國立臺灣大學電機資訊學院資訊工程學系

博士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering & Computer Science

National Taiwan University

doctoral dissertation

條件限制下的序列搜尋與排序

Constrained Searching and Ordering Problems on Sequences

劉效飛

Hsiao-Fei Liu

指導教授：趙坤茂 博士

Advisor: Dr. Kun-Mao Chao

中華民國 97 年 6 月

June, 2008

誌　謝

六月十七號凌晨一點二十三分，我開始寫這篇誌謝，代表一段旅程已接近尾聲。

感謝我的指導教授趙坤茂老師和 ACB 的好朋友們: 耀廷、佳熒、容任、昆宜、弘倫、冠宇、志亘、小錦、一軒、揚和、寶萱、偉哥、家榮、爽哥、維均、Juju、帥朋、謦儀、Popo、Roger、智鐸、小孟、晏禕、煜樟、韋仰、韋霖、明江、Wasabe、大餅、強哥、琨哥、佑任。謝謝你們的善良和快樂，與你們共處的這段時光，才是我這趟旅程的意義。這些日子的點點滴滴，都將是我珍貴的回憶，記憶中的畫面也許會隨著時間而有些模糊，但曾共同經歷的感動卻不會淡忘。

感謝小時候的家教黃坤常老師和師母，十分謝謝黃老師和師母當初對我的耐心教導，這份恩情我一直牢記在心中。有一次上課，老師和師母讓我們自己挑了一個英文名字，當時我選了 Ken，這個英文名字我一直記在心裡，我在 BBS 上用的代號 Forken，意思就是“for Ken”。

感謝在政大資科時的專題指導教授沈錳坤老師，謝謝沈老師當初的細心指導和照顧，尤其老師對我的關心，我真的由衷感激。每當有人問我是哪裡畢業的，我總是很驕傲的說:“政大資科。”因為我曾在那遇到了最關心學生的好老師。

感謝張雅惠教授在我寫第一篇論文時，給予了相當多的幫忙和指導，讓我獲益良多。

感謝呂育道教授在論文計畫審查時，花了相當多的時間閱讀修改我的論文，並提出許多寶貴的建議，真的十分謝謝。

感謝陳健輝教授在論文口試時發人省思的提問，指引我未來努力的方向。

感謝林耀玲教授、徐熊健教授、歐陽彥正教授和劉邦鋒教授在百忙之中仍挪出空檔參與我的論文計畫審查和口試，指出許多疏失並給予指導。

最後，我想引用張懸的一段歌詞當結尾:

擁有一切之後
就讓它走
在某個角落放一首歌
別忘了 要溫柔
別忘了 要快樂

感謝過去五年來所擁有的一切，
祝福所有 ACB 的夥伴們都能溫柔、快樂。

珍重再見

六月十八號凌晨四點五十分，我完成這篇誌謝，旅程結束前，讓我再一次祝福我的指導教授，趙坤茂老師，生日快樂。

摘　　要

在這份論文中，我們將探討一系列有關序列分析的計算問題。這些問題依據其性質，可分為以下四類：最佳化、列舉、挑選、排序。其中最佳化、列舉、挑選這三類問題性質較為接近，所以又統稱為搜尋問題。在最佳化問題中，我們的目標是找出最佳解；在列舉問題中，我們希望能將前 k 好的解都列舉出來；在挑選問題中，我們希望能挑選出第 k 好的解；在排序問題中，我們必須重新排序一個序列，使得這個序列滿足給定的限制。本論文中所討論的問題都已知能在多項式時間內解決，所以我們的目標將鎖定在設計能更快找出精確解的演算法。

Abstract

In this dissertation, we study a series of problems on sequences. These problems are broadly categorized into four types: Optimization, enumeration, selection, and ordering. Problems of the first three types are generally called the searching problems. In the optimization problem, we seek for the best feasible solution. In the enumeration problem, we have to enumerate the k best feasible solutions. In the selection problem, we pick out the k^{th} best feasible solution. In the ordering problem, we are required to reorder a sequence subject to some constraints. All problems considered in this dissertation are already known to be polynomial-time solvable, so we aim at giving efficient exact algorithms for them.

Contents

List of Figures

Chapter 1

Introduction

Optimization, enumeration, selection, and ordering are four basic types of algorithmic problems. In the optimization problem, the goal is to find the best feasible solution. In the enumeration problem, it is no longer satisfactory if only the best feasible solution is found, and we are required to enumerate the k best feasible solutions. In the selection problem, we are interested in finding the k^{th} best feasible solution instead of the best feasible solution. In the ordering problem, we have to reorder a sequence subject to some constraints. Problems of the first three types are generally called the searching problems. In this dissertation, we study a series of searching and ordering problems on sequences and provide efficient algorithms for them.

1.1 Preliminaries

When analyzing the running time, we adopt the real RAM model of computation in which each simple instruction (e.g., $+, -, \times, \div, \sqrt{}$, or $\leq$) takes unit time.

Given a sequence $A = (a_1, a_2, \ldots, a_n)$ of n real numbers, define the length, sum, and density of a segment $A[i, j] = (a_i, \ldots, a_j)$ to be $length(i, j) = j - i + 1$, $sum(i, j) = \sum_{h=i}^{j} a_h$, and $d(i, j) = \frac{sum(i,j)}{length(i,j)}$, respectively. Under the real RAM model, it is clear that $length(i, j) = j-i+1$ can be evaluated in constant time. To evaluate $sum(i, j) = \sum_{h=i}^{j} a_h$ in constant time, we have to construct the prefix-sum array of sequence A first. An array $P_a[0..n]$ is said to be the prefix-sum array of sequence A if and only if $P_a[i] = a_1 + a_2 + \cdots + a_i$ for each $i > 0$ and $P_a[0] = 0$. The prefix-sum array $P_a[0..n]$ can be computed in linear time by setting $P_a[0]$ to 0

and $P_a[i]$ to $P_a[i-1]+a_i$ for i from 1 to n. After constructing the prefix-sum array $P_a[0..n]$, evaluation of $sum(i,j)$ can be done in constant time as $sum(i,j) = (P_a[j] - P_a[i-1])$. Since $d(i,j) = \frac{sum(i,j)}{length(i,j)}$, it follows that evaluation of $d(i,j)$ can also be done in constant time after we construct $P_a[0,\ldots,n]$.

