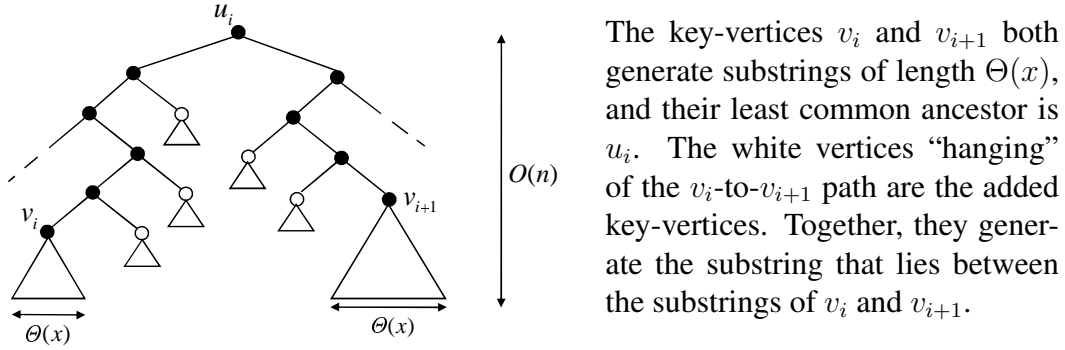


Figure 4-3: A closer look on the parse tree of an SLP \mathcal{A} .

A'_i is the “missing” substring of A that lies between A_{i-1} and A_i . We now show that every A'_i is a concatenation of substrings of length smaller than x generated by at most $O(n)$ vertices.

Let u_i be the lowest common ancestor of v_i and v_{i+1} and let P_i (resp. P_{i+1}) be the unique path between u_i and v_i (resp. v_{i+1}). For every vertex $v \in P_i - \{u_i\}$ such that v 's left child is also in P_i mark v 's right child as a key-vertex. Similarly, for every vertex $v \in P_{i+1} - \{u_i\}$ such that v 's right child is also in P_{i+1} mark v 's left child as a key-vertex. It is easy to verify that A'_i is the concatenation of substrings generated by these newly marked key-vertices. There are at most $2n$ of these key-vertices since the depth of the parse tree is bounded by the number of different SLP variables. Moreover, they each generate a substring of length smaller than x for the following reason. Assume for contradiction that one of them generates a string of length greater than x . This would imply the existence of some vertex between v_i and v_{i+1} in the v_1, v_2, \dots, v_ℓ ordering.

To conclude, we showed that $A = A'_1 A_1 A'_2 A_2 \dots A'_\ell A_\ell A'_{\ell+1}$ where $\ell \leq N/x$, every A_i is of length $\Theta(x)$ and is generated by one vertex, and every A'_i is a concatenation of $O(n)$ substrings each of length smaller than x and generated by one vertex. Overall, we get that $y = O(n\ell) = O(nN/x)$ vertices suffice to generate A for every $x \leq N$. It is easy to see that we can identify these vertices in $O(N)$ time thus proving Lemma 4.8. By choosing $x = N^{2/3}/(n \lg N)^{1/3}$, and using the block edit distance time complexity of Lemma 4.7, this implies an $O(n^{1.34} N^{1.34})$ time algorithm for arbitrary scoring functions.

4.4 Improvement for Rational Scoring Functions

In this section we show that in the case of rational scoring functions, the time complexity of the block edit distance procedure can be reduced substantially by using a recursive construction of the *DIST* tables. In particular, we complete the proof of Theorem 4.1 by showing that in this case the repository of *DIST* tables can be computed in $O(n^2x^{1.5})$ time. This implies an $O(n^{1.4}N^{1.2})$ time algorithm for rational scoring functions by considering $x = N^{0.8}/n^{0.4}$ and the xy -partition with $y = nN/x$.

Before we describe how to compute the repository in $O(n^2x^{1.5})$ time, we need to introduce some features that *DIST* tables over rational scoring functions have. The first property, discovered by Schmidt [86], is what is known as the succinct representation property: Any $x \times x$ *DIST* table can be succinctly stored using only $O(x)$ space. This follows from considering the vector obtained by subtracting a *DIST* column from the column to its right, and observing that this vector has only a constant number of value changes. The second property is that succinct representations allow to efficiently merge two *DIST* tables. That is, if D_1 and D_2 are two *DIST* tables, one between a pair of substrings A' and B' and the other between A' and B'' , then we refer to the *DIST* table between A' and $B'B''$ as the product of *merging* D_1 and D_2 . A recent important result of Tiskin [96] shows how to utilize the succinct representation of *DIST* tables in order to merge two succinct $x \times x$ *DIST* tables in $O(x^{1.5})$ time.

Lemma 4.9 *The block edit distance algorithm runs in $O(n^2x^{1.5} + Ny)$ time in case the underlying scoring function is rational.*

Proof. To prove the lemma it suffices to show how to compute the repository of *DIST* tables in step 1 of the block edit-distance procedure in $O(n^2x^{1.5})$ time, in case the underlying scoring function is rational. We will work with succinct representations of the *DIST* tables as described above. Say $X \rightarrow X_pX_q$ and $Y \rightarrow Y_sY_t$ are two rules in the SLPs \mathcal{A} and \mathcal{B} respectively. To compute the *DIST* table that corresponds to the strings generated by X and Y , we first recursively compute the four *DIST* tables that correspond to the pairs (X_p, Y_s) , (X_p, Y_t) , (X_q, Y_s) , and (X_q, Y_t) . We then merge these four tables to obtain the *DIST* table that corresponds to (X, Y) . To do so we

use Tiskin’s procedure to merge (X_p, Y_s) with (X_p, Y_t) into $(X_p, Y_s T_t)$, then merge (X_q, Y_s) with (X_q, Y_t) into $(X_q, Y_s T_t)$, and finally we merge $(X_p, Y_s T_t)$ and $(X_q, Y_s T_t)$ into $(X_p X_q, Y_s T_t) = (X, Y)$. This recursive procedure computes each succinct *DIST* table by three merge operations, each taking $O(x^{1.5})$ time and $O(x)$ space. Since the number of different *DIST* tables is bounded by $O(n^2)$, the $O(n^2 x^{1.5})$ time for constructing the repository follows. ■

To conclude, we have shown an $O(n^2 x^{1.5} + Ny)$ time algorithm for computing the edit distance. Using the xy -partition from Lemma 4.8 with $x = N^{0.8}/n^{0.4}$ and $y = nN/x$, we get a time complexity of $O(n^{1.4} N^{1.2})$.

4.5 Four-Russian Interpretation

In the previous sections we showed how SLPs can be used to speed up the edit distance computation of strings that compress well under some compression scheme. In this section, we conclude the presentation of our SLP framework by presenting an $\Omega(\lg N)$ speed-up for strings that do not compress well under any compression scheme. To do so, we adopt the Four Russians approach of Masek and Paterson [72] that utilizes a naive property that every string over a fixed alphabet has. Namely, that short enough substrings must appear many times. However, while the Masek and Paterson algorithm can only handle rational scoring functions, the SLP version that we propose can handle arbitrary scoring functions.

Consider a string A of length N over an alphabet Σ . The parse tree of the naive SLP \mathcal{A} is a complete binary tree with N leaves². This way, for every $x \leq N$ we get that A is the concatenation of $O(N/x)$ substrings each of length $\Theta(x)$ and each can be generated by some variable in \mathcal{A} . This partition of A suggests an xy -partition in which $y = N/x$. At first glance, this might seem better than the partition guarantee of Lemma 4.8 in which $y = nN/x$. However, notice that in the naive SLP we have $n \geq N$ so we can not afford to compute a repository of $O(n^2)$ *DIST* tables.

To overcome this problem, we choose x small enough so that $|\Sigma|^x$, the number of possible substrings of length x , is small. In particular, by taking $x = \frac{1}{2} \log_{|\Sigma|} N$ we get that the number of

²We assume without loss of generality that N is a power of 2.

possible substrings of length x is bounded by $|\Sigma|^x = \sqrt{N}$. This implies an xy -partition in which $x = \frac{1}{2} \log_{|\Sigma|} N$, $y = N/x$, and the number of distinct blocks n' is $O(N)$. Using this partition, we get that the total construction time of the *DIST* repository is $O(n'x^2 \lg x)$. Similar to Lemma 4.7, we get that the total running time of the block edit distance algorithm is $O(n'x^2 \lg x + Ny)$ which gives $O(N^2 / \lg N)$.

Chapter 5

Partial Tables

In typical DP settings, a table is filled in its entirety, where each cell corresponds to some subproblem. For example, the naive DP table used for solving the edit distance problem between two strings A and B of length n (see previous section) has n^2 cells. This entire table is filled row by row, where the cells stores the edit distance between any two prefixes of A and B . In some cases however, by changing the DP, it is possible to compute asymptotically less cells of the table (i.e. only a subset of relevant subproblems). The advantage of this technique is that a small change in the DP can make a significant improvement in both time and space complexity. Although the complexity analysis can be difficult, the actual DPs are very simple to describe and to implement.

In this chapter, we use this technique for DPs that measure the similarity between two trees. In Section 5.1 we explain how the similarity of trees is captured by the *tree edit distance* metric, and we give a simple and unified presentation of the two well-known tree edit distance algorithms on which our algorithm is based. These algorithms, as well as ours, are based on the same DP and differ only in the actual DP entries that they compute. We present and analyze the $O(n^3)$ time-complexity of our algorithm in Section 5.2, and show how the space-complexity can be reduced to $O(n^2)$ in Section 5.3. In Section 5.4 we prove the optimality of our algorithm among the family of *decomposition strategy algorithms* by tightening the known lower bound of $\Omega(n^2 \log^2 n)$ to $\Omega(n^3)$, matching our algorithm's running time. This family, which also includes the previous fastest algorithms, consists of all algorithms that are based on the same DP and differ only in the

relevant subproblems that they compute. Finally, in Section 5.5, we show how tree edit distance can be used in computational biology for comparing RNA sequences.

5.1 Tree Edit Distance

The problem of comparing trees occurs in diverse areas such as structured text databases like XML, computer vision, compiler optimization, natural language processing, and computational biology [15, 25, 54, 88, 95]. The *tree edit distance* metric is a common similarity measure for rooted ordered trees. It was introduced by Tai in the late 1970's [95] as a generalization of the well-known string edit distance problem [100]. Let F and G be two rooted trees with a left-to-right order among siblings and where each vertex is assigned a label from an alphabet Σ . The *edit distance* between F and G is the minimum cost of transforming F into G by a sequence of elementary operations consisting of deleting and relabeling existing nodes, as well as inserting new nodes (allowing at most one operation to be performed on each node). These operations are illustrated in Fig. 5-1. Formally, given a non-root node v in F with parent v' , *relabel* changes the label of v , *delete* removes v and sets the children of v as the children of v' (the children are inserted in the place of v as a subsequence in the left-to-right order of the children of v'), and *insert* (the complement of delete) connects a new node v as a child of some v' in F making v the parent of a consecutive subsequence of the children of v' . The cost of the elementary operations is given by two functions, c_{del} and c_{match} , where $c_{\text{del}}(\tau)$ is the cost of deleting or inserting a vertex with label τ , and $c_{\text{match}}(\tau_1, \tau_2)$ is the cost of changing the label of a vertex from τ_1 to τ_2 . Since a deletion in F is equivalent to an insertion in G and vice versa, we can focus on finding the minimum cost of a sequence of just deletions and relabelings in both trees that transform F and G into isomorphic trees.

Previous results. To state running times, we need some basic notation. Let n and m denote the sizes $|F|$ and $|G|$ of the two input trees, ordered so that $n \geq m$. Let n_{leaves} and m_{leaves} denote the corresponding number of leaves in each tree, and let n_{height} and m_{height} denote the corresponding height of each tree, which can be as large as n and m respectively.

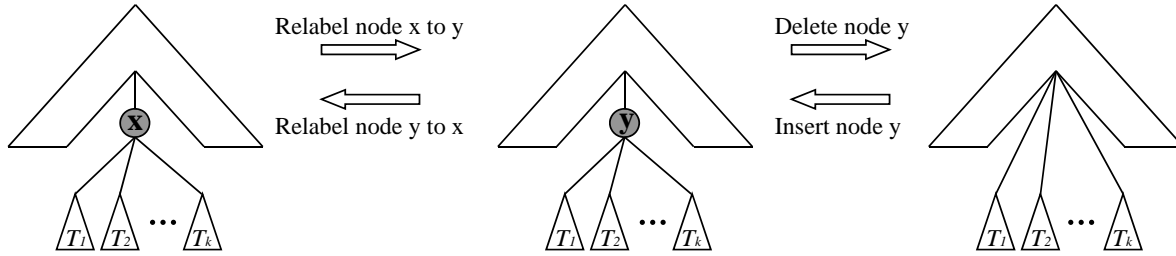


Figure 5-1: The three editing operations on a tree with vertex labels.

Tai [95] presented the first algorithm for computing tree edit distance, which requires $O(n_{\text{leaves}}^2 m_{\text{leaves}}^2 nm)$ time and space, and thus has a worst-case running time of $O(n^3 m^3) = O(n^6)$. Shasha and Zhang [88] improved this result to an $O(\min\{n_{\text{height}}, n_{\text{leaves}}\} \cdot \min\{m_{\text{height}}, m_{\text{leaves}}\} \cdot nm)$ time algorithm using $O(nm)$ space. In the worst case, their algorithm runs in $O(n^2 m^2) = O(n^4)$ time. Klein [53] improved this result to a worst-case $O(m^2 n \log n) = O(n^3 \log n)$ time algorithm using $O(nm)$ space. These last two algorithms are based on closely related DPs, and both present different ways of computing only a subset of a larger DP table; these entries are referred to as *relevant subproblems*. In [32], Dulucq and Touzet introduced the notion of a *decomposition strategy* (see Section 5.1.3) as a general framework for algorithms that use this type of DP, and proved a lower bound of $\Omega(nm \log n \log m)$ time for any such strategy.

Many other solutions have been developed; see [7, 15, 98] for surveys. The most recent development is by Chen [26], who presented a different approach that uses results on fast matrix multiplication. Chen's algorithm uses $O(nm + nm_{\text{leaves}}^2 + n_{\text{leaves}} m_{\text{leaves}}^{2.5})$ time and $O(n + (m + n_{\text{leaves}}^2) \min\{n_{\text{leaves}}, n_{\text{height}}\})$ space. In the worst case, this algorithm runs in $O(nm^{2.5}) = O(n^{3.5})$ time. Among all these algorithms, Klein's is the fastest in terms of worst-case time complexity, and previous improvements to Klein's $O(n^3 \log n)$ time bound were achieved only by constraining the edit operations or the scoring scheme [25, 87, 89, 102].

Our results. We present a new algorithm for computing the tree edit distance that falls into the same *decomposition strategy* framework of [32, 53, 88]. In the worst case, our algorithm requires $O(nm^2(1 + \log \frac{n}{m})) = O(n^3)$ time and $O(nm)$ space. The corresponding sequence of edit

operations can easily be obtained within the same time and space bounds. We therefore improve upon all known algorithms in the worst case time complexity. Our approach is based on Klein’s, but whereas the recursion scheme in Klein’s algorithm is determined by just one of the two input trees, in our algorithm the recursion depends alternately on both trees. Furthermore, we prove a worst-case lower bound of $\Omega(nm^2(1+\log \frac{n}{m}))$ time for all decomposition strategy algorithms. This bound improves the previous best lower bound of $\Omega(nm \log n \log m)$ time [32], and establishes the optimality of our algorithm among all decomposition strategy algorithms. Finally, we show how our algorithm can be adapted (with the same time and space bounds) to a computational biology setting in which k consecutive deletions are more probable to occur than k separate deletions.

Notations Both the existing algorithms and ours compute the edit distance of finite ordered Σ -labeled forests. These are forests that have a left-to-right order among siblings and each vertex is assigned a label from a given finite alphabet Σ such that two different vertices can have the same label or different labels. The unique empty forest/tree is denoted by \emptyset . The vertex set of a forest F is written simply as F , as when we speak of a vertex $v \in F$. For a forest F and $v \in F$, $\sigma(v)$ denotes the label of v , F_v denotes the subtree of F rooted at v , and $F - v$ denotes the forest F after deleting v . The special case of $F - \text{root}(F)$ where F is a tree and $\text{root}(F)$ is its root is denoted F° . The leftmost and rightmost trees of a forest F are denoted by L_F and R_F and their roots by ℓ_F and r_F . We denote by $F - L_F$ the forest F after deleting the entire leftmost tree L_F ; similarly $F - R_F$. A left-to-right postorder traversal of F is the postorder traversal of all its trees L_F, \dots, R_F from left to right. For a tree T , the postorder traversal is defined recursively as the postorder traversal of the forest T° followed by a visit of $\text{root}(T)$ (as apposed to a preorder traversal that first visits $\text{root}(T)$ and then T°). A forest obtained from F by a sequence of any number of deletions of the leftmost and rightmost roots is called a *subforest* of F .