More generally, given a sequence $D = ((s_1,\ell_1),(s_2,\ell_2),\ldots,(s_n,\ell_n))$ of number pairs, where $\ell_i > 0$ for all i, we define the sum, length, and density of a segment $D[i,j] = ((s_i,\ell_i),\ldots,(s_j,\ell_j))$ to be $sum(i,j) = \sum_{r=i}^{j} s_r$, $length(i,j) = \sum_{r=i}^{j} \ell_r$, and $d(i,j) = \frac{sum(i,j)}{length(i,j)}$, respectively. In addition, we define the sum, length, and density of an index interval $[i,j] = \{i, i+1, \ldots, j\}$ with respect to sequence D in the same way. Let $P_s[0..n]$ be the prefix-sum array of sequence $(s_1, s_2, \ldots, s_n)$ and $P_\ell[0..n]$ be the prefix-sum array of sequence $(\ell_1, \ell_2, \ldots, \ell_n)$. Note that $sum(i,j) = P_s[i] - P_s[j-1]$, $length(i,j) = P_\ell[i] - P_\ell[j-1]$, and $d(i,j) = \frac{sum(i,j)}{length(i,j)}$. Therefore, by constructing P_s and P_ℓ first, evaluation of $sum(i,j)$, $length(i,j)$, or $d(i,j)$ can be done in constant time.

1.2 Problem Formulation and Contribution

In the following, we give formulation of problems studied in this dissertation and state our contribution toward these problems. The results are summarized in Figure 1.1.

- Optimization Problems:
  - Disjoint Segments with Maximum Sum of Densities. Given a sequence of n real numbers $A = (a_1, a_2, \ldots, a_n)$ and two positive integers l and k, where $k \leq \frac{n}{l}$, the problem is to locate k disjoint segments of A, each of length at least l, such that the sum of their densities is maximized. For the case where $k = 1$, this problem was well studied in biomolecular sequence analysis [15, 25, 41, 44, 48, 53] and can be solved in linear time [15, 25, 41]. For general k, Chen *et al.* [22] proposed an $O(nkl)$-time algorithm and an improved $O(nl + k^2l^2)$-time algorithm was given by Bergkvist and Damaschke [14]. In this dissertation, give an $O(n + k^2 l \log l)$-time algorithm for this problem.
  - Length-Constrained Max-Eccentricity Segments. Given a sequence of

n real numbers $A = (a_1, a_2, \ldots, a_n)$ and an integer L, define the eccentricity of a segment $A[i, j]$ to be $ecc(i, j) = \frac{sum(i,j)}{\sqrt{length(i,j)}}$. The problem is to find a segment $A[i, j]$ maximizing $ecc(i, j)$ subject to $length(i, j) \geq L$. We give an algorithm which runs in $O(n\frac{T(L^{1/2})}{L^{1/2}})$ time, where $T(n')$ is the time required to solve the all-pairs shortest paths problem on a graph of n' nodes. By the latest result of Chan [19], $T(n') = O(n'^3\frac{(\log\log n')^3}{(\log n')^2})$, so our algorithm runs in subquadratic time $O(nL\frac{(\log\log L)^3}{(\log L)^2})$. To the best of our knowledge, it is the first subquadratic result for this problem. Lipson *et al.* [56] studied a more restricted case where there is no length constraint, i.e., $L = 1$. They proposed an efficient approximation scheme for this case; however, their algorithm needs quadratic time if exact solutions are required. Since the length lower bound L for the case considered by Lipson *et al.* is a constant, our algorithm solves it in $O(n)$ time.

- Sum-Constrained Max-Density Intervals.

  Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2) \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a number L, the problem is to find an index interval $[i, j]$ maximizing $d(i, j)$ subject to $sum(i, j) \geq L$. In this dissertation, we give an $O(n)$-time algorithm for this problem.

- Density Finding. Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2) \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for $i = 1, 2, \ldots, n$, two positive numbers L, U with $L < U$, and a real number δ, the Density Finding Problem is to compute the density of the index interval $[i, j]$ which minimizes $|d(i, j) - \delta|$ subject to $L \leq length(i, j) \leq U$. Lee *et al.* [51] proved that the Density Finding Problem has a lower bound of $\Omega(n \log n)$ in the algebraic decision tree model and provided an $O(n \log^2 m)$ algorithm for it, where $m = \min(\lfloor\frac{U-L}{\ell_{min}}\rfloor, n)$ and $\ell_{min} = \min_{1\leq k\leq n} \ell_k$. In this dissertation, we give an $O(n \log n)$-time algorithm for the Density Finding Problem.

- Enumeration Problems:

  - Length-Constrained k Max-Sum Segments. Given a sequence of n real numbers $A = (a_1, a_2, \ldots, a_n)$, two integers L and U with $1 \leq L \leq U \leq n$, and a positive integer k, the problem is to find the k max-sum segments among all segments with

lengths between L and U. For the case where k is fixed to 1, this problem is studied in [31, 53] and can be solved in linear time [31]. For the case where L is fixed to 1 and U is fixed to n, this problem was well studied in [5, 6, 11, 18, 23, 30, 54] and can be solved in $O(n+k)$ time [18, 30]. Recently, Lin and Lee [55] proposed an expected $O(n\log(U-L+1)+k)$-time algorithm for the general case. In this dissertation, we give an optimal $O(n+k)$-time algorithm for the general case.