Given forests F and G and vertices $v \in F$ and $w \in G$, we write $c_{\text{del}}(v)$ instead of $c_{\text{del}}(\sigma(v))$ for the cost of deleting or inserting $\sigma(v)$, and we write $c_{\text{match}}(v, w)$ instead of $c_{\text{match}}(\sigma(v), \sigma(w))$ for the cost of relabeling $\sigma(v)$ to $\sigma(w)$. $\delta(F, G)$ denotes the edit distance between the forests F and G .

Because insertion and deletion costs are the same (for a node of a given label), insertion in

one forest is tantamount to deletion in the other forest. Therefore, the only edit operations we need to consider are relabelings and deletions of nodes in both forests. We next briefly present the algorithms of Shasha and Zhang, and of Klein. This presentation, inspired by the tree similarity survey of Bille [15], is somewhat different from the original presentations and is essential for understanding our algorithm and the idea of partial tables.

5.1.1 Shasha and Zhang’s Algorithm

Given two forests F and G of sizes n and m respectively, the following lemma is easy to verify. Intuitively, the lemma says that in any sequence of edit operations the two rightmost roots in F and G must either be matched with each other or else one of them is deleted.

Lemma 5.1 ([88]) $\delta(F, G)$ can be computed as follows:

- $\delta(\emptyset, \emptyset) = 0$
- $\delta(F, \emptyset) = \delta(F - r_F, \emptyset) + c_{\text{del}}(r_F)$
- $\delta(\emptyset, G) = \delta(\emptyset, G - r_G) + c_{\text{del}}(r_G)$
- $\delta(F, G) = \min \begin{cases} \delta(F - r_F, G) + c_{\text{del}}(r_F), \\ \delta(F, G - r_G) + c_{\text{del}}(r_G), \\ \delta(R_F^\circ, R_G^\circ) + \delta(F - R_F, G - R_G) + c_{\text{match}}(r_F, r_G) \end{cases}$

Lemma 5.1 yields an $O(m^2n^2)$ DP solution. If we index the vertices of the forests F and G according to their left-to-right postorder traversal position, then entries in the DP table correspond to pairs (F', G') of subforests F' of F and G' of G where F' contains vertices $\{i_1, i_1 + 1, \dots, j_1\}$ and G' contains vertices $\{i_2, i_2 + 1, \dots, j_2\}$ for some $1 \leq i_1 \leq j_1 \leq n$ and $1 \leq i_2 \leq j_2 \leq m$.

However, as we will presently see, only $O(\min\{n_{\text{height}}, n_{\text{leaves}}\} \cdot \min\{m_{\text{height}}, m_{\text{leaves}}\} \cdot nm)$ different *relevant subproblems* are encountered by the recursion computing $\delta(F, G)$. We calculate the number of *relevant subforests* of F and G independently, where a forest F' (respectively G') is a relevant subforest of F (respectively G) if it occurs in the computation of $\delta(F, G)$. Clearly,

multiplying the number of relevant subforests of F and of G is an upper bound on the total number of relevant subproblems.

We now count the number of relevant subforests of F ; the count for G is similar. First, notice that for every node $v \in F$, F_v° is a relevant subproblem. This is because the recursion allows us to delete the rightmost root of F repeatedly until v becomes the rightmost root; we then match v (i.e., relabel it) and get the desired relevant subforest. A more general claim is stated and proved later on in Lemma 5.3. We define

$$\text{keyroots}(F) = \{\text{the root of } F\} \cup \{v \in F \mid v \text{ has a left sibling}\}.$$

It is easy to see that every relevant subforest of F is a prefix (with respect to the postorder indices) of F_v° for some node $v \in \text{keyroots}(F)$. If we define v 's collapse depth $\text{cdepth}(v)$ to be the number of keyroot ancestors of v , and $\text{cdepth}(F)$ to be the maximum $\text{cdepth}(v)$ over all nodes $v \in F$, we get that the total number of relevant subforest of F is at most

$$\sum_{v \in \text{keyroots}(F)} |F_v| = \sum_{v \in F} \text{cdepth}(v) \leq \sum_{v \in F} \text{cdepth}(F) = |F| \text{cdepth}(F).$$

This means that given two trees, F and G , of sizes n and m we can compute $\delta(F, G)$ in $O(\text{cdepth}(F)\text{cdepth}(G)nm) = O(n_{\text{height}}m_{\text{height}}nm)$ time. Shasha and Zhang also proved that for any tree T of size n , $\text{cdepth}(T) \leq \min\{n_{\text{height}}, n_{\text{leaves}}\}$, hence the result. In the worst case, this algorithm runs in $O(m^2n^2) = O(n^4)$ time.

5.1.2 Klein's Algorithm

Klein's algorithm is based on a recursion similar to Lemma 5.1. Again, we consider forests F and G of sizes $|F| = n \geq |G| = m$. Now, however, instead of recursing always on the rightmost roots of F and G , we recurse on the leftmost roots if $|L_F| \leq |R_G|$ and on the rightmost roots otherwise. In other words, the "direction" of the recursion is determined by the (initially) larger of the two forests. We assume the number of relevant subforests of G is $O(m^2)$; we have already established that this is an upper bound.

We next show that Klein’s algorithm yields only $O(n \log n)$ relevant subforests of F . The analysis is based on a technique called *heavy path decomposition* [43, 93]. We mark the root of F as *light*. For each internal node $v \in F$, we pick one of v ’s children with maximal number of descendants and mark it as *heavy*, and we mark all the other children of v as *light*. We define $\text{ldepth}(v)$ to be the number of light nodes that are proper ancestors of v in F , and $\text{light}(F)$ as the set of all light nodes in F . It is easy to see that for any forest F and vertex $v \in F$, $\text{ldepth}(v) \leq \log |F| + O(1)$. Note that every relevant subforest of F is obtained by some $i \leq |F_v|$ consecutive deletions from F_v for some light node v . Therefore, the total number of relevant subforests of F is at most

$$\sum_{v \in \text{light}(F)} |F_v| \leq \sum_{v \in F} 1 + \text{ldepth}(v) \leq \sum_{v \in F} (\log |F| + O(1)) = O(|F| \log |F|).$$

Thus, we get an $O(m^2 n \log n) = O(n^3 \log n)$ algorithm for computing $\delta(F, G)$.

5.1.3 The Decomposition Strategy Framework

Both Klein’s and Shasha and Zhang’s algorithms are based on Lemma 5.1, and both compute a different subset of relevant subproblems. The difference between them lies in the choice of when to recurse on the rightmost roots and when on the leftmost roots. The family of *decomposition strategy* algorithms based on this lemma was formalized by Dulucq and Touzet in [32].

Definition 5.2 (Strategy, Decomposition Algorithm) *Let F and G be two forests. A strategy is a mapping from pairs (F', G') of subforests of F and G to $\{\text{left}, \text{right}\}$. A decomposition algorithm is an algorithm based on Lemma 5.1 with the directions chosen according to a specific strategy.*

Each strategy is associated with a specific set of recursive calls (or a DP algorithm). The strategy of Shasha and Zhang’s algorithm is $S(F', G') = \text{right}$ for all F', G' . The strategy of Klein’s algorithm is $S(F', G') = \text{left}$ if $|L_{F'}| \leq |R_{F'}|$, and $S(F', G') = \text{right}$ otherwise. Notice that Shasha and Zhang’s strategy does not depend on the input trees, while Klein’s strategy depends only on the larger input tree. Dulucq and Touzet proved a lower bound of $\Omega(mn \log m \log n)$ time for any decomposition strategy algorithm.

The following lemma states that every decomposition algorithm computes the edit distance between every two root-deleted subtrees of F and G .

Lemma 5.3 *Given a decomposition algorithm with strategy S , the pair (F_v°, G_w°) is a relevant subproblem for all $v \in F$ and $w \in G$ regardless of the strategy S .*

Proof. First note that a node $v' \in F_v$ (respectively, $w' \in G_w$) is never deleted or matched before v (respectively, w) is deleted or matched. Consider the following computational path:

- Delete from F until v is either the leftmost or the rightmost root.
- Next, delete from G until w is either the leftmost or the rightmost root.

Let (F', G') denote the resulting subproblem. There are four cases to consider.

1. v and w are the rightmost (leftmost) roots of F' and G' , and $S(F', G') = \text{right (left)}$.

Match v and w to get the desired subproblem.

2. v and w are the rightmost (leftmost) roots of F' and G' , and $S(F', G') = \text{left (right)}$.

Note that at least one of F', G' is not a tree (since otherwise this is case (1)). Delete from one which is not a tree. After a finite number of such deletions we have reduced to case (1), either because S changes direction, or because both forests become trees whose roots are v, w .

3. v is the rightmost root of F' , w is the leftmost root of G' .

If $S(F', G') = \text{left}$, delete from F' ; otherwise delete from G' . After a finite number of such deletions this reduces to one of the previous cases when one of the forests becomes a tree.

4. v is the leftmost root of F' , w is the rightmost root of G' .

This case is symmetric to (3). ■

5.2 Our Tree Edit Distance Algorithm

In this section we present our algorithm for computing $\delta(F, G)$ given two trees F and G of sizes $|F| = n \geq |G| = m$. The algorithm recursively uses a decomposition strategy in a divide-and-conquer manner to achieve $O(nm^2(1 + \log \frac{n}{m})) = O(n^3)$ running time in the worst case. For clarity we describe the algorithm recursively and analyze its time complexity. In Section 5.3 we give an explicit DP and prove that the space complexity can be made $O(mn) = O(n^2)$.

Before presenting our algorithm, let us try to develop some intuition. We begin with the observation that Klein’s strategy always determines the direction of the recursion according to the F -subforest, even in subproblems where the F -subforest is smaller than the G -subforest. However, it is not straightforward to change this since even if at some stage we decide to choose the direction according to the other forest, we must still make sure that all subproblems previously encountered are entirely solved. At first glance this seems like a real obstacle since apparently we only add new subproblems to those that are already computed. Our key observation is that there are certain subproblems for which it is worthwhile to choose the direction according to the *currently* larger forest, while for other subproblems we had better keep choosing the direction according to the *originally* larger forest.

The *heavy path* of a tree F is the unique path starting from the root (which is light) along heavy nodes. Consider two trees, F and G , and assume we are given the distances $\delta(F_v^\circ, G_w^\circ)$ for all $v \in F$ and $w \in G$. By lemma 5.3, these are relevant subproblems for any decomposition strategy algorithm. How would we go about computing $\delta(F, G)$ in this case? Using Shasha and Zhang’s strategy would require $O(|F||G|)$ time, while using Klein’s strategy would take $O(|F||G|^2)$ time. Let us focus on Klein’s strategy since Shasha and Zhang’s strategy is independent of the trees. Note that even if we were not given the distance $\delta(F_u^\circ, G_w^\circ)$ for a node u on the heavy path of F , we would still be able to solve the problem in $O(|F||G|^2)$ time. To see why, note that in order to compute the relevant subproblem $\delta(F_u, G_w)$, we must compute all the subproblems required for solving $\delta(F_u^\circ, G_w^\circ)$ even if $\delta(F_u^\circ, G_w^\circ)$ is given.

We next define the set $\text{TopLight}(F)$ to be the set of roots of the forest obtained by removing the heavy path of F . Note that $\text{TopLight}(F)$ is the set of light nodes with $\text{ldepth} = 1$ in F (see

The algorithm's first step makes sure that F is the larger forest, and the second step makes sure that $\delta(F_{v'}^\circ, G_w^\circ)$ is computed and stored for all v' not in the heavy path of F and for all $w \in G$. Note that the strategy in the third step is equivalent to Klein's strategy for binary trees. For higher valence trees, this variant first makes all left deletions and then all right deletions, while Klein's strategy might change direction many times. They are equivalent in the important sense that both delete the heavy child last. The algorithm is evidently a decomposition strategy algorithm, since for all subproblems, it either deletes or matches the leftmost or rightmost roots. The correctness of the algorithm follows from the correctness of decomposition strategy algorithms in general.

Time Complexity. We show that our algorithm has a worst-case running time of $O(m^2n(1 + \log \frac{n}{m})) = O(n^3)$. We proceed by counting the number of subproblems computed in each step of the algorithm. We call a subproblem trivial if at least one of the forests in this subproblem is empty. Obviously, the number of distinct trivial subproblems is $O(n^2)$. Let $R(F, G)$ denote the number of non-trivial relevant subproblems encountered by the algorithm in the course of computing $\delta(F, G)$. From now on we will only count non-trivial subproblems, unless explicitly indicated otherwise.

We observe that any tree F has the following two properties:

$$(*) \quad \sum_{v \in \text{TopLight}(F)} |F_v| \leq |F|. \text{ Because } F_v \text{ and } F_{v'} \text{ are disjoint for all } v, v' \in \text{TopLight}(F).$$

$$(**) \quad |F_v| < \frac{|F|}{2} \text{ for every } v \in \text{TopLight}(F). \text{ Otherwise } v \text{ would be a heavy node.}$$

In step (2) we compute $\delta(F_v, G)$ for all $v \in \text{TopLight}(F)$. Hence, the number of subproblems encountered in this step is $\sum_{v \in \text{TopLight}(F)} R(F_v, G)$. For step (3), we bound the number of relevant subproblems by multiplying the number of relevant subforests in F and in G . For G , we count all possible $O(|G|^2)$ subforests obtained by left and right deletions. Note that for any node v' not in the heavy path of F , the subproblem obtained by matching v' with any node w in G was already computed in step (2). This is because any such v' is contained in F_v for some $v \in \text{TopLight}(F)$, so $\delta(F_{v'}^\circ, G_w^\circ)$ is computed in the course of computing $\delta(F_v, G)$ (by Lemma 5.3). Furthermore, note that in step (3), a node v on the heavy path of F cannot be matched or deleted until the remaining subforest of F is precisely the tree F_v . At this point, both matching v or deleting v result in the

same new relevant subforest F_v° . This means that we do not have to consider matchings of nodes when counting the number of relevant subproblems in step (3). It suffices to consider only the $|F|$ subforests obtained by deletions according to our strategy. Thus, the total number of new subproblems encountered in step (3) is bounded by $|G|^2|F|$.

We have established that if $|F| \geq |G|$ then

$$R(F, G) \leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} R(F_v, G)$$

and if $|F| < |G|$ then

$$R(F, G) \leq |F|^2|G| + \sum_{w \in \text{TopLight}(G)} R(F, G_w)$$

We first show, by a crude estimate, that this leads to an $O(n^3)$ running time. Later, we analyze the dependency on m and n accurately.

Lemma 5.4 $R(F, G) \leq 4(|F||G|)^{3/2}$.

Proof. We proceed by induction on $|F| + |G|$. In the base case, $|F| + |G| = 0$, so both forests are empty and $R(F, G) = 0$. For the inductive step there are two symmetric cases. If $|F| \geq |G|$ then $R(F, G) \leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} R(F_v, G)$. Hence, by the induction hypothesis,

$$\begin{aligned} R(F, G) &\leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} 4(|F_v||G|)^{3/2} = |G|^2|F| + 4|G|^{3/2} \sum_{v \in \text{TopLight}(F)} |F_v|^{3/2} \\ &\leq |G|^2|F| + 4|G|^{3/2} \sum_{v \in \text{TopLight}(F)} |F_v| \max_{v \in \text{TopLight}(F)} \sqrt{|F_v|} \\ &\leq |G|^2|F| + 4|G|^{3/2}|F| \sqrt{\frac{|F|}{2}} = |G|^2|F| + \sqrt{8}(|F||G|)^{3/2} \leq 4(|F||G|)^{3/2} \end{aligned}$$

Here we have used facts (*) and (**) and the fact that $|F| \geq |G|$. The case where $|F| < |G|$ is symmetric. ■

This crude estimate gives a worst-case running time of $O(n^3)$. We now analyze the dependence on m and n more accurately. Along the recursion defining the algorithm, we view step (2) as only

making recursive calls, but not producing any relevant subproblems. Rather, every new relevant subproblem is created in step (3) for a unique recursive call of the algorithm. So when we count relevant subproblems, we sum the number of new relevant subproblems encountered in step (3) over all recursive calls to the algorithm. We define sets $A, B \subseteq F$ as follows:

$$A = \{a \in \text{light}(F) : |F_a| \geq m\}$$

$$B = \{b \in F - A : b \in \text{TopLight}(F_a) \text{ for some } a \in A\}$$

Note that the root of F belongs to A . Intuitively, the nodes in both A and B are exactly those for which recursive calls are made with the entire G tree. The nodes in B are the last ones, along the recursion, for which such recursive calls are made. We count separately:

- (i) the relevant subproblems created in just step (3) of recursive calls $\delta(F_a, G)$ for all $a \in A$,
and
- (ii) the relevant subproblems encountered in the entire computation of $\delta(F_b, G)$ for all $b \in B$
(i.e., $\sum_{b \in B} R(F_b, G)$).