– Weight-Constrained k Longest Paths. Given a tree $T=(V,E)$ with a length function $\ell : E \to \mathbb{R}$ and a weight function $w : E \to \mathbb{R}$, a positive integer k, and an interval $[L,U]$, the problem is to find the k longest paths among all paths in T with weights in the interval $[L,U]$. For the case where there are no weight constraints, i.e., $L=-\infty$ and $U=\infty$, Megiddo *et al.* [61] proposed an $O(|V|\log^2|V|+k)$-time algorithm and Frederickson and Johnson [36] proposed an $O(|V|\log|V|+k)$-time algorithm. Wu *et al.* [74] gave an $O(|V|\log^2|V|)$-time algorithm for the case where $L=-\infty$ and $k=1$. Kim [49] gave an $O(|V|\log|V|)$-time algorithm for the case where $L=0$, $U=\infty$, $k=1$, $\ell(e)=1\ \forall e\in E$, and T has a constant degree. In this dissertation, we give an $O(|V|\log|V|+k)$-time algorithm for the general case. In addition, we prove that this problem has a lower bound of $\Omega(|V|\log|V|+k)$ in the algebraic computation tree model.

- Selection Problems:

  – Length-Constrained Sum Selection. Given a sequence $A=(a_1,a_2,\ldots,a_n)$ of $n\geq 2$ real numbers and two positive integers L, U with $1\leq L<U\leq n$, the Length-Constrained Sum Selection problem, for a given k, is to find the k^{th} largest sum among all sums of segments of A with lengths in $[L,U]$. When there are no length constraints, i.e., $L=1$ and $U=n$, the Length-Constrained Sum Selection problem becomes the Sum Selection Problem. Bengtsson and Chen [11] first studied the Sum Selection Problem and gave an $O(n\log^2 n)$-time algorithm for it. Lin and Lee provided an $O(n\log n)$-time algorithm [54] for the Sum Selection Problem and an expected $O(n\log(U-L+1))$-time randomized algorithm [55] for the Length-Constrained Sum Selection problem. We give an

$O(n \log(U - L + 1))$-time algorithm for the Length-Constrained Sum Selection problem. In addition, we prove that the Sum Selection Problem has an $\Omega(n \log n)$ lower bound in terms of n.

- Ordering Problems:

  - Online Topological Ordering. A topological order *ord* of a directed acyclic graph (DAG) $G = (V, E)$ is a linear order of its vertices such that if G contains an edge (u, v), then $ord(u) < ord(v)$. We study an online variant of the topological ordering problem in which the edges of the DAG are given one at a time and we have to update the order *ord* each time an edge is added. A naïve algorithm for this problem is to compute a new topological order from scratch with the offline algorithm whenever an edge is inserted. Since the offline algorithm runs in linear time, this naïve algorithm takes $O(m^2 + mn)$ time for inserting m edges. Marchetti-Spaccamela *et al.* [60] made the first breakthrough by giving an $O(mn)$-time algorithm for m edge insertions. Alpern *et al.* [4] proposed an algorithm but did not analyze the time complexity of their algorithm for a sequence of edge insertions. Katriel and Bodlaender [47] proposed a slight variation of Alpern *et al.*'s algorithm, which is referred to as the Katriel-Bodlaender algorithm in this dissertation, and showed that their variation has an $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$ upper bound on running time for m edge insertions. This is the best result for sparse graphs so far. On the other hand, Ajwani *et al.* [2] proposed an $O(n^{2.75})$-time algorithm, independent of the number of edges inserted. This is the best result for dense graphs so far. The only non-trivial lower bound is due to Ramalingam and Reps [68], who showed that any algorithm needs $\Omega(n \log n)$ time while inserting $n - 1$ edges in the worst case if all labels are maintained explicitly. For sparse graphs, we improve the time upper bound from $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$ to $O(m^{3/2} + mn^{1/2} \log n)$ through a tighter analysis of the Katriel-Bodlaender algorithm. For dense graphs, we obtain an improved $\tilde{O}(n^{2.5})$-time algorithm.[1]

[1] The symbol $\tilde{O}$ means O with log factors ignored. Depending on the implementation, the running time may vary from $O(n^{2.5} \log^2 n)$ to $O(n^{2.5} \log n)$.

Figure 1.1: Results on problems studied in this dissertation.

Problem	Previous best result	Our result
Disjoint Segments with Maximum Sum of Densities	$O(nl + k^2l^2)$ [14]	$O(n + k^2 l \log l)$
Length-Constrained Max-Eccentricity Segments	$O(n^2)^*$	$O(nL\frac{(\log\log L)^3}{(\log L)^2})$
Sum-Constrained Max-Density Intervals	$O(n \log n)$ [15]	$O(n)$
Density Finding	$O(n \log^2 n)$ [51]	$O(n \log n)$
Length-Constrained k Max-Sum Segments	Expected $O(n \log(U - L + 1) + k)$ [55]	$O(n + k)$
Weight-Constrained k Longest Paths	$O(\|V\|^2)^*$	$O(\|V\| \log \|V\| + k)$
Length-Constrained Sum Selection	Expected $O(n \log(U - L + 1))$ [55]	$O(n \log(U - L + 1))$
Online Topological Ordering	$O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$ [47] $O(n^{2.75})$ [2]	$O(m^{3/2} + mn^{1/2} \log n)$ $\tilde{O}(n^{2.5})$

* Formerly only naïve algorithms were known for the general case.

1.3 Motivation

In this section, we describe the origins and applications of problems studied in this dissertation.

Searching Problems

First, let us introduce two more problems: the Length-Constrained Max-Density Segments problem and the Length-Constrained Max-Sum Segments problem. All searching problems studied in this dissertation are either generalizations or variations of these two problems. The relation is depicted in Figure 1.2. Given a sequence of real numbers and a length lower bound, the Length-Constrained Max-Density Segments problem is to locate a segment that maximizes the density subject to the length lower bound; and the Length-Constrained Max-Sum Segments problem is to locate a segment that maximizes the sum subject to the length lower bound.

Both of the two problems are first formulated by Huang [44] and motivated by their use in finding GC-enriched regions of a DNA sequence. After that, many generalizations are formulated one after another to fulfill more requirements. The Disjoint Segments with Maximum Sum of Densities problem and the Density Finding problem are two generalizations of the Length-Constrained Max-Density Segments problem and first formulated by Chen, Lu, and Tang [22] and Lee, Lin, and Lu [51], respectively. The Length-Constrained Sum Selection problem and the Length-Constrained k Max-Sum Segments problem are two generalizations of the Length-Constrained Max-Sum Segments problem and first formulated by Lin and Lee [55]. In this dissertation, we further generalize the Length-Constrained k Max-Sum Segments problem to the Weight-Constrained k Longest Paths problem.

On the other hand, more and more variations are found in other applications. The Length-Constrained Max-Density Segments problem and its variation, the Sum-Constrained Max-Density Segments problem, are found in mining association rules for numerical attributes [39]. The Length-Constrained Max-Eccentricity Segments problem, which can be regarded as a variation of the Length-Constrained Max-Sum Segments problem, is applied to detecting DNA copy number variations [56].

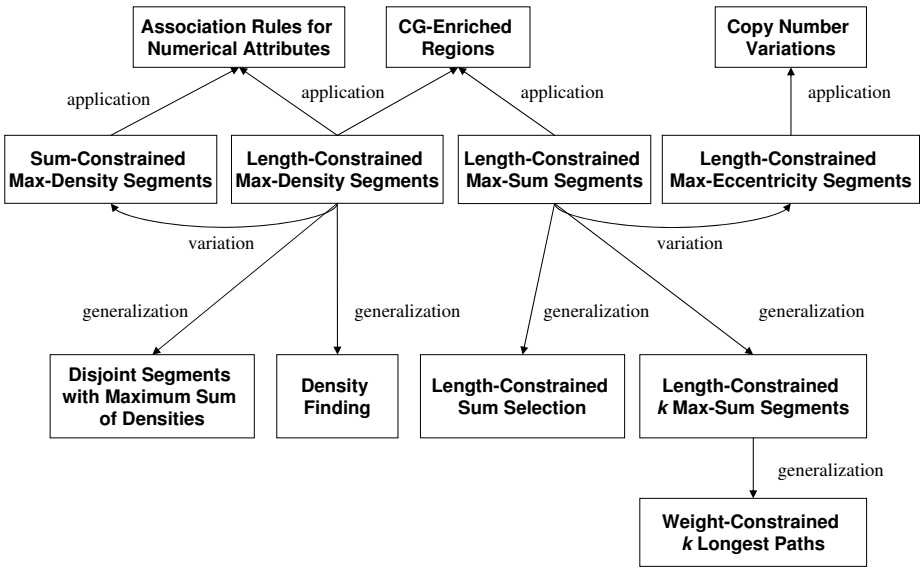


Figure 1.2: Relation between searching problems.

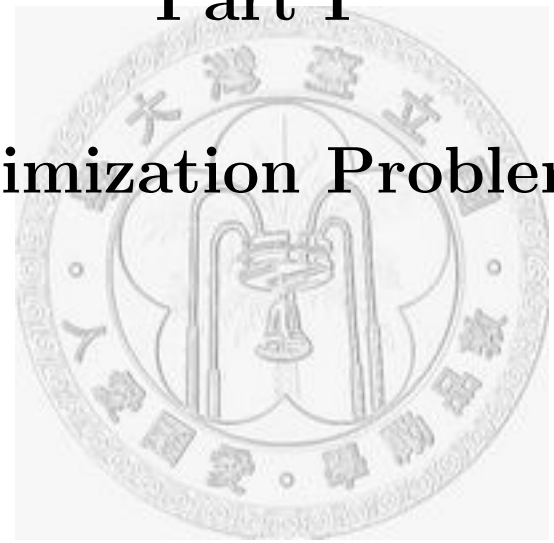
Ordering Problems

Now, let us look at the Online Topological Ordering problem. When dealing with DAGs, the topological order of vertices often provides very crucial information for further algorithm development. Thus online topological ordering is of interests because it is very likely to be required when one has to develop online algorithms on DAGs. For example, the online topological ordering has appeared in the following contexts.

- Incremental Evaluation of Computational Circuits [4].
- Incremental Compilation [59, 63], where dependencies between modules are maintained to reduce the amount of recompilation performed when an update occurs.
- Online Computation of Strongly Connected Components [64].
- Online Cycle Detection [47, 64, 66]. Currently the best online cycle detection algorithm for sparse directed graphs is built upon the Katriel-Bodlaender algorithm and has the same complexity as the Katriel-Bodlaender algorithm. Thus our analysis improves the upper bound of the online cycle detection problem to $O(m^{3/2} + mn^{1/2} \log n)$.
- Source Code Analysis [64, 66], where the aim is to determine the target set for all pointer variables in a program, without executing it.

Part I

Optimization Problems



Chapter 2

Disjoint Segments with Maximum Sum of Densities

Given a sequence $A = (a_1, a_2, \ldots, a_n)$ of n real numbers and two positive integers l and k, where $k \leq \frac{n}{l}$, the Disjoint Segments with Maximum Sum of Densities (DSMSD) problem is to find k disjoint segments $\{s_1, s_2, \ldots, s_k\}$ of A, each of length at least l, such that $\sum_{1 \leq i \leq k} d(s_i)$ is maximized.