Together, this counts all relevant subproblems for the original $\delta(F, G)$. To see this, consider the original call $\delta(F, G)$. Certainly, the root of F is in A . So all subproblems generated in step (3) of $\delta(F, G)$ are counted in (i). Now consider the recursive calls made in step (2) of $\delta(F, G)$. These are precisely $\delta(F_v, G)$ for $v \in \text{TopLight}(F)$. For each $v \in \text{TopLight}(F)$, notice that v is either in A or in B ; it is in A if $|F_v| \geq m$, and in B otherwise. If v is in B , then all subproblems arising in the entire computation of $\delta(F_v, G)$ are counted in (ii). On the other hand, if v is in A , then we are in analogous situation with respect to $\delta(F_v, G)$ as we were in when we considered $\delta(F, G)$ (i.e., we count separately the subproblems created in step (3) of $\delta(F_v, G)$ and the subproblems coming from $\delta(F_u, G)$ for $u \in \text{TopLight}(F_v)$).

Earlier in this section, we saw that the number of subproblems created in step (3) of $\delta(F, G)$ is $|G|^2|F|$. In fact, for any $a \in A$, by the same argument, the number of subproblems created in step (3) of $\delta(F_a, G)$ is $|G|^2|F_a|$. Therefore, the total number of relevant subproblems of type (i) is $|G|^2 \sum_{a \in A} |F_a|$. For $v \in F$, define $\text{depth}_A(v)$ to be the number of proper ancestors of v that lie in

the set A . We claim that $\text{depth}_A(v) \leq 1 + \log \frac{n}{m}$ for all $v \in F$. To see this, consider any sequence a_0, \dots, a_k in A where a_i is a descendent of a_{i-1} for all $i \in [1, k]$. Note that $|F_{a_i}| \leq \frac{1}{2}|F_{a_{i-1}}|$ for all $i \in [1, k]$ since the a_i s are light nodes. Also note that $F_{a_0} \leq n$ and that $|F_{a_k}| \geq m$ by the definition of A . It follows that $k \leq \log \frac{n}{m}$, i.e., A contains no sequence of descendants of length $> 1 + \log \frac{n}{m}$. So clearly every $v \in F$ has $\text{depth}_A(v) \leq 1 + \log \frac{n}{m}$.

We now have the number of relevant subproblems of type (i) as

$$|G|^2 \sum_{a \in A} |F_a| \leq m^2 \sum_{v \in F} 1 + \text{depth}_A(v) \leq m^2 \sum_{v \in F} (2 + \log \frac{n}{m}) = m^2 n (2 + \log \frac{n}{m}).$$

The relevant subproblems of type (ii) are counted by $\sum_{b \in B} R(F_b, G)$. Using Lemma 5.4, we have

$$\sum_{b \in B} R(F_b, G) \leq 4|G|^{3/2} \sum_{b \in B} |F_b|^{3/2} \leq 4|G|^{3/2} \sum_{b \in B} |F_b| \max_{b \in B} \sqrt{|F_b|} \leq 4|G|^{3/2} |F| \sqrt{m} = 4m^2 n.$$

Here we have used the facts that $|F_b| < m$ and $\sum_{b \in B} |F_b| \leq |F|$ (since the trees F_b are disjoint for different $b \in B$). Therefore, the total number of relevant subproblems for $\delta(F, G)$ is at most $m^2 n (2 + \log \frac{n}{m}) + 4m^2 n = O(m^2 n (1 + \log \frac{n}{m}))$. This implies:

Theorem 5.5 *The running time of the algorithm is $O(m^2 n (1 + \log \frac{n}{m}))$.* ■

5.3 A Space-Efficient DP formulation

The recursion presented in Section 5.2 for computing $\delta(F, G)$ translates into an $O(m^2 n (1 + \log \frac{n}{m}))$ time and space algorithm (by using memoization). Apart from the overhead of using recursion, the space complexity is an obvious drawback. In this section, we present a DP formulation of our algorithm that solves the relevant problems from smaller to larger, and requires only $O(mn)$ space. We achieve this by ordering the relevant subproblems in such a way that we need to record the edit distance of only $O(mn)$ relevant subproblems at any point in time. For simplicity, we assume the input trees F and G are binary. At the end of this section, we show how to remove this assumption.

The algorithm TED fills a global n by m table Δ with values $\Delta_{vw} = \delta(F_v^\circ, G_w^\circ)$ for all $v \in F$ and $w \in G$.

TED(F, G)

1: If $|F| < |G|$ do TED(G, F).

2: For every $v \in \text{TopLight}(F)$ do TED(F_v, G).

3: Fill Δ_{vw} for all $v \in \text{HeavyPath}(F)$ and $w \in G$.

Step 3 runs in $O(|F||G|^2)$ time and assumes Δ_{vw} has already been computed in step 2 for all $v \in F - \text{HeavyPath}(F)$ and $w \in G$ (see Section 5.2). In the remainder of this section we prove that it can be done in $O(|F||G|)$ space.

In step 3 we go through the nodes v_1, \dots, v_t on the heavy path of F starting with the leaf v_1 and ending with the root v_t where $t = |\text{HeavyPath}(F)|$. Throughout the computation we maintain a table T of size $|G|^2$. When we start handling v_p ($1 \leq p \leq t$), the table T holds the edit distance between $F_{v_{p-1}}$ and all possible subforests of G . We use these values to calculate the edit distance between F_{v_p} and all possible subforests of G and store the newly computed values back into T . We refer to the process of updating the entire T table (for a specific v_p) as a period. Before the first period, in which F_{v_1} is a leaf, we set T to hold the edit distance between \emptyset and G' for all subforests G' of G (this is just the cost of deleting G').

Note that since we assume F is binary, during each period the direction of our strategy does not change. Let $\text{left}(v)$ and $\text{right}(v)$ denote the left and right children of a node v . If $v_{p-1} = \text{right}(v_p)$, then our strategy is left throughout the period of v_p . Otherwise it is right. We now explain what goes into computing a period. This process, which we refer to as $\text{COMPUTEPERIOD}(v_p)$, both uses and updates tables T and Δ . At the heart of this procedure is a DP. Throughout this description we assume that our strategy is left. The right analogue is obvious. We now describe two simple subroutines that are called by $\text{COMPUTEPERIOD}(v_p)$.

If $F_{v_{p-1}}$ can be obtained from F_{v_p} by a series of left deletions, the *intermediate left subforest enumeration* with respect to $F_{v_{p-1}}$ and F_{v_p} is the sequence $F_{v_{p-1}} = F_0, F_1 \dots, F_k = F_{v_p}$ such that $F_{k'-1} = F_{k'} - \ell_{F_{k'}}$ for all $1 \leq k' \leq k = |F_{v_p}| - |F_{v_{p-1}}|$. This concept is illustrated in Fig. 5-

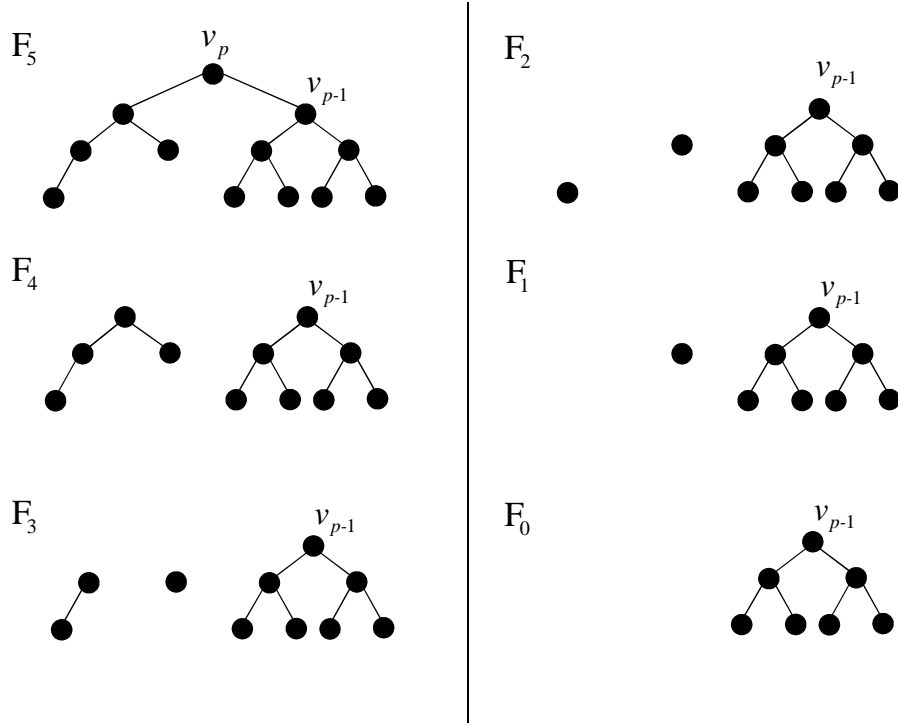


Figure 5-3: The *intermediate left subforest enumeration* with respect to $F_{v_{p-1}}$ and F_{v_p} is the sequence of forests $F_{v_{p-1}} = F_0, F_1, \dots, F_5 = F_{v_p}$.

3. The subroutine `INTERMEDIATELEFTSUBFORESTENUM`($F_{v_{p-1}}, F_{v_p}$) associates every $F_{k'}$ with $\ell_{F_{k'}}$ and lists them in the order of the intermediate left subforest enumerations with respect to $F_{v_{p-1}}$ and F_{v_p} . This is the order in which we access the nodes and subforests during the execution of `COMPUTEPERIOD`(v_p), so each access will be done in constant time. The intermediate left and right subforest enumerations required for all periods (i.e., for all of the v_{p_s} along the heavy path) can be prepared once in $O(|F|)$ time and space by performing $|F|$ deletions on F according to our strategy and listing the deleted vertices in reverse order.

Let $w_0, w_1, \dots, w_{|G|-1}$ be the right-to-left preorder traversal of a tree G . We define $G_{i,0}$ as the forest obtained from G by making i right deletions. Notice that the rightmost tree in $G_{i,0}$ is G_{w_i} (the subtree of G rooted at w_i). We further define $G_{i,j}$ as the forest obtained from G by first making i right deletions and then making j left deletions. Let $j(i)$ be the number of left deletions required to turn $G_{i,0}$ into the tree G_{w_i} . We can easily compute $j(0), \dots, j(|G|-1)$ in $O(|G|)$ time and space

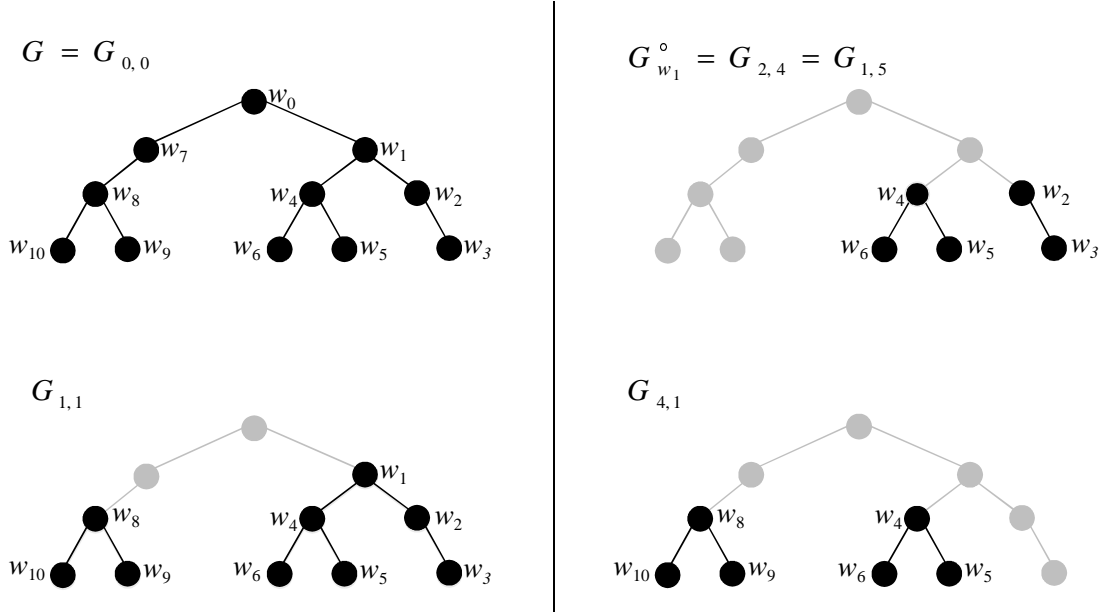


Figure 5-4: The indexing of various subforests (shown in solid black) of G (shown in gray). The right-to-left preorder traversal of G is $w_0, w_1, \dots, w_{|G|-1}$. The subforest $G_{i,j}$ is the forest obtained from G by first making i right deletions and then making j left deletions. All nonempty subforests of G are captured by all $0 \leq i \leq |G| - 1$ and $0 \leq j \leq j(i) = |G| - i - |G_{w_i}|$. The index of G itself is $G_{0,0}$. In the special case of $G_{w_1}^\circ = G_{2,4}$ we sometimes use the equivalent index $G_{1,5}$.

by noticing that $j(i) = |G| - i - |G_{w_i}|$. Note that distinct nonempty subforests of G are represented by distinct $G_{i,j}$ s for $0 \leq i \leq |G| - 1$ and $0 \leq j \leq j(i)$. For convenience, we sometimes refer to $G_{w_i}^\circ$ as $G_{i,j(i)+1}$ and sometimes as the equivalent $G_{i+1,j(i)}$. The two subforest are the same since the forest $G_{i,j(i)}$ is the tree G_{w_i} , so making another left deletion, namely $G_{i,j(i)+1}$ is the same as first making an extra right deletion, namely $G_{i+1,j(i)}$. The *left subforest enumeration* of all nonempty subforests of G is defined as

$$G_{|G|-1,j(|G|-1)}, \dots, G_{|G|-1,1}, G_{|G|-1,0}, \dots, G_{2,j(2)}, \dots, G_{2,1}, G_{2,0}, G_{1,j(1)}, \dots, G_{1,1}, G_{1,0}, G_{0,0}$$

The subroutine LEFTSUBFORESTENUM(G) associates every $G_{i,j}$ with the left deleted vertex $\ell_{G_{i,j}}$ and lists them in the order of the left subforest enumeration with respect to G , so that we will be able to access $\ell_{G_{i,j}}$ in this order in constant time per access. This procedure takes $O(|G|)$ time and space for each i by performing first i right deletions and then j left deletions, and listing the

left deleted vertices in reverse order. The entire subroutine therefore requires $O(|G|^2)$ time and space. The above definitions are illustrated in Fig. 5-4. There are obvious “right” analogues of everything we have just defined.

The pseudocode for $\text{COMPUTEPERIOD}(v_p)$ is given below. As we already mentioned, at the beginning of the period for v_p , the table T stores the distance between $F_{v_{p-1}}$ and all subforests of G and our goal is to update T with the distance between F_{v_p} and any subforest of G . For each value of i in decreasing order (the loop in Line 3), we compute a temporary table S of the distances between the forests $F_{k'}$ in the intermediate left subforest enumeration with respect to $F_{v_{p-1}}$ and F_{v_p} and the subforest $G_{i,j}$ for $0 \leq j \leq j(i)$ in the left subforest enumeration of G . Clearly, there are $O(|F||G|)$ such subproblems. The computation is done for increasing values of k' and decreasing values of j according to the basic relation in Line 4. Once the entire table S is computed, we update T , in Line 5, with the distances between $F_k = F_{v_p}$ and $G_{i,j}$ for all $0 \leq j \leq j(i)$. Note that along this computation we encounter the subproblem which consists of the root-deleted-trees $F_{v_p}^\circ = F_{k-1}$ and $G_{w_{i-1}}^\circ = G_{i,j(i-1)}$. In line 7, we store the value for this subproblem in $\Delta_{v_p, w_{i-1}}$. Thus, going over all possible values for i , the procedure updates the entire table T and all the appropriate entries in Δ , and completes a single period.

COMPUTEPERIOD(v_p)

Overwrites T with values $\delta(F_{v_p}, G')$ for all subforests G' of G , and fills in Δ with values $\delta(F_{v_p}^\circ, G_w^\circ)$ for every $w \in G$.

Assumes T stores $\delta(F_{v_{p-1}}, G')$ for all subforests G' of G , and $v_{p-1} = \text{right}(v_p)$ (if $v_{p-1} = \text{left}(v_p)$ then reverse roles of “left” and “right” below).