For $k = 1$, the DSMSD problem was well studied in biomolecular sequence analysis [15, 25, 41, 44, 48, 53] and data mining [39]. For general k, Chen *et al.* [22] proposed an $O(nkl)$-time algorithm and an improved $O(nl + k^2l^2)$-time algorithm was given by Bergkvist and Damaschke [14]. In this chapter, we propose an $O(n + k^2 l \log l)$-time algorithm.

Lin *et al.* [52] formulated a related problem: Given a sequence A of n real numbers and two positive integers l and k, where $k \leq \frac{n}{l}$, find a sequence $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_k)$ of k disjoint segments of A such that for all i, γ_i is either a maximum-density segment of length between l and $2l - 1$ not overlapping any of the first $i - 1$ segments of Γ or NIL if all segments of length between l and $2l - 1$ overlap some of the first $i - 1$ segments of Γ. For example, let $A = (1, 2, 2, 5, 5, 5, 3, 3, 2)$, $l = 2$, and $k = 4$. Then there are three solutions: $(A[4, 6], A[7, 8], A[2, 3], \text{NIL})$, $(A[4, 5], A[6, 7], A[8, 9], A[2, 3])$, and $(A[5, 6], A[3, 4], A[7, 8], A[1, 2])$. We call this related problem the Disjoint Maximum-Density Segments (DMDS) problem. For the DMDS problem, Lin *et al.* [52] proposed an algorithm which runs in $O(n \log k)$ time in some particular situations but needs $\Omega(nk)$ time in the worst case. In this chapter, an optimal

$O(n + k \log k)$-time algorithm for the DMDS problem is given.

The rest of this chapter is organized as follows. In Section 2, we introduce some preliminary knowledge. In Section 3, we propose an optimal algorithm for the DMDS problem. In Section 4, we show how to reduce the length of the input sequence of the DSMSD problem by using the algorithm developed in Section 3. In Section 5, we give our main algorithm for the DSMSD problem. Section 6 gives some concluding remarks.

2.1 Preliminaries

The following lemma was proved in [22].

Lemma 2.1: Let $S' = \{s'_1, s'_2, \ldots, s'_k\}$ be a set of k disjoint segments of length at least l such that $\sum_{1 \le i \le k} d(s'_i)$ is maximized. There exists a set $S = \{s_1, s_2, \ldots, s_k\}$ of k disjoint segments of length between l and $2l - 1$ such that $\sum_{1 \le i \le k} d(s_i) = \sum_{1 \le i \le k} d(s'_i)$.

Lemma 2.1 states that there exists a solution to the DSMSD problem instance (A, k, l) composed of segments of length between l and $2l - 1$. It allows us to reformulate the DSMSD problem as follows: Given a sequence $A = (a_1, a_2, \ldots, a_n)$ of n real numbers and two positive integers l and k, where $k \le \frac{n}{l}$, find k disjoint segments $\{s_1, s_2, \ldots, s_k\}$ of A, each of length between l and $2l - 1$, such that $\sum_{1 \le i \le k} d(s_i)$ is maximized. From now on, we adopt this problem formulation.

2.2 The DMDS Problem

Given a sequence A of n real numbers and two positive integers l and k, where $k \le \frac{n}{l}$, the Disjoint Maximum-Density Segments (DMDS) problem is to find a sequence $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_k)$ of k disjoint segments of A such that for all i, γ_i is either a maximum-density segment of length between l and $2l - 1$ not overlapping any of the first $i - 1$ segments of Γ or NIL if all segments of length between l and $2l - 1$ overlap some of the first $i - 1$ segments of Γ.

In this section, we give an optimal $O(n + k \log k)$ algorithm for the DMDS problem, which appears to be a digression at first. Later, in Section 2.3, it turn out that the algorithm developed

here can be used to compress an input sequence of the DSMSD problem of length n into a sequence of length $O(kl)$.

The algorithm for the DMDS problem is given in Figure 2.1. For simplicity, we assume that $4l$ divides n. Otherwise let $\hat{a} = \max\{|a_1|, |a_2|, \ldots, |a_n|\} + 1$ and $E = -4l \cdot \hat{a}$. We append $\overbrace{(E, E, \ldots, E)}^{(4l \cdot \lceil \frac{n}{4l} \rceil - n)}$ to the end of the sequence A and run the algorithm on the extended sequence. Note that a segment (with length between l and $2l-1$) of the extended sequence has density greater than or equal to $-\hat{a}$ if and only if it contains only elements from A. After the algorithm stops, we check the output answer $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_k)$ and reset γ_i to NIL if γ_i has density lower than $-\hat{a}$ for all $i = 1, 2, \ldots, k$.

Lemma 2.2: Let $B_p = A[2pl+1, 2pl+4l]$ for $p = 0, \ldots, \frac{n}{2l} - 2$. After the i^{th} iteration of line 8 of Algorithm SolveDMDS, for all p, either N_p.seg is a maximum-density segment of length between l and $2l-1$ in B_p not overlapping any segment in $\{\gamma_1, \gamma_2, \ldots, \gamma_i\}$ or N_p.seg is NIL if all segments of length between l and $2l-1$ in B_p overlap some segment in $\{\gamma_1, \gamma_2, \ldots, \gamma_i\}$.

Proof: It is clear that the claim is true if we replace line 12 of our algorithm by "**for** $p \leftarrow 0$ to $\frac{n}{2l} - 2$ **do**." Note that N_j.seg can not overlap N_p.seq if $|p - j| > 1$. Thus, we only have to update N_{j-1}.seq, N_j.seq and N_{j+1}.seq after γ_i is assigned N_j.seq. □

Theorem 2.1: Algorithm SolveDMDS correctly solves the DMDS problem in $O(n + k \log k)$ time.

Proof: Let $B_p = A[2pl+1, 2pl+4l]$ for $p = 0, \ldots, \frac{n}{2l} - 2$. Because any segment of length between l and $2l-1$ must be a subsegment of some B_p, Lemma 2.2 ensures the correctness.

Now let us look at the time complexity. By using the linear time algorithm stated in [25], each execution of line 3 can be done in $O(l)$ time. Since all elements in $\gamma_1 \cup \gamma_2 \cup \cdots \cup \gamma_i$ have been marked as "deleted" before the execution of line 14 in the i^{th} iteration of line 8, the execution of line 14 can be done in $O(l)$ time by running the linear time algorithm stated in [25] on $A[2pl+1, 2pl+4l]$ and taking each deleted element in $A[2pl+1, 2pl+4l]$ as $-\infty$. Since line 3 is executed at most $\frac{n}{2l} - 1$ times and line 14 is executed at most $3k$ times, the time spent on line 3 and line 14 is at most $O(n) + O(kl) = O(n)$. It is well known that an n-element

Algorithm SolveDMDS

Input: A sequence A of n real numbers and two positive integers k and l.

Output: A sequence $\Gamma = (\gamma_1, \ldots, \gamma_k)$.

```
1  Create n/(2l) − 1 nodes N_0, N_1, ..., N_{n/(2l)−2}.
2  for i ← 0 to n/(2l) − 2 do
3      N_i.seg ← maximum-density segment of length between l and 2l − 1
4          in A[2il + 1, 2il + 4l].
5      N_i.value ← density of N_i.seg.
6  end for
7  Build a binary max-heap H consisting of N_0, N_1, ..., N_{n/(2l)−2}.
8  for i ← 1 to k do
9      Reset the value of variable j such that N_j = H.max().
10     γ_i ← N_j.seg.
11     Mark each element in γ_i as being deleted.
12     for p ← j − 1 to j + 1 do
13         H.Delete(N_p).
14         N_p.seg ← maximum-density segment of length between l and 2l − 1
15             in A[2pl + 1, 2pl + 4l] not overlapping any segment in {γ_1, ..., γ_i}.
16         N_p.value ← density of N_p.seg; /*Set N_p.value to −∞ if N_p.seg = NIL.*/
17         H.Insert(N_p).
18     end for
19 end for
```

Figure 2.1: The $O(n + k \log k)$-time algorithm for the DMDS problem.

binary max-heap can be built in $O(n)$ time, so the execution of line 7 takes $O(\frac{n}{l})$ time. Each execution of lines 9, 13 and 17 takes only $O(\log n)$ time because the size of H is at most $\frac{n}{2l} - 1$. The time spent on lines 9, 13 and 17 is $O(k \log n)$ since each of them is executed at most $3k$

times. Thus, the total time we need is $O(n + k \log n)$. Note that

$$n + k \log n \leq \begin{cases} 2n = O(n + k \log k) & \text{if } k \leq \frac{n}{\log n}; \\ n + k \log(k \log n) = O(n + k \log k^2) = O(n + k \log k) & \text{if } k > \frac{n}{\log n}. \end{cases}$$

It follows that $n + k \log n = O(n + k \log k)$, so our algorithm runs in $O(n + k \log k)$ time. □

The theorem below states that the DMDS problem has a lower bound of $\Omega(n + k \log k)$. Thus, Algorithm SolveDMDS is optimal.

Theorem 2.2: Any algorithm needs $\Omega(n + k \log k)$ time to solve the DMDS problem.

Proof: Let $T(n, l, k)$ be the time complexity of an algorithm for the DMDS problem. It is clear that any algorithm for the DMDS problem has to see the input sequence once, so we only have to show $T(n, l, k) = \Omega(k \log k)$. We define the SDMDS (Simplified DMDS) problem as follows. The input of the SDMDS problem is composed of a sequence $A = (\overbrace{v_1, v_1 \ldots, v_1}^{l}, \overbrace{v_2, v_2 \ldots, v_2}^{l}, \ldots, \overbrace{v_k, v_k \ldots, v_k}^{l}, \overbrace{v_{k+1}, v_{k+1} \ldots, v_{k+1}}^{n-kl})$ of n real numbers and two positive numbers k and l with $kl \leq n$, where $v_i \neq v_j$ if $i \neq j$ and $v_{k+1} < v_i$ for all $i = 1, 2, \ldots, k$. The goal of the SDMDS problem is to sort the k segments $\{A[1, l], A[l+1, 2l], \ldots, A[(k-1)l+1, kl]\}$ into decreasing order of density. Let $T'(n, l, k)$ be the time complexity of an algorithm for the SDMDS problem. Then $T'(n, l, k)$ is $\Omega(k \log k)$.

Let $(\gamma_1, \gamma_2, \ldots, \gamma_k)$ and $(\gamma'_1, \gamma'_2, \ldots, \gamma'_k)$ be a SDMDS solution and a DMDS solution to the same instance (A, l, k), respectively. We prove that $(\gamma_1, \gamma_2, \ldots, \gamma_k) = (\gamma'_1, \gamma'_2, \ldots, \gamma'_k)$. It follows that any algorithm for the DMDS problem is an algorithm for the SDMDS problem, so $T(n, l, k)$ is $\Omega(k \log k)$. Suppose for contradiction that i is the smallest index such that $\gamma_i \neq \gamma'_i$. By specification of the DMDS problem, γ'_i is the maximum-density segment of length between l and $2l - 1$ not overlapping any of $\{\gamma_1, \gamma_2, \ldots, \gamma_{i-1}\}$. Since γ_i is of length l and does not overlap any of $\{\gamma_1, \gamma_2, \ldots, \gamma_{i-1}\}$, it follows that $d(\gamma_i) \leq d(\gamma'_i)$. Because $\gamma_i \neq \gamma'_i$ and γ'_i cannot contain any element in $\bigcup_{j<i} \gamma_j$, some elements of γ'_i must be from $\bigcup_{j>i} \gamma_j$. Since all elements in $\bigcup_{j>i} \gamma_j$ are smaller than elements in γ_i, it follows that $d(\gamma_i) > d(\gamma'_i)$, a contradiction. □