- 1: $F_0, \dots, F_k \leftarrow \text{IntermediateLeftSubforestEnum}(F_{v_{p-1}}, F_{v_p})$
- 2: $G_{|G|-1, j(|G|-1)}, \dots, G_{|G|-1, 0}, \dots, G_{1, j(1)}, \dots, G_{1, 0}, G_{0, j(0)}, \dots, G_{0, 0} \leftarrow \text{LeftSubforestEnum}(G)$
- 3: for $i = |G| - 1, \dots, 0$ do
- 4: compute table $S \leftarrow \left(\delta(F_{k'}, G_{i, j}) \right)_{\substack{k'=1, \dots, k \\ j=j(i), \dots, 0}}$ via the dynamic program:

$$\delta(F_{k'}, G_{i, j}) = \min \begin{cases} c_{\text{del}}(\ell_{F_{k'}}) + \delta(F_{k'-1}, G_{i, j}), \\ c_{\text{del}}(\ell_{G_{i, j}}) + \delta(F_{k'}, G_{i, j+1}), \\ c_{\text{match}}(\ell_{F_{k'}}, \ell_{G_{i, j}}) + \delta(L_{F_{k'}}^\circ, L_{G_{i, j}}^\circ) + \delta(F_{k'-1} - L_{F_{k'}}, G_{i, j} - L_{G_{i, j}}) \end{cases}$$
- 5: $T \leftarrow \delta(F_{v_p}, G_{i, j})$ for all $0 \leq j \leq j(i)$, via S
- 6: $Q \leftarrow \delta(F_{k'}, G_{i, j(i-1)})$ for all $1 \leq k' \leq k$, via S
- 7: $\Delta \leftarrow \delta(F_{v_p}^\circ, G_{i, j(i-1)})$ via S
- 8: end do

The correctness of COMPUTEPERIOD(v_p) follows from Lemma 5.1. However, we still need to show that all the required values are available when needed in the execution of Line 4. Let us go over the different subproblems encountered during this computation and show that each of them is available when required along the computation.

$\delta(F_{k'-1}, G_{i, j})$:

- when $k' = 1$, F_0 is $F_{v_{p-1}}$, so it is already stored in T from the previous period.

- for $k' > 1$, $\delta(F_{k'-1}, G_{i,j})$ was already computed and stored in S , since we go over values of k' in increasing order.

$\delta(F_{k'}, G_{i,j+1})$:

- when $j = j(i)$ and $i + j(i) = |G| - 1$, then $G_{i,j(i)+1} = \emptyset$ so $\delta(F_{k'}, \emptyset)$ is the cost of deleting $F_{k'}$, which may be computed in advance for all subforests within the same time and space bounds.
- when $j = j(i)$ and $i + j(i) < |G| - 1$, recall that $\delta(F_{k'}, G_{i,j(i)+1})$ is equivalent to $\delta(F_{k'}, G_{i+1,j(i)})$ so this problem was already computed, since we loop over the values of i in decreasing order. Furthermore, this problem was stored in the the array Q when line 6 was executed for the previous value of i .
- when $j < j(i)$, $\delta(F_{k'}, G_{i,j+1})$ was already computed and stored in S , since we go over values of j in decreasing order.

$\delta(L_{F_{k'}}^\circ, L_{G_{i,j}}^\circ)$:

- this value was computed previously (in step 2 of TED) as Δ_{vw} for some $v \in F - \text{HeavyPath}(F)$ and $w \in G$.

$\delta(F_{k'} - L_{F_{k'}}, G_{i,j} - L_{G_{i,j}})$:

- if $j \neq j(i)$ then $F_{k'} - L_{F_{k'}} = F_{k''}$ where $k'' = k' - |L_{F_{k'}}|$ and $G_{i,j} - L_{G_{i,j}} = G_{i,j'}$ where $j' = j + |L_{G_{i,j}}|$, so $\delta(F_{k''}, G_{i,j'})$ was already computed and stored in S earlier in the loop.
- if $j = j(i)$, then $G_{i,j}$ is a tree, so $G_{i,j} = L_{G_{i,j}}$. Hence, $\delta(F_{k'} - L_{F_{k'}}, G_{i,j} - L_{G_{i,j}})$ is simply the cost of deleting $F_{k''}$.

The space required by this algorithm is evidently $O(|F||G|)$ since the size of S is at most $|F||G|$, the size of T is at most $|G|^2$, the size of Q is at most $|F|$, and the size of Δ is $|F||G|$. The time complexity does not change, since we still handle each relevant subproblem exactly once, in constant time per relevant subproblem.

Note that in the last time COMPUTEPERIOD() is called, the table T stores (among other things) the edit distance between the two input trees. In fact, our algorithm computes the edit distance between any subtree of F and any subtree of G . We could store these values without changing the space complexity. This property will be useful in the application described in Section 5.5.

This concludes the description of our $O(mn)$ space algorithm. All that remains to show is why we may assume the input trees are binary. If they are not binary, we construct in $O(m+n)$ time binary trees F' and G' where $|F'| \leq 2n$, $|G'| \leq 2m$, and $\delta(F, G) = \delta(F', G')$ using the following procedure: Pick a node $v \in F$ with $k > 2$ children which are, in left to right order, $\text{left}(v) = v_1, v_2, \dots, v_k = \text{right}(v)$. We leave $\text{left}(v)$ as it is, and set $\text{right}(v)$ to be a new node with a special label ε whose children are v_2, v_3, \dots, v_k . To ensure this does not change the edit distance, we set the cost of deleting ε to zero, and the cost of relabeling ε to ∞ . Repeat the same procedure for G . We note that another way to remove the binary trees assumption is to modify COMPUTEPERIOD() to work directly with non-binary trees at the cost of slightly complicating it. This can be done by splitting it into two parts, where one handles left deletions and the other right deletions.

5.4 A Tight Lower Bound for Decomposition Algorithms

In this section we present a lower bound on the worst case running time of decomposition strategy algorithms. We first give a simple proof of an $\Omega(m^2n)$ lower bound. In the case where $m = \Theta(n)$, this gives a lower bound of $\Omega(n^3)$ which shows that our algorithm is worst-case optimal among all decomposition algorithms. To prove that our algorithm is worst-case optimal for any $m \leq n$, we analyze a more complicated scenario that gives a lower bound of $\Omega(m^2n(1 + \log \frac{n}{m}))$, matching the running time of our algorithm, and improving the previous best lower bound of $\Omega(nm \log n \log m)$ time [32].

In analyzing strategies we will use the notion of a *computational path*, which corresponds to a specific sequence of recursion calls. Recall that for all subforest-pairs (F', G') , the strategy S determines a direction: either right or left. The recursion can either delete from F' or from G' or match. A computational path is the sequence of operations taken according to the strategy in a

specific sequence of recursive calls. For convenience, we sometimes describe a computational path by the sequence of subproblems it induces, and sometimes by the actual sequence of operations: either “delete from the F -subforest”, “delete from the G -subforest”, or “match”.

We now turn to the $\Omega(m^2n)$ lower bound on the number of relevant subproblems for any strategy.

Lemma 5.6 *For any decomposition algorithm, there exists a pair of trees (F, G) with sizes n, m respectively, such that the number of relevant subproblems is $\Omega(m^2n)$.*

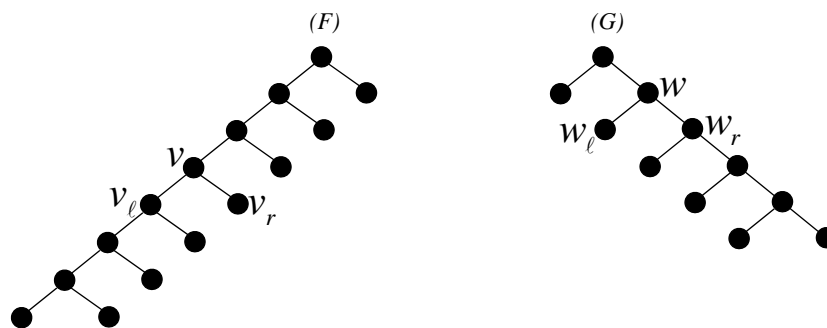


Figure 5-5: The two trees used to prove an $\Omega(m^2n)$ lower bound.

Proof. Let S be the strategy of the decomposition algorithm, and consider the trees F and G depicted in Fig. 5.4. According to lemma 5.3, every pair (F_v°, G_w°) where $v \in F$ and $w \in G$ is a relevant subproblem for S . Focus on such a subproblem where v and w are internal nodes of F and G . Denote v 's right child by v_r and w 's left child by w_ℓ . Note that F_v° is a forest whose rightmost root is the node v_r . Similarly, G_w° is a forest whose leftmost root is w_ℓ . Starting from (F_v°, G_w°) , consider the computational path $c_{v,w}$ that deletes from F whenever the strategy says left and deletes from G otherwise. In both cases, neither v_r nor w_ℓ is deleted unless one of them is the only node left in the forest. Therefore, the length of this computational path is at least $\min\{|F_v|, |G_w|\} - 1$. Recall that for each subproblem (F', G') along $c_{v,w}$, the rightmost root of F' is v_r and the leftmost root of G' is w_ℓ . It follows that for every two distinct pairs $(v_1, w_1) \neq (v_2, w_2)$ of internal nodes in F and G , the relevant subproblems occurring along the computational paths c_{v_1, w_1} and c_{v_2, w_2} are disjoint. Since there are $\frac{n}{2}$ and $\frac{m}{2}$ internal nodes in F and G respectively, the total number of

subproblems along the $c_{v,w}$ computational paths is given by:

$$\sum_{(v,w) \text{ internal nodes}} \min\{|F_v|, |G_w|\} - 1 = \sum_{i=1}^{\frac{n}{2}} \sum_{j=1}^{\frac{m}{2}} \min\{2i, 2j\} = \Omega(m^2 n)$$

■

The $\Omega(m^2 n)$ lower bound established by Lemma 5.6 is tight if $m = \Theta(n)$, since in this case our algorithm achieves an $O(n^3)$ running time. To establish a tight bound when m is not $\Theta(n)$, we use the following technique for counting relevant subproblems. We associate a subproblem consisting of subforests (F', G') with the unique pair of vertices (v, w) such that F_v, G_w are the smallest trees containing F', G' respectively. For example, for nodes v and w with at least two children, the subproblem (F_v°, G_w°) is associated with the pair (v, w) . Note that all subproblems encountered in a computational path starting from (F_v°, G_w°) until the point where either forest becomes a tree are also associated with (v, w) .

Lemma 5.7 *For every decomposition algorithm, there exists a pair of trees (F, G) with sizes $n \geq m$ such that the number of relevant subproblems is $\Omega(m^2 n \log \frac{n}{m})$.*

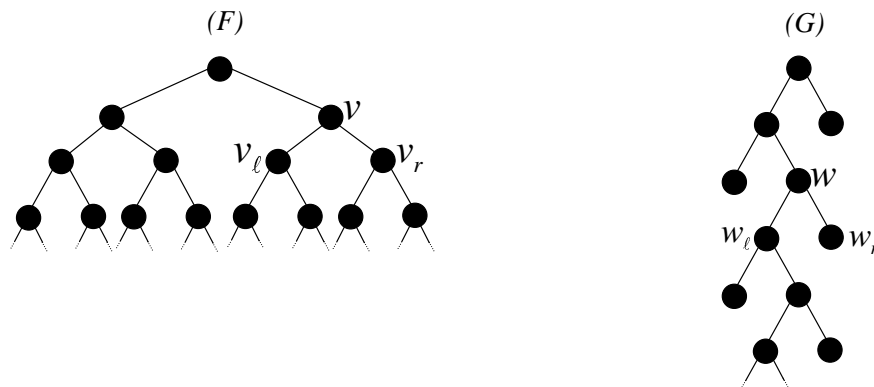


Figure 5-6: The two trees used to prove an $\Omega(m^2 n \log \frac{n}{m})$ lower bound.

Proof. Consider the trees illustrated in Fig. 5-6. The n -sized tree F is a complete balanced binary tree, and G is a “zigzag” tree of size m . Let w be an internal node of G with a single node w_r as its right subtree and w_l as a left child. Denote $m' = |G_w|$. Let v be a node in F such that F_v is a

tree of size $n' + 1$ where $n' \geq 4m \geq 4m'$. Denote v 's left and right children v_ℓ and v_r respectively. Note that $|F_{v_\ell}| = |F_{v_r}| = \frac{n'}{2}$

Let S be the strategy of the decomposition algorithm. We aim to show that the total number of relevant subproblems associated with (v, w) or with (v, w_ℓ) is at least $\frac{n'}{4}(m' - 2)$. Let c be the computational path that always deletes from F (no matter whether S says left or right). We consider two complementary cases.

CASE 1: $\frac{n'}{4}$ left deletions occur in the computational path c , and at the time of the $\frac{n'}{4}$ th left deletion, there were fewer than $\frac{n'}{4}$ right deletions.

We define a set of new computational paths $\{c_j\}_{1 \leq j \leq \frac{n'}{4}}$ where c_j deletes from F up through the j th left deletion, and thereafter deletes from F whenever S says right and from G whenever S says left. At the time the j th left deletion occurs, at least $\frac{n'}{4} \geq m' - 2$ nodes remain in F_{v_r} and all $m' - 2$ nodes are present in G_{w_ℓ} . So on the next $m' - 2$ steps along c_j , neither of the subtrees F_{v_r} and G_{w_ℓ} is totally deleted. Thus, we get $m' - 2$ distinct relevant subproblems associated with (v, w) . Notice that in each of these subproblems, the subtree F_{v_ℓ} is missing exactly j nodes. So we see that, for different values of $j \in [1, \frac{n'}{4}]$, we get disjoint sets of $m' - 2$ relevant subproblems. Summing over all j , we get $\frac{n'}{4}(m' - 2)$ distinct relevant subproblems associated with (v, w) .

CASE 2: $\frac{n'}{4}$ right deletions occur in the computational path c , and at the time of the $\frac{n'}{4}$ th right deletion, there were fewer than $\frac{n'}{4}$ left deletions.

We define a different set of computational paths $\{\gamma_j\}_{1 \leq j \leq \frac{n'}{4}}$ where γ_j deletes from F up through the j th right deletion, and thereafter deletes from F whenever S says left and from G whenever S says right (i.e., γ_j is c_j with the roles of left and right exchanged). Similarly as in case 1, for each $j \in [1, \frac{n'}{4}]$ we get $m' - 2$ distinct relevant subproblems in which F_{v_r} is missing exactly j nodes. All together, this gives $\frac{n'}{4}(m' - 2)$ distinct subproblems. Note that since we never make left deletions from G , the left child of w_ℓ is present in all of these subproblems. Hence, each subproblem is associated with either (v, w) or (v, w_ℓ) .

In either case, we get $\frac{n'}{4}(m' - 2)$ distinct relevant subproblems associated with (v, w) or (v, w_ℓ) .

To get a lower bound on the number of problems we sum over all pairs (v, w) with G_w being a tree whose right subtree is a single node, and $|F_v| \geq 4m$. There are $\frac{m}{4}$ choices for w corresponding to tree sizes $4j$ for $j \in [1, \frac{m}{4}]$. For v , we consider all nodes of F whose distance from a leaf is at least $\log(4m)$. For each such pair we count the subproblems associated with (v, w) and (v, w_ℓ) . So the total number of relevant subproblems counted in this way is

$$\sum_{v,w} \frac{|F_v|}{4} (|G_w| - 2) = \frac{1}{4} \sum_v |F_v| \sum_{j=1}^{\frac{m}{4}} (4j - 2) = \frac{1}{4} \sum_{i=\log 4m}^{\log n} \frac{n}{2^i} \cdot 2^i \sum_{j=1}^{\frac{m}{4}} (4j - 2) = \Omega(m^2 n \log \frac{n}{m})$$

■

Theorem 5.8 *For every decomposition algorithm and $n \geq m$, there exist trees F and G of sizes $\Theta(n)$ and $\Theta(m)$ such that the number of relevant subproblems is $\Omega(m^2 n (1 + \log \frac{n}{m}))$.*

Proof. If $m = \Theta(n)$ then this bound is $\Omega(m^2 n)$ as shown in Lemma 5.6. Otherwise, this bound is $\Omega(m^2 n \log \frac{n}{m})$ which was shown in Lemma 5.7. ■

5.5 Tree Edit Distance for RNA Comparison

One major application of tree edit distance is the analysis of RNA molecules in computational biology. *Ribonucleic acid* (RNA) is a polymer consisting of a sequence of nucleotides (Adenine, Cytosine, Guanine, and Uracil) connected linearly via a backbone. In addition, complementary nucleotides (AU, GC, and GU) can form hydrogen bonds, leading to a structural formation called the *secondary structure* of the RNA. Because of the nested nature of these hydrogen bonds, the secondary structure of RNA (without pseudoknots) can be naturally represented by an ordered rooted tree [42, 101] as depicted in Fig. 5.5. Recently, comparing RNA sequences has gained increasing interest thanks to numerous discoveries of biological functions associated with RNA. A major fraction of RNA's function is determined by its secondary structure [78]. Therefore, computing the similarity between the secondary structure of two RNA molecules can help determine the functional similarities of these molecules.

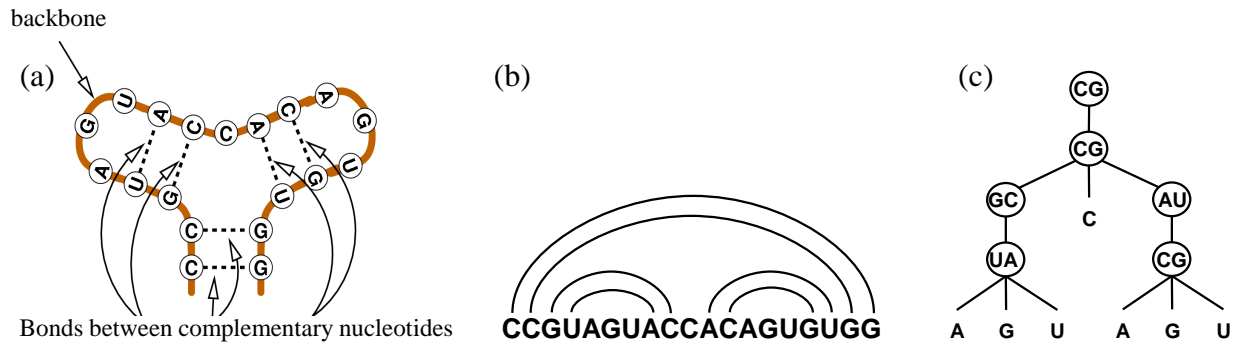


Figure 5-7: Three different ways of viewing an RNA sequence. In (a), a schematic 2-dimensional description of an RNA folding. In (b), a linear representation of the RNA. In (c), the RNA as a rooted ordered tree.

Shasha and Zhang [88] were the first to suggest representing RNA sequences as rooted ordered trees, and RNA similarity as tree edit distance. In this way, any tree editing algorithm (and therefore ours as well) can be used to compute the edit distance of two RNAs. In RNA sequences however, as in many other biological applications, the deletion of say k consecutive nucleotides is more probable to occur than k separate deletions. In this section, we show how any decomposition strategy DP can be adapted to account for this phenomena without changing the time and space bounds.

Notations. An RNA sequence \mathcal{R} is an ordered pair (S, P) , where $S = s_1 \cdots s_{|S|}$ is a string over the alphabet $\Sigma = \{A, C, G, U\}$, and $P \subseteq \{1, \dots, |S|\} \times \{1, \dots, |S|\}$ is the set of hydrogen bonds between bases of \mathcal{R} . We refer to a bond $(i', i) \in P$, $i' < i$, as an *arc*, and i' and i are referred to as the *left* and *right endpoints* of this arc. Also, we let $|\mathcal{R}|$ denote the number of nucleotides in \mathcal{R} , i.e., $|\mathcal{R}| = |S|$. Any base in \mathcal{R} can bond with at most one other base, therefore we have $\forall (i', i), (j', j) \in P, i' = j' \Leftrightarrow i = j$. Following Zuker [105, 106], we assume a model where the bonds in P are *non crossing*, i.e., for any $(i', i), (j', j) \in P$, we cannot have $i' < j' < i < j$ nor $j' < i' < j < i$. This non-crossing formation conveniently allows representing an RNA as a rooted ordered tree. Each arc (i', i) is identified with a set of ordered children which are all unpaired bases j such that $i' < j < i$, and all outermost arcs (ℓ, ℓ') with $i < \ell < \ell' < i'$ (see

Figure 5.5). We denote the two RNAs to be compared by $\mathcal{R}_1 = (S_1, P_1)$ and $\mathcal{R}_2 = (S_2, P_2)$, and we set $|\mathcal{R}_1| = |S_1| = n$ and $|\mathcal{R}_2| = |S_2| = m$. We assume without loss of generality that $m \leq n$.

5.5.1 RNA Alignment

As in the case of strings, RNA edit distance is analogous to RNA alignment. An *alignment* of \mathcal{R}_1 and \mathcal{R}_2 is another way of viewing a sequence of edit operations on these two RNAs.

Definition 5.9 (Alignment) *An alignment \mathcal{A} of \mathcal{R}_1 and \mathcal{R}_2 is an ordered subset of $\{1, \dots, n\} \cup \{-\} \times \{1, \dots, m\} \cup \{-\}$ satisfying the following conditions:*

1. $(-, -) \notin \mathcal{A}$.
2. $\forall (i, j) \in \{1, \dots, n\} \times \{1, \dots, m\} : i \text{ and } j \text{ appear exactly once in } \mathcal{A}$.
3. $\forall (i', j'), (i, j) \in \mathcal{A} \cap \{1, \dots, n\} \times \{1, \dots, m\} : i' < i \iff j' < j$. *That is, any two pairs in \mathcal{A} are non-crossing.*
4. $\forall (i, j) \in \mathcal{A} \cap \{1, \dots, n\} \times \{1, \dots, m\} : i \text{ is a left (resp. right) arc endpoint in } \mathcal{R}_1 \iff j \text{ is a left (resp. right) arc endpoint in } \mathcal{R}_2$.
5. $\forall (i', i) \in P_1, (j', j) \in P_2 : (i', j') \in \mathcal{A} \iff (i, j) \in \mathcal{A}$. *That is, the left endpoints of any pair of arcs are aligned against each other in \mathcal{A} iff their right endpoints are also aligned against each other in \mathcal{A} .*

The ordering of the pairs in \mathcal{A} is required to be a linear extension of the natural ordering between pairs of integers. That is, (i', j') is before (i, j) iff $i' < i$ or $j' < j$.

In terms of editing operations, a pair $(i, j) \in \mathcal{A} \cap \{1, \dots, n\} \times \{1, \dots, m\}$ corresponds to relabeling the i th nucleotide (unpaired or not) of \mathcal{R}_1 so it would match the j th nucleotide of \mathcal{R}_2 , while pairs $(i, -)$ and $(-, j)$ corresponds to deleting the i th and j th nucleotides in \mathcal{R}_1 and \mathcal{R}_2 respectively. The ordering between the pairs corresponds to the order of edit operations. The first three conditions in the above definition require any position in \mathcal{R}_1 and \mathcal{R}_2 to be aligned, and that $(-, -) \notin \mathcal{A}$, since $(-, -)$ does not correspond to any valid edit operation. The next condition enforces the order of the

subsequences to be preserved in \mathcal{A} , and the last two conditions restrict any arc to be either deleted or aligned against another arc in the opposite RNA. Figure 5-8 gives two example alignments for a pair of RNA sequences.

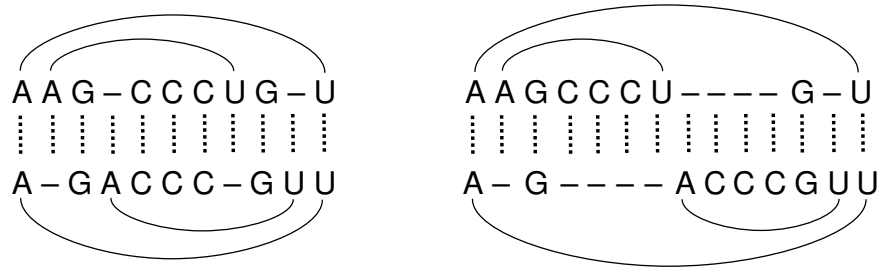


Figure 5-8: Two example alignments for a pair of RNA sequences.

5.5.2 Affine Gap Penalties

We now show how to extend any decomposition strategy algorithm to handle *affine gap penalties* in the same time and space bounds. Given an alignment \mathcal{A} of \mathcal{R}_1 and \mathcal{R}_2 , a *gap* is a consecutive sequence of pairs $(i, -) \in \mathcal{A}$ or a consecutive sequence of pairs $(-, j) \in \mathcal{A}$. For example, in the left alignment of Fig. 5-8 the gaps are $(A, -)$, $(-, A)$, $(U, -)$ and $(-, U)$; whereas in the right alignment the gaps are $(A, -)$, $(CCCU, - - - -)$, $(- - - -, ACCC)$ and $(-, U)$.

In biological context, one single long gap is more probable to occur than a multitude of separated short gaps. This situation can be modeled by using an affine gap penalty function $f(k) = g + k\alpha$, where $f(k)$ is the score of a gap of length k , g is an initiation penalty to a gap, and α is the cost of extending a gap by one. This was first introduced by Gotoh [41] in the context of string editing, and we extend the technique to ordered tree editing.

Notice that deleting an arc means deleting its two endpoints, each of which might be a part of a different gap. To account for this, we slightly modify the construction of an ordered tree from an RNA sequence. To each node in the tree that corresponds to some arc (i, j) we add a leftmost child i^* and a rightmost child j^* . Both i^* and j^* are leaves, and relabeling them to anything costs ∞ . When we delete the rightmost (respectively leftmost) root we also delete its rightmost (respectively leftmost) child, and when we match a node we remove both its leftmost and rightmost children.

In what follows, we modify the decomposition strategy recursion of Lemma 5.1 to handle sequential affine gap penalties. In order to modify the recursion, we notice that any deletion may either continue an existing gap or may start a new gap. For forests F and G , we define nine different “problems”. Denoting by ℓ_F and r_F the leftmost and rightmost roots of a forest F , these problems are:

- $[F, G]$ is the edit distance with affine gap penalties between F and G .
- ${}_F[F, G]$ (respectively: ${}_G[F, G]$, $[F, G]_F$, $[F, G]_G$) is the edit distance with affine gap penalties between F and G *subject to the condition* that ℓ_F (respectively: ℓ_G , r_F , r_G) is deleted.
- ${}_F[F, G]_F$ (respectively: ${}_F[F, G]_G$, ${}_G[F, G]_F$, ${}_G[F, G]_G$) is the edit distance with affine gap penalties between F and G *subject to the condition* that both ℓ_F and r_F (respectively: ℓ_F and r_G , ℓ_G and r_F , ℓ_G and r_G) are deleted.

The following recursion computes the values of all the nine problems. Our recursion can follow any given decomposition strategy that determines the direction of the recursion in every recursive call. We present the recursion for a recursive call in which the strategy says left, the case in which the strategy says right is symmetric (simply replace the left and right subscripts of the '[''s and ']''s). For every $X \in \{F, G, \varepsilon\}$ (where ε is used to denote the empty string):

$$\begin{aligned}
- [F, G]_X &= \min \begin{cases} {}_F[F, G]_X, \\ {}_G[F, G]_X, \\ [F_{\ell_F} - \ell_F, G_{\ell_G} - \ell_G] + [F - F_{\ell_F}, G - G_{\ell_G}]_X + \text{cost of relabeling } \ell_F \text{ to } \ell_G. \end{cases} \\
- {}_F[F, G]_X &= \min \begin{cases} {}_F[F - \ell_F, G]_X + \alpha, \\ [F - \ell_F, G]_X + g + \alpha. \end{cases} \\
- {}_G[F, G]_X &= \min \begin{cases} {}_G[F, G - \ell_G]_X + \alpha, \\ [F, G - \ell_G]_X + g + \alpha. \end{cases}
\end{aligned}$$

The halting conditions of this recursion are:

$$- [\emptyset, \emptyset] = 0$$

- ${}_F[\emptyset, \emptyset] = {}_G[\emptyset, \emptyset] = [\emptyset, \emptyset]_F = [\emptyset, \emptyset]_G = {}_F[\emptyset, \emptyset]_F = {}_G[\emptyset, \emptyset]_G = g$
- ${}_F[\emptyset, \emptyset]_G = {}_G[\emptyset, \emptyset]_F = 2g$
- $[\emptyset, G] = [\emptyset, G]_G = {}_G[\emptyset, G] = {}_G[\emptyset, G]_G = g + \alpha|G|$
- $[F, \emptyset] = [F, \emptyset]_F = {}_F[F, \emptyset] = {}_F[F, \emptyset]_F = g + \alpha|F|$
- $[F, \emptyset]_G = {}_G[F, \emptyset] = {}_G[F, \emptyset]_G = {}_F[F, \emptyset]_G = {}_G[F, \emptyset]_F = 2g + \alpha|F|$
- $[\emptyset, G]_F = {}_F[\emptyset, G] = {}_F[\emptyset, G]_F = {}_F[\emptyset, G]_G = {}_G[\emptyset, G]_F = 2g + \alpha|G|$

Time and space complexity. For every subproblem encountered by the decomposition strategy, the above recursion encounters nine subproblems. Therefore, since the number of subproblems corresponds to the time complexity, our recursion requires $O(m^2n(1 + \lg \frac{n}{m}))$ time by using our strategy from Section 5.2. Computing these subproblems can be done in $O(mn)$ space as shown in Section 5.3.

Chapter 6

Fractional Subproblems

In the chapters we have seen so far, the solution to a subproblem of a DP is some unique number that we want to compute. In some DPs however, the solution to a subproblem is an entire data structure rather than a single number. In such DPs, the entire data structure of a subproblem is processed and used to construct the data structure of larger subproblems. In this chapter, we discuss the idea of processing only parts of a subproblem's data structure.

We show that in some cases, such fractional parts of a data structure remain unchanged when constructing the data structure of the larger subproblem. The general idea is to partition the data structure of the smaller subproblem into two parts. Then, to construct the data structure of the larger subproblem, one of these parts will be read and processed while the other part will be copied as is (using a constant number of pointer changes). We show how this idea can be used for the problem of finding the optimal tree searching strategy in linear time.

This chapter is organized as follows. In Section 6.1 we describe the problem of finding an optimal tree searching strategy as a generalization of the well known binary search technique. In Section 6.2 we outline the machinery required for devising DP solutions to this problem, and in Section 6.3 we give a new $O(n^2)$ -time DP. The fractional subproblems idea is described in Section 6.4 and is used to reduce the time complexity to $O(n)$. Our solution represents the optimal strategy in the form of a weight function on the tree's edges. In the final Section 6.5 we show how to convert this representation into a decision tree representation in $O(n)$ time.

6.1 Finding an Optimal Tree Searching Strategy in Linear Time

The binary search technique is a fundamental method for finding an element in a sorted array or a totally ordered set. If we view the sorted elements as a line of vertices connected by edges, then searching for the target element is done by querying edges such that a query on edge e tells us which endpoint of e is closer to the target. It is well known that in such a search, the optimal way to minimize the number of queries in the worst case is to perform a binary search (see Knuth's book [56]). This technique repeatedly queries the middle edge of the searched segment and eliminates half of the segment from further consideration. Binary search can therefore be described by a complete decision tree where every decision node corresponds to a query on some edge and has degree two, associated with the two possible outcomes of the query.

The problem of locating an element in a sorted array naturally generalizes to the problem of locating a vertex in a tree [14, 57, 83]. Again, we are allowed to query an edge to find out which of each endpoints is closer to the required vertex. Another natural generalization is searching in partially ordered sets (posets) rather than totally ordered sets [14, 21, 65, 66]. When searching a poset for a target element x , the queries are of the form " $x \leq y$?" for some member y of the poset. A negative answer to a query means that either $x > y$ or that x and y are incomparable. These two generalizations are equivalent when the partial order can be described by a forest-like diagram.

Both search problems can be formalized as follows. *Given a tree (or a partially ordered set), construct a decision tree of the lowest possible height that enables the discovery of every target element. A decision node corresponds to a query and has degree two associated with the two possible outcomes of the query.* Unlike searching in a sorted array or a totally ordered set, the optimal decision tree is now not necessarily complete and depends on the structure of the input tree (or the partial order diagram). This is illustrated in Figure 6-1 for the case of searching a tree. A searching strategy based on this decision tree is called the *optimal strategy* and is guaranteed to minimize the number of queries in the worst case.

Carmo *et al.* [21] showed that finding an optimal strategy for searching in general posets is NP-hard, and gave an approximation algorithm for random posets. For trees and forest-like posets, however, an optimal strategy can be computed in polynomial time as was first shown by Ben-

Asher, Farchi, and Newman [14]. Ben-Asher *et al.* gave an $O(n^4 \log^3 n)$ -time algorithm that finds an optimal strategy. This was improved to $O(n^3)$ by Onak and Parys [83] who introduced a general machinery of bottom-up DP for constructing optimal strategies. Laber and Nogueira [57] gave an $O(n \log n)$ -time algorithm that produces an additive $\lg n$ -approximation. This yields a 2-multiplicative approximation, since the depth of a valid decision tree is always at least $\lg n$.

Our Results. We follow [14, 57, 83] and focus on trees and forest-like posets. That is, we are interested in computing the optimal strategy for searching a tree where querying an edge tells us which endpoint of the edge is closer to the target. We present a worst-case $O(n)$ -time algorithm for this problem, improving the previous best $O(n^3)$ -time algorithm. Our result requires a novel approach for computing subproblems in the bottom-up DP framework of [83]. In addition to proving the correctness of this approach, we introduce two new ideas that are crucial for obtaining a linear-time algorithm. The first is a method for reusing parts of already computed subproblems. This is the technique we refer to as *fractional subproblems* and is the main theme of this chapter. The second idea is a linear-time transformation from an edge-weighted tree into a decision tree. Our result improves the running time of algorithms for searching in forest-like partial orders as well, as discussed in [83].

Applications. One practical application of our problem is file system synchronization. Suppose we have two copies of a file system on two remote servers and we wish to minimize the communication between them in order to locate a directory or a file at which they differ. Such a scenario occurs when a file system or database is sent over a network, or after a temporary loss of connection. The two servers can compare directory or file checksums to test whether two directories or files differ. Such a checksum test can detect if the fault is in a subdirectory or in a parent directory. Directories are normally structured as rooted trees and a checksum test on a rooted subtree is equivalent to an edge query on an unrooted subtree. Notice that we assume edge queries on unrooted trees but this is equivalent to subtree queries on rooted trees.

Software testing or “bug detection” is another motivation for studying search problems in posets (and in particular in trees). Consider the problem of locating a buggy module in a pro-

gram where dependencies between modules constitute a tree. For each module we have a set of exhaustive tests that verify correct behavior of the module. Such tests can check, for instance, whether all branches and statements in a given module work properly. Minimizing the worst-case number of modules that we test in order to locate the buggy module reduces to our searching problem.