2.3 Preprocessing

In this section, we show how to compress an input sequence A of the DSMSD problem of length $n \geq 10kl$ into a sequence A' of length $O(kl)$ in $O(n + k \log k)$ time. First we run SolveDMDS$(A, 3k, l)$ to find a sequence $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_{3k})$ of $3k$ disjoint segments of length between l and $2l - 1$ such that for all i, γ_i is either a maximum-density segment of length between l and $2l - 1$ not overlapping any of the first $i - 1$ segments in Γ or NIL if all segments of length between l and $2l - 1$ overlap some of the first $i - 1$ segments of Γ. Since $n \geq 10kl$, we have $\gamma_i \neq$ NIL for each $i = 1, \ldots, 3k$. Let $\gamma_i = A[p_i, q_i]$ for all i. We extend each segment γ_i to get $\gamma'_i = A[p'_i, q'_i]$, where

$$p'_i = \begin{cases} p_i - 2l + 1 & \text{if } p_i \geq 2l, \\ 1 & \text{if } p_i < 2l, \end{cases} \quad \text{and} \quad q'_i = \begin{cases} q_i + 2l - 1 & \text{if } q_i \leq n - 2l + 1, \\ n & \text{if } q_i > n - 2l + 1. \end{cases}$$

We say that $A[p, q]$ is a segment consisting of only elements in $\bigcup_{i=1}^{3k} \gamma'_i$ if and only if for each index $j \in [p, q]$, there exists a $\gamma'_i = A[p'_i, q'_i]$ such that $j \in [p'_i, q'_i]$. A segment $A[p, q]$ consisting of only elements in $\bigcup_{i=1}^{3k} \gamma'_i$ is maximal if and only if $A[p, q]$ is not a subsegment of any other segment consisting of only elements in $\bigcup_{i=1}^{3k} \gamma'_i$. Note that any two different maximal segments consisting of only elements in $\bigcup_{i=1}^{3k} \gamma'_i$ must be disjoint according to our definition. Let $R = (r_1, r_2, \ldots, r_{|R|})$ be all of the maximal segments, in left-to-right order, consisting of only elements in $\bigcup_{i=1}^{3k} \gamma'_i$. We set A' to $r_1 \cdot (-\infty) \cdot r_2 \cdot (-\infty) \cdot r_3 \cdots (-\infty) \cdot r_{|R|}$, where the symbol "$\cdot$" means concatenation. Since $\sum_{1 \leq i \leq |R|} |r_i| \leq \sum_{1 \leq i \leq 3k} |\gamma'_i| < \sum_{1 \leq i \leq 3k} 6l = 18kl$, A' is of length $\sum_{1 \leq i \leq |R|} |r_i| + |R| - 1 < 18kl + 3k - 1 < 21kl$.

The correctness follows from the next lemma which ensures that it is safe to delete elements not in any segments of R.

Lemma 2.3: There exists a solution $S = \{s_1, s_2, \ldots, s_k\}$ to the DSMSD problem instance (A, k, l) such that each segment in S is a subsegment of some segment of R.

Proof: First we show that there exists a solution $S = \{s_1, s_2, \ldots, s_k\}$ to the DSMSD problem instance (A, k, l) such that s_i overlaps some segment of Γ for $i = 1, \ldots, k$. Let $S' = \{s'_1, s'_2, \ldots, s'_k\}$ be a solution with fewest segments in it not overlapping any segments of Γ. Let s'_i be a segment not overlapping any segment of Γ. Since each segment in S' has length shorter than $2l$ and

each segment of Γ has length at least l, each segment in S' can overlap at most three segments of Γ. It follows that at most $3(k-1)$ segments of Γ are overlapped with some segment in S'. Since Γ is composed of $3k$ segments, there exists some γ_j not overlapping any segment in S'. By the specification of Γ, we know $d(\gamma_j) \geq d(s'_i)$. Thus, $(S'/\{s'_i\}) \cup \{\gamma_j\}$ is a solution with fewer segments in it not overlapping any segment of Γ, a contradiction.

It remains to prove that each segment in S is a subsegment of some segment of R. Let s_i be overlapped with γ_{j_i} for $i = 1, 2, \ldots, k$. Since each s_i is of length shorter than $2l$, s_i must be a subsegment of γ'_{j_i} for $i = 1, 2, \ldots, k$. Since each s_i is a subsegment of γ'_{j_i} and each γ'_{j_i} is a subsegment of some segment of R, each s_i is a subsegment of some segment of R. □

Now we analyze the time complexity of our preprocessing.

Lemma 2.4: It takes $O(n + k \log k)$ time to compute A'.

Proof: By Theorem 2.1, Γ can be computed in $O(n + k \log k)$ time. Let $\Gamma' = (\gamma'_1, \ldots, \gamma'_{3k})$. It is clear that Γ' can be computed in $O(kl)$ time and R can be computed in $O(n)$ time. Since $\sum_{1 \leq i \leq |R|} |r_i| \leq \sum_{1 \leq i \leq 3k} |\gamma'_i| < \sum_{1 \leq i \leq 3k} 6l = 18kl$, A' is of length $O(kl)$ and can be computed in $O(kl)$ time. The total complexity is therefore $O(n + kl + k \log k) = O(n + k \log k)$ time. □

The next theorem summarizes the results of this section.