Related research. Many other extensions of binary search are reported in the literature. These include querying vertices rather than edges [83], Fibonacci search [36], interpolation search [84], searching when query costs are non-uniform [24, 55, 81], and searching an order ideal in a poset [65, 66].

After publishing our result in SODA 2008, we discovered that the problem of *tree ranking* is highly related to our problem of searching in trees. The term tree ranking is essentially identical to the term strategy function that we use, and results for tree ranking and tree searching were developed in parallel for more than ten years now. To our knowledge, the relation between these problems has never been stated. This is probably because the applications differ a lot. While the applications for tree searching are filesystem synchronization and bug detection, applications for tree ranking are in VLSI design and matrix factorization.

For the problem of tree ranking, Torre, Greenlaw, and Schaffer [30] gave an $O(n^3 \log n)$ -time algorithm, Iyer, Ratliff, and Vijayan [47] gave a 2-multiplicative approximation algorithm running in $O(n \log n)$ time, and finally, Lam and Yue [58] presented an optimal $O(n)$ solution. While the high level descriptions of the Lam-Yue algorithm and of ours are similar, the two solutions differ in the representation of visibility lists and choice of data structures (i.e. in implementing the fractional subproblems technique). In addition, unlike Lam and Yue’s algorithm, our solution achieves $O(n)$ running time without using word-level parallelism (“bit-tricks”). Finally, we show how to transform in linear time a ranking/strategy into the corresponding decision tree. This is an important component that is unique for the tree searching problem and was not required for the tree ranking problem.

6.2 Machinery for Solving Tree Searching Problems

In this section we review the techniques required for a bottom-up construction of the optimal strategy as introduced by Onak and Parys [83].

Strategy functions. Recall that given a tree $T = (V, E)$, our goal is to find an *optimal strategy* for searching in T . This strategy should minimize the worst-case number of queries required to locate a target vertex, or equivalently, correspond to a correct decision tree of the lowest possible height. Onak and Parys showed that finding an optimal strategy is equivalent to finding an *optimal strategy function*. A *strategy function* $f: E \rightarrow \mathbb{Z}_+$ is a function from the set of edges into the set of positive integers that satisfies the following condition. If f takes the same value on two different edges e_1 and e_2 , then on the simple path from e_1 to e_2 , there is an edge e_3 on which the function takes a greater value (i.e. $f(e_3) > f(e_1) = f(e_2)$). An *optimal strategy function* is one with the lowest possible maximum. We make use of the following lemma.

Lemma 6.1 ([83]) *For every tree T , the worst-case number of queries in an optimal searching strategy in T equals the lowest possible maximum of a strategy function on T .*

The intuition behind strategy functions is that if $f(e) = k$, then when we query the edge e we have at most k more queries until we find the target vertex. It turns out that a strategy function f with maximum k easily transforms into a searching strategy with at most k queries in the worst-case. The first query in the strategy being constructed is about the edge with the maximal value k . If we remove this edge, the tree breaks into two subtrees and the problem reduces to finding the target vertex in one of the subtrees, say T' . The second query is about the edge with maximal value in T' and we continue recursively. By definition of the strategy function f , in the i th query there is a unique edge with maximal value. An example of a strategy function and the corresponding search strategy is illustrated in Figure 6-1. In Section 6.5 we present an $O(n)$ -time algorithm that transforms a strategy function into a decision tree.

Bottom-up DP. Our objective is thus to find an optimal strategy function f . To do so, we arbitrarily root the tree T , and compute $f(e)$ for every $e \in E$ using DP in a bottom-up fashion.

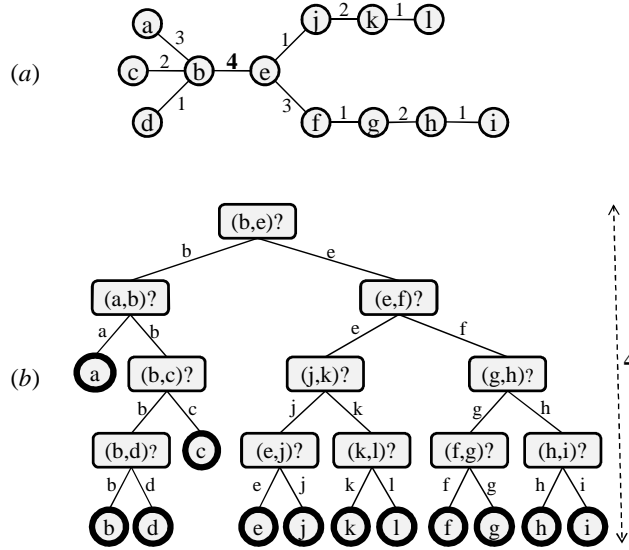


Figure 6-1: (a) a sample input tree with its optimal strategy function. (b) the corresponding optimal decision tree. The height of the decision tree is equal to the maximum value assigned by the strategy function

More formally, suppose we have a node u with children u_1, \dots, u_k connected to u by the edges e_1, \dots, e_k . Assuming that f has already been computed for all $T(u_1), \dots, T(u_k)$ (where $T(u)$ is the subtree rooted at u), we extend the function to $T(u)$ by computing $f(e_1), \dots, f(e_k)$ without changing $f(e)$ for any $e \notin \{e_1, \dots, e_k\}$. This means that the restriction to $T(u_i)$ of our desired optimal strategy function for searching $T(u)$ is optimal for searching $T(u_i)$ for every $1 \leq i \leq k$.

To describe this extension we need the notion of *visibility*. We say that an edge e is *visible* from a vertex u , if on the simple path from u ending with the edge e there is no edge e' such that $f(e') > f(e)$. In other words, the visible values from u are those which are not “screened” by greater values of f . Note that by the definition of a strategy function, each value of f is visible from u at most once. The enumeration in descending order of all values visible from u in $T(u)$ is called the *visibility sequence* of u .

Note that in order to extend the strategy functions on $T(u_1), T(u_2), \dots, T(u_k)$ to a correct strategy function on the entire $T(u)$, it suffices to know just the visibility sequence of each u_i . Denote by s_i the visibility sequence at u_i in $T(u_i)$. We want to assign values to the edges e_1 to e_k so that we achieve a correct strategy function on $T(u)$. We need to make sure that:

- For every two edges $e_i \neq e_j$, $f(e_i) \neq f(e_j)$.
- For every edge e_i , $f(e_i)$ is not present in s_i .
- For every edge e_i , if $f(e_i)$ is present in s_j , then $f(e_j) > f(e_i)$.
- If the same value v appears in two visibility sequences s_i and s_j , where $i \neq j$, then the maximum of $f(e_i)$ and $f(e_j)$ is greater than v .

One can easily verify that these conditions suffice to obtain a valid extension of the strategy functions on $T(u_1), \dots, T(u_k)$ to a strategy function on $T(u)$.

Consider a lexicographical order on visibility sequences. A valid extension that yields the smallest visibility sequence at u is called a *minimizing extension*. An extension is called *monotone* if increasing the visibility sequences at the children does not decrease the visibility sequence which the extension computes for their parent. Onak and Parys proved that extensions that are both minimizing and monotone accumulate to an optimal strategy function. They further showed that, for the tree searching problem being considered, every minimizing extension is also monotone.

Lemma 6.2 ([83]) *The bottom up approach yields an optimal strategy if at every node we compute a minimizing extension.*

6.3 Computing a Minimizing Extension

In this section we describe a novel algorithm for computing a minimizing extension. An efficient implementation of this algorithm is presented in Section 6.4. This implementation uses the fractional subproblems technique and yields an $O(n)$ -time algorithm for computing an optimal strategy function.

6.3.1 Algorithm Description.

We first describe the intuition behind the algorithm, and introduce helpful notions. Along the explanation we refer to the relevant line numbers in the pseudocode of the algorithm, which is

```

1: let all  $g_i = 0$ 
2: let all entries of the array  $U$  contain the value free
3: add the value 0 to every  $s_i$ 
4:  $v \leftarrow$  largest value in all  $s_i$ 
5: while not all edges assigned a positive value:
6:     if  $v$  is exposed at least twice or  $v = 0$ :
7:          $w \leftarrow$  smallest  $i$  s.t.  $i > v$  and  $U[i]$  is free
8:          $T \leftarrow \{i \in \{1, \dots, k\} : s_i \text{ contains an exposed value smaller than } w\}$ 
9:          $j \leftarrow$  any  $i \in T$  such that  $s_i \geq_w s_{i'}$  for every  $i' \in T$ 
10:         $U[w] \leftarrow$  taken
11:        for all values  $w'$  in  $s_j$  such that  $w'$  is exposed and  $v < w' < w$ :
12:             $U[w'] \leftarrow$  free
13:         $g_j \leftarrow w$ 
14:    else
15:         $U[v] \leftarrow$  taken
16:         $v \leftarrow$  largest exposed value in  $S$  smaller than  $v$ 

```

Figure 6-2: Algorithm for computing a minimizing extension

given in Figure 6-2.

Consider a vertex u with k children u_1, u_2, \dots, u_k , connected to u along edges e_1, e_2, \dots, e_k respectively. Let $S = \{s_1, s_2, \dots, s_k\}$ be the set of already computed visibility sequences at the children. To understand the intuition leading to the formulation of the algorithm, Consider the largest value that appears in more than one of the visibility sequences. Denote this value v (Line 6). In a sense, v is the most problematic value, since any valid assignment of values to the edges must assign some value $w > v$, to one of the edges corresponding to the visibility sequences in which v appears.

What would be a good value for w ? We say that a positive value is *free* if it is not visible from u . The set of free values changes as we modify the values assigned to e_1, e_2, \dots, e_k during the execution of the algorithm. Obviously, choosing a value which is not free for w will not result in a valid visibility sequence. Our algorithm therefore chooses the smallest free value greater than v as w (Line 7). But to which edge should w be assigned?

If we assign w to an edge e_i , all values in s_i that are smaller than w become “hidden” from all the edges not in $T(u_i)$. In other words, these values, which were not free until w was assigned (they

appeared in s_i), may now become free, and will not contribute to the resulting visibility sequence at u . This means that we should assign w to such an edge so that the greatest possible values will be freed. In other words, we should assign w to an edge whose visibility sequence contains the greatest elements smaller than w (Lines 8–9,13). We call such an edge *maximal with respect to w* , a notion that is formalized in Definition 6.4. It turns out that it is worthwhile to assign w to the maximal edge with respect to w regardless of whether this edge contains an occurrence of the multiple value v we originally wanted to take care of.

Once w is assigned to e_i , we only care about values in s_i that are greater than w . We refer to these values as *exposed* values. The values in s_i that are smaller than w no longer affect the visibility sequence in u (Lines 11-12). We then repeat the same process for the largest value currently exposed more than once in S (Lines 5,16). Note that this value may still be v , or some other value smaller than v . The process is repeated until no value is exposed multiple times. Note that during this process we may decide to assign a greater value to an edge e_i that was previously assigned a smaller value. However, as we will see, this procedure never decreases the values we assign to the edges e_i . It is important to emphasize that the only values assigned by the extension algorithm are to the edges e_i , connecting u to its children u_i . We never change values of edges in the subtrees $T(u_i)$. Once all values are exposed at most once, we have to assign values to any edges that were not yet assigned. This is done by assigning the smallest free value according to the same considerations described above. In Section 6.3.1 we prove that the values assigned to the edges at the end of this process constitute a valid minimizing extension.

We now formally state the necessary definitions.

Definition 6.3 ($s_i \geq s_j$) *The \geq relation denotes lexicographic order on visibility sequences. We define $s_i > s_j$ analogously.*

Definition 6.4 ($s_i \geq_v s_j$) *We say that s_i is larger than s_j with respect to v , denoted $s_i \geq_v s_j$, if after removing all values greater than v from both visibility sequences, s_i is not lexicographically smaller than s_j . We define the relations $>_v$ and $=_v$ analogously.*

For example, if $s_1 = \{5, 1\}$ and $s_2 = \{3\}$ then $s_1 >_6 s_2$, $s_1 >_2 s_2$, but $s_2 >_4 s_1$.

Throughout the execution of the algorithm, we maintain values g_i , which keep track of the values assigned to the edges. Eventually, at the end of the execution, these values describe our extension. Below we define two different kinds of values.

Definition 6.5 (Exposed values) *Let v be a value in a visibility sequence s_i . During execution, we say that v is exposed in s_i if v is greater than or equal to the current value of g_i . We define $\text{exposed}(s_i)$ to be the set of exposed values in s_i .*

Definition 6.6 (Free values) *A positive value v is free at a given time during execution if at that time it is neither exposed in any visibility sequence s_i , nor equals any g_i .*

We keep track of the free values using the array U . Recall that the algorithm in the form of pseudocode is given in Figure 6-2.

Proof of Correctness. Let us start with a simple observation about the algorithm.

Observation 6.7 *The algorithm has the properties:*

1. *The values g_j never decrease.*
2. *If $g_j > 0$ then g_j is greater than v 's current value.*
3. *If $g_j > 0$ then g_j is not exposed in any of the input sequences.*
4. *The current v in the algorithm is always greater than or equal to the current largest value exposed at least twice.*
5. *If $g_j > 0$ then there are no values greater than g_j which are exposed more than once.*

To be able to describe the state of the algorithm at intermediate stages, we introduce a slightly modified problem. This problem captures the fact that some edges have already been assigned a value, and that our algorithm will never decrease this value. We will discuss a natural one-to-one correspondence between instances of the modified problem and of the original one.

Definition 6.8 (Modified problem)

Given a set of k sequences $\bar{S} = \{\bar{s}_1, \dots, \bar{s}_k\}$, where each sequence $\bar{s}_i \subseteq \{0, 1, \dots, n\}$, find a sequence of non-negative values $F = f_1, \dots, f_k$ such that:

1. (no duplicates within sequence) $\forall i : f_i \notin \bar{s}_i$
2. (no duplicates between sequences) $\forall 1 \leq v \leq n : |\{i : v = f_i\} \cup \{i : v \in \bar{s}_i \text{ and } v > f_i\}| \leq 1$
3. (no decreasing assignments) $\forall i : f_i > 0 \Rightarrow f_i > \min(\bar{s}_i)$

The modified problem can be used to describe intermediate situations, where some of the edges e_i were already assigned a value.

Definition 6.9 ($\bar{S}(S, G)$) *Let S be the original set of visibility sequences at the children and $G = g_1, \dots, g_k$ be the values assigned to the edges so far ($g_i = 0$ if e_i was not assigned any value). We define the modified problem $\bar{S}(S, G)$ associated with S and G as the set of modified input sequences $\bar{s}_i(s_i, g_i) \stackrel{\text{def}}{=} \{g_i\} \cup \text{exposed}(s_i)$.*

Obviously, g_i is the smallest element in $\bar{s}_i(s_i, g_i)$. If $g_i = 0$, then e_i must be assigned a positive value by f_i (the first condition in definition 6.8). If $g_i > 0$, then we do not have to assign a value to e_i , so f_i may be zero. We may, however, increase the assigned value by choosing $f_i > g_i$ if we wish to (the last condition in definition 6.8).

Definition 6.10 ($Q(S, F)$) *Let $Q(S, F)$ be the set of visible values at the root for a valid solution F of the original problem S .*

Definition 6.11 ($\bar{Q}(\bar{S}, F)$) *Let $\bar{Q}(\bar{S}, F) \stackrel{\text{def}}{=} \cup_i \bar{q}_i(\bar{s}_i, f_i)$ be the set of visible values at the root u for a valid solution F of the modified problem \bar{S} . $\bar{q}_i(\bar{s}_i, f_i) \stackrel{\text{def}}{=} \{f_i \text{ if } f_i > 0\} \cup \{v : v \in \bar{s}_i \text{ and } v > f_i\}$.*

In other words, $\bar{q}_i(\bar{s}_i, f_i)$ is the set consisting of $\max(g_i, f_i)$ and all values in \bar{s}_i greater than $\max(g_i, f_i)$. Note that the validity of the solution F assures that the \bar{q}_i 's are disjoint sets.

The correspondence between intermediate situations of the original problem and inputs to the modified problem leads to a correspondence between valid assignments of the two. The formal proof of this observation is omitted for brevity.

Observation 6.12 Let $G = g_1, \dots, g_k$ be an intermediate assignment in the execution of the algorithm for input visibility sequences $S = s_1, \dots, s_k$. Let $\bar{S} = \bar{S}(S, G)$. The minimal $Q(S, F)$, where F ranges over valid solutions to S and such that for each i , $f_i \geq g_i$, equals the minimal $\bar{Q}(\bar{S}, F')$, where F' ranges over valid solutions of \bar{S} .

We now proceed with the proof of correctness. Consider the running of our algorithm and let w be as in Line 7 of our algorithm. That is, if v is the maximal element which is exposed at least twice in S , then w is the smallest free element greater than v . Also, let G denote the current values assigned to the edges (G is the set of values assigned when Line 13 was executed in the previous loops).

Lemma 6.13 In any valid assignment F for $\bar{S}(S, G)$, there is a value $z \in F$ such that $z \geq w$.

Proof. Let F be a valid assignment for \bar{S} . Assume, contrary to fact, that $w' < w$ is the largest value in F . If $w' \leq v$, then v appears twice in $\bar{Q}(\bar{S}, F)$ in contradiction to the validity of F (the second condition in Definition 6.8 is violated). Otherwise, $w' > v$. The fact that w is the smallest free element greater than v implies that $w' \in \bar{s}_i$ for some i . If $f_i = w'$, then the first condition in definition 6.8 is violated. Otherwise, $f_i < w'$, so the second condition in Definition 6.8 is violated. ■

We next prove the main technical lemma which essentially shows that the way our algorithm assigns values to edges does not eliminate all optimal solutions.

Lemma 6.14 Let $G = g_1, \dots, g_k$ be an intermediate assignment in execution of the algorithm for input visibility sequences $S = s_1, \dots, s_k$. Furthermore, let s_j be as in Line 9 of the algorithm. That is, s_j contains an exposed value smaller than w , and $s_j \geq_w s_i$, for any s_i that has an exposed value smaller than w .

There is an optimal solution F to the problem $\bar{S} \stackrel{\text{def}}{=} \bar{S}(S, G)$ such that $f_j \geq w$.

Proof. Let F' be an optimal solution to $\bar{S}(S, G)$. If $f'_j \geq w$, we are done. Otherwise, $f'_j < w$.

Case 1: If $w \notin F'$, then consider the smallest w' in F' such that $w' > w$. Such w' must exist by Lemma 6.13. Let i be such that $f'_i = w'$. We know the following:

1. $g_i < f'_i = w'$ (by Definition 6.8, third condition).
2. F' does not contain any values in the range $\{w, w + 1, \dots, w' - 1\}$ (by definition of w').
3. No value greater than or equal to w appears in \bar{S} more than once (by definition of w).
4. The value w does not appear in \bar{S} and F' .

Consider the assignment F with $f_i \stackrel{\text{def}}{=} \{w \text{ if } w > g_i, 0 \text{ otherwise}\}$ and $f_\ell \stackrel{\text{def}}{=} f'_\ell$ for all other values of ℓ . By the above properties, F is a valid assignment with $\bar{Q}(\bar{S}, F) < \bar{Q}(\bar{S}, F')$ in contradiction to the optimality of F' .

Case 2: If $w \in F'$, then let i be such that $f'_i = w$.

Case 2.1: If \bar{s}_i does not contain any values smaller than w , then g_i must be greater than w . We have $g_i > w = f'_i > 0$, in contradiction to the validity of F' (the third condition in Definition 6.8 is violated).

Case 2.2: If \bar{s}_i contains just a single value smaller than w , then this value must be g_i .

- If $g_i = 0$, then we may exchange the values assigned to e_i and e_j . The desired assignment F is therefore: $f_j \stackrel{\text{def}}{=} f'_i = w$, $f_i \stackrel{\text{def}}{=} \max\{f'_j, g_j\}$, and $f_\ell \stackrel{\text{def}}{=} f'_\ell$ for all other values of ℓ .
- If $g_i > 0$, then there was no need to increase the value assigned to e_i from g_i to w . In particular, g_i must equal f'_m , for some m . Otherwise, by setting f'_i to 0 in F' , we would get a valid solution better than F' . To be able to assign g_i to e_i , we must assign a different value to e_m . The assignment F'' with $f''_m \stackrel{\text{def}}{=} f'_i = w$, $f''_i \stackrel{\text{def}}{=} 0$, and $f''_\ell \stackrel{\text{def}}{=} f'_\ell$, for all other values of ℓ , is a valid assignment with $\bar{Q}(\bar{S}, F'') = \bar{Q}(\bar{S}, F')$. We may repeat the entire proof with F'' in the place of F' . The fact that $g_m < f'_m = g_i$ assures us that we will not repeat entering this case indefinitely.

Case 2.3: If \bar{s}_i contains more than one element smaller than w , then $\text{exposed}(s_i)$ is not empty, so $i \in T$ in Line 8 of the algorithm.

- If $\text{exposed}(s_j) =_w \text{exposed}(s_i)$, then by property 5 in Observation 6.7, $g_i = g_j = 0$. We may therefore exchange f'_j and f'_i , and we are done.

- Otherwise, $exposed(s_j) >_w exposed(s_i)$. To see that, consider m_i and m_j , the largest exposed values smaller than w in s_i and s_j respectively. Since $s_j \geq_w s_i$, we get that $m_j \geq m_i$. If $m_j = m_i$, then by property 5 in Observation 6.7, $g_i = g_j = 0$, so $s_i = exposed(s_i) \neq_w exposed(s_j) = s_j$. Therefore, $exposed(s_j) >_w exposed(s_i)$. Let x be the largest value smaller than w that is exposed in s_j but not in s_i . Consider the assignment F with $f_j \stackrel{\text{def}}{=} f'_j = w$, $f_i \stackrel{\text{def}}{=} \max\{f'_j, x\}$ and $f_\ell \stackrel{\text{def}}{=} f'_\ell$ for all other values of ℓ . $\bar{Q}(\bar{S}, F)$ is not larger than $\bar{Q}(\bar{S}, F')$, so F is an optimal solution with $f_j = w$. F is valid because $\max\{f'_j, x\}$ is not exposed in s_i . x is not exposed in s_i by definition, and if $f'_j > x$ then f'_j cannot be exposed in s_j since $exposed(s_j) >_w exposed(s_i)$. ■

Theorem 6.15 *Our algorithm finds an optimal assignment in finite time.*

Proof. We will show that there is an optimal assignment $F = f_1, \dots, f_k$, such that throughout the execution of our algorithm, $\forall i : g_i \leq f_i$, where g_i are the values assigned to the edges in Line 13 of our algorithm.

We proceed by induction on t , the number of times Line 13 has been executed. For $t = 0$, $g_i = 0$ for all i , so the claim trivially holds. Assume that the claim is true for $t-1$ and let $G = \{g_1, \dots, g_k\}$ be the values assigned to the edges just before the t -th time Line 13 was executed. On the t -th time we execute Line 13, g_j will be increased by setting it to w , where w, j are as in the conditions of Lemma 6.13 and Lemma 6.14. Applying Lemma 6.14 with G shows that there exists an optimal solution F which assigns $f_j \geq w$, as required to prove the inductive step.

Since the algorithm keeps increasing the assigned values g_i , and since the sum of g_i 's in an optimal solution is bounded, the algorithm will eventually terminate. ■

6.4 Linear Time Solution via Fractional Subproblems

In this section we show that the algorithm for computing a minimizing extension can be efficiently implemented, so that the total time spent on the bottom-up computation of an optimal strategy function is linear in the number of nodes. The proof is composed of two parts. First we bound the

amount of time required for the computation of a single extension. Next, we use this to bound the total time required to find the optimal strategy function.

One might be tempted to think that the sum of lengths of all visibility sequences at the children of a node v is a lower bound on the time required to compute a minimizing extension and the resulting visibility sequence at v . This, in fact, is not true. An important observation is that in many cases, the largest values of the largest visibility sequence at the children of v appear unchanged as the largest values in the visibility sequence at v itself. By using linked-lists we reuse this part of the visibility sequence when computing an extension without ever reading or modifying it. This idea is what we refer to as the *fractional subproblems* idea.

To state this idea accurately, we define the quantities $k(v)$, $q(v)$ and $t(v)$ at each node v in the rooted tree as follows.

- $k(v)$ is the number of children of v .
- Let S denote the set of visibility sequences at the children of v , and let s_1 be the largest sequence in S . We define $q(v)$ as the sum of the largest values in each sequence over all input sequences in S except s_1 . If a sequence is empty, then we say that its largest value is 0. If S is empty, $q(v) = 0$.
- Let s be the visibility sequence at v . We define $t(v)$ to be the largest value that appears in s but does not appear in s_1 . If S is empty, $t(v) = 0$.

Lemma 6.16 bounds the time required to compute a minimizing extension and the resulting visibility sequence. The proof constructively describes how to implement each line of the algorithm. The important points in the proof are that we never read or modify any values greater than $t(v)$ in s_1 , and that by using the appropriate data structures, we are able to find the desired sequence in Lines 8–9 of the algorithm efficiently.

Lemma 6.16 *A minimizing extension and its resulting visibility sequence can be computed in $O(k(v) + q(v) + t(v))$ time for each node v .*

Proof.

We keep visibility sequences as doubly-linked lists starting from the smallest value to the largest. This allows us to reuse the largest values of the largest input sequence.

We assume for simplicity that there are at least two input sequences. If there is no input sequence, then v is a leaf, and the corresponding visibility sequence is empty, so we can compute it in constant time. If there is just one input sequence, then we should assign the smallest positive value which does not appear in the input sequence to the edge leading to the only child of v . We can find this value in $O(t(v))$ time by going over the list representing the input sequence. We then create a new visibility sequence that starts with this value. The rest of the list is the same as the portion of input visibility sequence above $t(v)$.

Assume, without loss of generality, that s_1, s_2, \dots are ordered in descending order of the largest value in each sequence. Let l_1, l_2 be the largest values in s_1, s_2 respectively. We say that the largest value of an empty input sequence is zero. Note that we can compute l_2 in $O(k(v) + q(v))$ time by simultaneously traversing all the lists representing the input sequences until all but one are exhausted.

We modify the algorithm so that instead of starting from $v = l_1$ in Line 4, we start from v equal l_2 . Clearly, there are no values greater than l_2 that appear in more than one sequence, so the only difference this modification introduces is that all the values between l_2 and l_1 that belong to s_1 would have been marked by the original algorithm as taken in the vector U (Line 15), but are not marked so after this modification. We will take care of such values when we discuss the data structure that represents the vector U below.

Representing the vector U . The vector U is used in the description of our algorithm to keep track of free and taken values. It is modified or accessed in Lines 2, 7, 12 and 15. We maintain the vector U using a stack. Values on the stack are free values. The stack always keeps the following invariant: values on the stack are ordered from the largest at the bottom to the smallest at the top. We do not mark all elements as free as in Line 2. Instead, the stack is initially empty, and if, when we decrease v in Line 16, we encounter a value that is not exposed at all, we insert this value into the stack. When implementing the loop in Lines 11–12, we insert to the stack all values greater than v that are currently exposed, and are about to become free. Since all of them are smaller

than the current w , and as we will see below, all values on the stack are greater than the current w , we may insert them in decreasing order to the stack and maintain the stack invariant. We now describe how to find the value w in Line 7. If the stack is not empty, we pop the value from the top of the stack and use it as w since this is the smallest free value greater than v . If the stack is empty, then we must find a free value greater than l_2 (remember that such values were not inserted into the stack because we changed Line 4 to start from $v = l_2$). In this case, we traverse the list representing s_1 to find the smallest value that does not appear in it. In total, we will not spend more than $O(t(v))$ time on traversing s_1 in order to find such values.

The next two data structures we describe are used to efficiently find the correct s_j in Lines 8–9.

Lists of sorted sequences. For each i between 0 and l_2 , we maintain a list L_i of the visibility sequences sorted in descending order according to \leq_i . Moreover, the following invariant always holds: a sequence s_j appears in L_i if the greatest value in s_j is at least i , and there is an exposed value in s_j that is not greater than i .

Initially, we create the lists as follows. We use a modification of radix sort. L_0 is simply the list of all sequences in an arbitrary order. To create L_i , we take L_{i-1} and create two lists, L_i^+ and L_i^- . L_i^+ contains all the sequences in L_{i-1} that contain i , and L_i^- contains all the sequences in L_{i-1} that do not contain i , but do contain a value greater than i . In both new lists, sequences occur in the same order as in L_{i-1} . L_i is a concatenation of L_i^+ and L_i^- . By induction, L_i contains sequences in descending order according to \leq_i . The total length and total setup time of all the lists L_i is $O(k(v) + q(v))$. Within the same time constraints we keep pointers from each sequence to its occurrences in the lists L_i , which will be used to remove them from the lists in constant time per removal. To maintain the invariant, we have to update the lists if at some point g_j is increased in Line 13. When this happens, we remove s_j from all L_i such that i is smaller than the smallest exposed value in s_j . If the new value of g_j is greater than the largest value in s_j , then we remove s_j from all the lists. Since initially, the total size of the lists L_i is $O(k(v) + q(v))$, we do not spend more than $O(k(v) + q(v))$ time on modifying the lists along the entire computation of the extension.

Counters of long sequences with exposed values. We introduce one more structure. For each i from 0 to l_2 , let C_i keep track of the number of sequences whose maximal value is exposed and not smaller than i . We need to update this structure only if we set g_j in Line 13 with a value greater than the maximal exposed value, m , in s_j . If this happens, we subtract 1 from all C_0 to C_m (certainly, if $m > l_2$, we stop at C_{l_2}). Since this happens at most once for each visibility sequence, and by the definition of $k(v)$ and $q(v)$, we spend at most $O(k(v) + q(v))$ time on all such subtractions.

Finding the correct s_j in Line 9. We can finally describe how to find the correct sequence s_j in Lines 8-9. Suppose first that w is at most l_2 .

- If L_w is empty, then all the sequences corresponding to indices in T have their largest exposed values smaller than w . To find the correct s_j , we start from C_0 and look for i such that $C_{i+1} = C_w$, but $C_i > C_w$. The first sequence in L_i is the required sequence. This is true since there are no sequences with exposed values between i and w , and all sequences with an exposed value of i are in L_i , and the first sequence in L_i is the largest sequence with an exposed value i with respect to i and therefore also with respect to w . Each sequence can be found this way at most once because once it is found and assigned the value w , it no longer has any exposed values. Therefore, over the entire computation of the extension we pay for this case at most $O(k(v) + q(v))$.
- Suppose now that L_w is not empty. We look at the first, and largest with respect to w , sequence s_* in L_w . It may be the case that there is a greater sequence with respect to w among the ones that have an exposed value smaller than w , but that the maximal value of this sequence is also smaller than w , and therefore, this sequence does not appear in L_w . Let m be the maximal value in s_* that is smaller than w . If $C_m = C_w$, then we did not miss any sequence, and s_* is indeed the required sequence. Note that we found it in $O(1)$ time. If $C_m > C_w$, but $C_{m+1} = C_w$, then m is the largest exposed value smaller than w , and the first sequence in L_m is the required sequence (again we have found the right sequence in constant time). If both C_m and C_{m+1} are greater than C_w , then there exists a sequence that has an exposed value greater than m and smaller than w that is not present in L_w . We find

the largest exposed value i smaller than w as in the case of an empty L_w . Note that this value is at the same time the largest value of the first sequence in L_i . As was the case for an empty L_w , the desired sequence is the first one in L_i , and the total time spent on this case is at most $O(k(v) + q(v))$.

Now assume that w is greater than l_2 . If s_1 , the largest sequence, contains an exposed value smaller than w , then s_1 is the right sequence. We may keep track of the smallest exposed value in s_1 greater than l_2 in no more than $O(t(v))$ time. Then, we can check if this is the case in constant time. If s_1 has no exposed values between l_2 and w , we proceed as in the case of $w \leq l_2$, since it suffices to find the maximal sequence with respect to l_2 .

Conclusion. We have already presented fast implementation of most of the steps of our algorithm. The remaining Lines 4, 6, and 16 can be efficiently implemented as follows. First, as we have already mentioned, we initialize v in Line 4 to l_2 instead of l_1 . When we look for the next value of v , we simultaneously traverse the lists representing all sequences whose maximal element is at least v , and as v decreases we include into our working set new sequences that become relevant since v is their largest value. For each v we consider, we can check in constant time if there are such new relevant sequences, if we sort the sequences according to their maximum values less than l_2 in $O(k(v) + q(v))$ time at the beginning of the algorithm. The total time spent on decreasing v is at most $O(k(v) + q(v))$. When we find a new v , we count the number of times it is exposed, and we update this counter as some of them are removed. This way we implement the conditional statement in Line 6 efficiently. Finally, the total number of times we increment the value of g_i at all edges is at most $k(v) + q(v) + t(v)$, because each time at least one value in the corresponding sequence becomes unexposed. After we have computed the minimizing extension we create the new visibility sequence at v from the assigned values g_i and the exposed values we encounter while simultaneously traversing all visibility sequences until we reach the smallest value in s_1 , the largest visibility sequence, that is greater than l_2 and than any g_i . This takes $O(k(v) + q(v) + t(v))$ time. To this newly created linked list we link the remaining unscanned portion of s_1 in constant time. We have thus shown the total amount of time required to compute the extension and the resulting

visibility sequence is thus $O(k(v) + q(v) + t(v))$. ■

Theorem 6.17 *An optimal strategy function can be computed in linear time.*

Proof. Recall that we arbitrarily root the input tree. For every node v , given the visibility functions and sequences on the subtrees rooted at the children of v , we compute a minimizing extension of them to a function and a visibility sequence at the subtree rooted at v . By Lemma 6.16, this takes $O(k(v) + q(v) + t(v))$ time at each node v . Eventually, according to Lemma 6.2, we get an optimal strategy function. The total computation thus takes

$$O\left(\sum_v k(v) + q(v) + t(v)\right).$$

Obviously, the total number of sequences that are used in the computation of different sequences is $n - 1$, that is, $\sum k(v) = n - 1$.

We next bound the sums of $q(v)$ and $t(v)$. We first show that $\sum t(v) \leq 2(n - 1) + \sum q(v)$, which implies that it suffices to bound $q(v)$. Recall that $t(v)$ is the largest new value that appears in the largest visibility sequence. Let $l_2(v)$ be the maximal value in the second largest sequence, or 0 if there is no such sequence or if it is empty. Obviously, $l_2(v) \leq q(v)$ for every node v . What is the sum of all $t(v) - l_2(v)$? If $t(v) - l_2(v) > 0$, then at some point in our algorithm we assign $t(v)$ to one of the edges. This means that all the values $l_2(v) + 1$ to $t(v) - 1$ are taken at that moment. Each of them is taken either by a value assigned to one of the new edges or by an exposed value in the largest input sequence. If there are indeed exposed values between $l_2(v) + 1$ and $t(v)$ in the largest input sequence, then our algorithm assigns $t(v)$ to the edge corresponding to the largest sequence, so all of these exposed values will not be visible in subsequent computations at the ancestors of v . It follows that each edge contributes to the sum of all $t(v) - l_2(v)$ at most once in each case. Hence,

$$\sum_v t(v) = \sum_v t(v) - l_2(v) + \sum_v l_2(v) \leq 2(n - 1) + \sum_v q(v).$$

Now, it suffices to bound the sum of all $q(v)$. We show by induction that the sum of all $q(v)$ in the subtree $T(v)$ rooted at v is at most $n(v) - r(v) - 1$, where $n(v)$ is the number of nodes in

$T(v)$, and $r(v)$ is the maximum value in the visibility sequence computed for v . If v is a leaf, then $q(v) = 0$, $n(v) = 1$, and $r(v) = 0$, that is, our claim holds. Suppose that v has $k = k(v)$ children, and that the claim holds for each of them. Let u_1, u_2, \dots, u_k be the children of v . Assume, without loss of generality, that u_1 has the largest visibility sequence among the children of v . This implies that $r(v) \leq r(u_1) + k$, since we can create a valid extension by assigning the values $r(u_1) + 1$ to $r(u_1) + k$ to the edges between v and its children. Then,

$$\begin{aligned} \sum_{v' \in T(v)} q(v') &= \sum_{i=2}^k r(u_i) + \sum_{i=1}^k \sum_{u' \in T(u_i)} q(u') \\ &\leq \sum_{i=2}^k r(u_i) + \sum_{i=1}^k n(u_i) - r(u_i) - 1 \\ &= (n(v) - 1) - r(u_1) - k \leq n(v) - r(v) - 1. \end{aligned}$$

This concludes the inductive proof and implies that $\sum_v q(v) \leq n - 1$. Putting everything together we get

$$\sum_v k(v) + q(v) + t(v) \leq (n - 1) + (n - 1) + 3(n - 1) = O(n).$$

Thus proving that the time required to compute the optimal strategy function along with the intermediate visibility sequences at all nodes is linear in n . ■

6.5 From a Strategy Function to a Decision Tree in Linear Time

In this section we show how to construct an optimal decision tree in linear time using an optimal strategy function. We begin with some additional definitions and assumptions. To avoid confusion between the input tree and the decision tree we will refer to the decision tree by the acronym DT. Each node in DT represents an edge in the input tree. We refer to each node in DT by the edge in the input tree which it represents. Let r be the root of the tree. We introduce an additional node r' , and connect it to r . We assign the value ∞ to the edge (r', r) , and from now on we treat r' as

a root of the tree. For an edge e , let $top(e)$ denote the end point of e closer to the root of the tree and $bottom(e)$ the end point of e farther from the root. The node e in DT will have exactly two children, one for the case when the query about e returns $top(e)$ and the other for the case when it returns $bottom(e)$.

Initially, the nodes in DT are disconnected. We now describe an algorithm that uses the computed strategy function and visibility sequences to connect the nodes in DT. We assume that along with each value in each visibility sequence we keep a link to the edge to which that value is assigned. Clearly, this can be done within the same time complexity.

The Algorithm. We process all edges of the input tree in any order. For each edge e , let s be the visibility sequence at $bottom(e)$.

If s contains no values smaller than $f(e)$, then

(1) set $bottom(e)$ as the solution in DT when the query about e returns $bottom(e)$.

Otherwise, let $v_1 < v_2 < \dots < v_k$ be the values smaller than $f(e)$ in s , and let e_i be the edge v_i is assigned to.

(2a) set node e_k in DT as the solution when the query on e returns $bottom(e)$.

(2b) for every $1 \leq i < k$ set the node e_i in DT as the solution when the query on e_{i+1} returns $top(e_{i+1})$.

(2c) set $top(e_1)$ to be the solution in DT when the query on e_1 returns $top(e_1)$.

Finally, after applying the above procedure to all edges in the input tree, the root of DT is the only child of the DT node (r', r) .

Correctness and Time Complexity. Let e be an arbitrary edge. We prove that in DT, the children of e corresponding to the answers $top(e)$ and $bottom(e)$ are assigned correctly and exactly once. It is easy to see that the algorithm assigns $bottom(e)$ exactly once – when we process edge e . Recall that if the query on e returns $bottom(e)$, then the next query should be on the edge with largest value smaller than $f(e)$ which is visible from $bottom(e)$ in the subtree rooted at $bottom(e)$. This is

done in (2a). If there is no such value, there is no next query either, and $bottom(e)$ is the solution to the searching problem. This case is handled in (1). Therefore, $bottom(e)$ is assigned correctly and exactly once for each e in the input tree.

We now show that $top(e)$ is also assigned correctly and exactly once. Let e' be the first edge with value greater than $f(e)$ on the path from $top(e)$ to r' . Such an edge always exists since we assigned the value ∞ to the edge (r', r) . First notice that $top(e)$ is assigned exactly once – when we process edge e' . It therefore only remains to show that $top(e)$ is assigned correctly. Recall that when a query about e returns $top(e)$, then the next query should be on the edge with greatest value smaller than $f(e)$ that is visible from $top(e)$ in the entire tree (viewing the tree as unrooted). Note that this edge is also the edge with greatest value smaller than $f(e)$ that is visible from $bottom(e')$ in the subtree rooted at $bottom(e')$ (viewing the tree as rooted). If such an edge exists, we make it a child of e in (2b). If there is no such edge, then obviously, when the query about e results in $top(e)$, then $top(e)$ is the solution to the searching problem. We handle this case in (2c). Therefore, $top(e)$ is assigned correctly and exactly once for each e in the input tree. We have thus established the correctness.

To analyze the time complexity, notice that for each edge e , the algorithm runs in time proportional to the number of values in the visibility sequence of $bottom(e)$ that are screened by $f(e)$. Since no value can be screened twice, the total runtime is linear.

Bibliography

- [1] O. Agazzi and S. Kuo. HMM based optical character recognition in the presence of deterministic transformations. *Pattern Recognition*, 26:1813–1826, 1993.
- [2] A. Aggarwal and M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2):3–23, 1990.
- [3] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.
- [4] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Comp. and Sys. Sciences*, 52(2):299–307, 1996.
- [5] A. Amir, G.M. Landau, and D. Sokol. Inplace 2d matching in compressed images. In *Proc. of the 14th annual ACM-SIAM Symposium On Discrete Algorithms, (SODA)*, pages 853–862, 2003.
- [6] A. Apostolico, M.J. Atallah, L.L. Larmore, and S.McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.
- [7] A. Apostolico and Z. Galil, editors. *Pattern matching algorithms*. Oxford University Press, Oxford, UK, 1997.
- [8] A. Apostolico, G.M. Landau, and S. Skiena. Matching for run length encoded strings. *Journal of Complexity*, 15(1):4–16, 1999.
- [9] O. Arbell, G. M. Landau, and J. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2001.
- [10] V. L. Arlazarov, E. A. Dinic, A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11(5):1209–1210, 1970.
- [11] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, 11:1209–1210, 1975.
- [12] R. Backofen, S. Chen, D. Hermelin, G.M. Landau, M.A. Roytberg, O. Weimann, and K. Zhang. Locality and gaps in RNA comparison. *Journal of Computational Biology (JCB)*, 14(8):1074–1087, 2007.

- [13] L.E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process. *Inequalities*, 3:1–8, 1972.
- [14] Y. Ben-Asher, E. Farchi, and I. Newman. Optimal search in trees. *SIAM Journal on Computing*, 28(6):2090–2102, 1999.
- [15] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337:217–239, 2005.
- [16] P. Bille and M. Farach-Colton. Fast and compact regular expression matching. *CoRR*, cs.DS/0509069, 2005.
- [17] A.P. Bird. CpG-rich islands as gene markers in the vertebrate nucleus. *Trends in Genetics*, 3:342–347, 1987.
- [18] A. L. Buchsbaum and R. Giancarlo. Algorithmic aspects in speech recognition: An introduction. *ACM Journal of Experimental Algorithms*, 2:1, 1997.
- [19] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run length coded strings. *Information Processing Letters*, 54:93–96, 1995.
- [20] Burkard, Klinz, and Rudolf. Perspectives of monge properties in optimization. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 70, 1996.
- [21] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. Laber. Searching in random partially ordered sets. *Theoretical Computer Science*, 321(1):41–57, 2004.
- [22] T.M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. In *Proceedings of the 9th Workshop on Algorithms and Data Structures (WADS)*, pages 318–324, 2005.
- [23] T.M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the 39th ACM Symposium on Theory of Computing (STOC)*, pages 590–598, 2007.
- [24] M. Charikar, R. Fagin, V. Guruswami, J. Kleinberg, P. Raghavan, and A. Sahai. Query strategies for priced information. *Journal of Computer and System Sciences*, 64(4):785–819, 2002.
- [25] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 90–101, Edinburgh, Scotland, U.K., 1999.
- [26] W. Chen. New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40:135–158, 2001.
- [27] G.A. Churchill. Hidden Markov chains and the analysis of genome structure. *Computers Chem.*, 16:107–115, 1992.

- [28] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetical progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.
- [29] M. Crochemore, G.M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. on Computing*, 32:1654–1673, 2003.
- [30] P. de la Torre, R. Greenlaw, and A.A. Schaffer. Optimal edge ranking of trees in polynomial time. *Algorithmica*, 13:529–618, 1995.
- [31] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 146–157, 2007.
- [32] S. Dulucq and H. Touzet. Analysis of tree edit distance algorithms. In *Proceedings of the 14th annual symposium on Combinatorial Pattern Matching (CPM)*, pages 83–95, 2003.
- [33] R. Durbin, S. Eddy, A. Krigh, and G. Mitcheson. *Biological Sequence Analysis*. Cambridge University Press, 1998.
- [34] Y. Emek, D. Peleg, and L. Roditty. A near-linear time algorithm for computing replacement paths in planar directed graphs. In *SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM symposium on Discrete Algorithms*, pages 428–435, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.
- [35] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.
- [36] D. E. Ferguson. Fibonacci searching. *Communications of the ACM*, 3(12):648, 1960.
- [37] G. Fettweis and H. Meyr. High-rate viterbi processor: A systolic array solution. *IEEE Journal on Selected Areas in Communications*, 8(8):1520–1534, 1990.
- [38] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.
- [39] P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.
- [40] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995.
- [41] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.
- [42] D. Gusfield. *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. Press Syndicate of the University of Cambridge, 1997.
- [43] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

- [44] M. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.
- [45] D. Hermelin, G.M. Landau, S. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *Proceedings of the 26th international Symposium on Theoretical Aspects of Computer Science (STACS)*, 2009.
- [46] A. J. Hoffman. On simple linear programming problems. Technical report, Summerschool, Sogesta, Urbino, 1978.
- [47] A.V. Iyer, H.D. Ratliff, and G. Vijayan. On an edge ranking problem of trees and graphs. *Discrete Appl. Math.*, 30:43–52, 1991.
- [48] J. Karkkainen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. *Proc. of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 195–209, 2000.
- [49] J. Karkkainen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. of the 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.
- [50] M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In *Proc. of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 205–214, 1995.
- [51] M. Klawe and D. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal On Discrete Math*, 3(1):81–97, 1990.
- [52] P. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: a linear-space $o(n \log^2 n)$ -time algorithm. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 236–245, 2009.
- [53] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th annual European Symposium on Algorithms (ESA)*, pages 91–102, 1998.
- [54] P. N. Klein, S. Tirthapura, D. Sharvit, and B. B. Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 696–704, 2000.
- [55] W. J. Knight. Search in an ordered array having variable probe cost. *SIAM Journal on Computing*, 17(6):1203–1214, 1988.
- [56] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.
- [57] E. Laber and L. T. Nogueira. Fast searching in trees. *Electronic Notes in Discrete Mathematics*, 7:1–4, 2001.

- [58] L.W. Lam and F.L. Yue. Optimal edge ranking of trees in linear time. *Algorithmica*, 30(1):12–33, 2001.
- [59] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proc. of the 13th annual ACM-SIAM Symposium On Discrete Algorithms, (SODA)*, pages 205–212, 2002.
- [60] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
- [61] Y. Lifshits. Processing compressed texts: A tractability border. In *Proc. of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 228–240, 2007.
- [62] Y. Lifshits and M. Lohrey. Querying and embedding compressed texts. In *Proc. of the 31st international symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 681–692, 2006.
- [63] Y. Lifshits, S. Mozes, O. Weimann, and M. Ziv-Ukelson. Speeding up HMM decoding and training by exploiting sequence repetitions. *Algorithmica*, to appear, 2008.
- [64] H.D. Lin and D.G. Messerschmitt. Algorithms and architectures for concurrent viterbi decoding. *IEEE International Conference on Communications*, 2:836–840, 1989.
- [65] N. Linial and M. Saks. Every poset has a central element. *Journal of combinatorial theory*, A 40(2):195–210, 1985.
- [66] N. Linial and M. Saks. Searching ordered structures. *Journal of algorithms*, 6(1):86–103, 1985.
- [67] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.
- [68] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.
- [69] V. Makinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. *Proc. of the 12th Annual Symposium On Combinatorial Pattern Matching (CPM)*, pages 1–13, 1999.
- [70] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc of the 5th Annual Symposium On Combinatorial Pattern Matching (CPM)*, pages 31–49, 1994.
- [71] C. Manning and H. Schutze. *Statistical Natural Language Processing*. The MIT Press, 1999.
- [72] W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *J. of Computer and System Sciences*, 20(1):18–31, 1980.

- [73] G. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986.
- [74] G. Miller and J. Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24(5):1002–1017, 1995.
- [75] J. Mitchell. A geometric shortest path problem, with application to computing a longest common subsequence in run-length encoded strings. *Technical Report, Dept. of Applied Mathematics, SUNY Stony Brook*, 1997.
- [76] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 1–11, 1997.
- [77] G. Monge. Mémoire sur la théorie des déblais et ramblais. *Mém. Math. Phys. Acad. Roy. Sci. Paris*, pages 666–704, 1781.
- [78] P.B Moore. Structural motifs in RNA. *Annual review of biochemistry*, 68:287–300, 1999.
- [79] S. Mozes. private communication. 2008.
- [80] S. Mozes, K. Onak, and O. Weimann. Finding an optimal tree searching strategy in linear time. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1096–1105, 2008.
- [81] G. Navarro, R. Baeza-Yates, E. Barbosa, N. Ziviani, and W. Cunto. Binary searching with non-uniform costs and its application to text retrieval. *Algorithmica*, 27(2):145–169, 2000.
- [82] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. *Proc of the Data Compression Conference (DCC)*, pages 459–468, 2001.
- [83] K. Onak and P. Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *Proceedings of the 47th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 379–388, 2006.
- [84] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.
- [85] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.
- [86] J.P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.
- [87] S.M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.

- [88] D. Shasha and K. Zhang. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.
- [89] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581–621, 1990.
- [90] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte Pair encoding: A text compression scheme that accelerates pattern matching. *Technical Report DOI-TR-161, Department of Informatics, Kyushu University*, 1999.
- [91] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. of the 4th Italian Conference Algorithms and Complexity (CIAC)*, pages 306–315, 2000.
- [92] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. *Lecture Notes in Computer Science*, 1767:306–315, 2000.
- [93] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
- [94] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
- [95] K. Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery (JACM)*, 26(3):422–433, 1979.
- [96] A. Tiskin. All semi-local longest common subsequences in subquadratic time. In *Proc. of the first international Computer Science symposium in Russia (CSR)*, pages 352–363, 2006.
- [97] A. Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences, to appear*, 2008.
- [98] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.
- [99] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, 1967.
- [100] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
- [101] M.S. Waterman. *Introduction to computational biology: maps, sequences and genomes*, chapters 13,14. Chapman and Hall, 1995.
- [102] K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463–474, 1995.

- [103] J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
- [104] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
- [105] M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244(4900):48–52, 1989.
- [106] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, 1981.