Theorem 2.3: Given a DSMSD problem instance $(A[1 \ldots n], k, l)$ with $n \geq 10kl$, we can reduce it to a new DSMSD problem instance (A', k, l) in $O(n + k \log k)$ time, where A' is of length less than $21kl$.

Proof: Immediate from Lemmas 2.3 and 2.4. □

2.4 The Main Algorithm

In the following, we describe an $O(nk \log l)$-time algorithm for finding a solution to the DSMSD problem instance $(A[1 \ldots n], k, l)$.

Algorithm MAIN$(A[1 \ldots n], k, l)$

1 **for** $j \leftarrow 1$ to k **do**

2 Compute $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ for all $i \in \{1, 2, \ldots, \frac{n}{l}\}$.

3 Run DC$(tl, (t+1)l, j, Pos_j[tl], Pos_j[(t+1)l])$ for all $0 \leq t \leq \frac{n}{l} - 1$.

4 **end for**

5 Compute the solution with the help of $\bigcup_{j=1}^{k}\{Seg_j[1 \ldots n]\}$ and $\bigcup_{j=1}^{k}\{Pos_j[1 \ldots n]\}$.

Figure 2.2: A sketch of the main algorithm for the DSMSD problem.

Definition 2.1: Let $Sol_{i,j}$ be a solution to the DSMSD problem instance $(A[1 \ldots i], j, l)$ such that the position of the leftmost element of the rightmost segment in $Sol_{i,j}$ is maximized. Define $D_j[i]$ to be the sum of densities of segments in $Sol_{i,j}$, $Seg_j[i]$ to be the rightmost segment in $Sol_{i,j}$, and $Pos_j[i]$ to be the position of the leftmost element of $Seg_j[i]$.

Let DC$(p, q, j, Pos_j[p], Pos_j[q])$ be a procedure for computing $D_j[i]$, $Seg_j[i]$, and $Pos_j[i]$ for all $i \in \{p+1, p+2, \ldots, q-1\}$, where $q - p \leq l$. A sketch of our main algorithm is given in Figure 2.2. For simplicity, we assume that n is a multiple of l. Otherwise let $\hat{a} = \max\{|a_1|, |a_2|, \ldots, |a_n|\} + 1$ and $E' = -4kl \cdot \hat{a}$. We then append $\overbrace{(E', E', \ldots, E')}^{(l \cdot \lceil \frac{n}{l} \rceil - n)}$ to the end of the sequence A. Note that for all k disjoint segments (with lengths between l and $2l-1$) of the extended sequence, their sum of densities is greater than or equal to $-k \cdot \hat{a}$ if and only if the k disjoint segments are composed of only elements from A. It follows that the solution doesn't change with the extension.

We now explain the algorithm in detail.

Lemma 2.5: After $Seg_j[1 \ldots n]$ and $Pos_j[1 \ldots n]$ are known for $j = 1, \ldots, k$, a solution to the DSMSD problem instance $(A[1 \ldots n], k, l)$ can be found in $O(k)$ time.

Proof: Suppose now Seg_j and Pos_j are known for $j = 1, \ldots, k$. We describe an $O(k)$-time procedure for finding a solution to the DSMSD problem instance $(A[1 \ldots n], k, l)$ as follows.

1. Initiate i with n and Y with $\{\}$.
2. **for** $j \leftarrow k$ to 1 **do**
   (a) $Y \leftarrow Y \cup \{Seg_j[i]\}$.
   (b) $i \leftarrow Pos_j[i] - 1$.
   end for
3. **return** Y.

□

By Lemma 2.5, the challenge now lies in computing D_j, Seg_j, and Pos_j for all j in $[1, k]$ in $O(nk \log l)$ time. The computation consists of k iterations. In the j^{th} iteration, D_j, Seg_j, and Pos_j are computed in $O(n \log l)$ time. In the following, we describe how to compute D_j, Seg_j, and Pos_j in $O(n \log l)$ time for each j by utilizing Lemma 2.6 below and the Chung-Lu algorithm [25]. Note that given $g < h$, Definition 1 does not eliminate the possibilities where $Seg_j[g]$ is a proper subsegment of $Seg_j[h]$. Therefore, Lemma 2.6 is not a trivial consequence of Definition 1.

Lemma 2.6: $Pos_j[1] \leq Pos_j[2] \leq \cdots \leq Pos_j[n-1] \leq Pos_j[n]$ for $j = 1, \ldots, k$.

Proof: Suppose not. Let $p < q$ and $Pos_j[p] > Pos_j[q]$. Let $Sol_{p,j} = \{s_1, s_2, \ldots, s_j\}$ be a solution to the DSMSD problem instance $(A[1, p], j, l)$, in left-to-right order, such that the position of the leftmost element of s_j is $Pos_j[p]$. Let $Sol_{q,j} = \{s'_1, s'_2, \ldots, s'_j\}$ be a solution to the DSMSD problem instance $(A[1, q], j, l)$, in left-to-right order, such that the position of the leftmost element of s'_j is $Pos_j[q]$. We choose $Sol_{q,j}$ so that the position of the rightmost element of s'_j is minimized. Let $s_j = A[l_1, r_1]$ and $s'_j = A[l_2, r_2]$. It follows that r_1 is less than r_2; otherwise, we can either improve $D_j[p]$ (in case $D_j[q] > D_j[p]$) or increase the value of $Pos_j[q]$ from l_2 to l_1 (in case $D_j[p] \geq D_j[q]$), where both lead to contradiction to the optimality of $Sol_{p,j}$ or maximality of $Pos_j[q]$. Let $L = A[l_2, l_1 - 1]$ and $R = A[r_1 + 1, r_2]$. By $r_1 < r_2$, we have $d(R) > d(L \cup s_j)$; otherwise, we can replace s'_j with $A[l_2, r_1]$ and obtain an alternative solution for $Sol_{q,j}$, but it contradicts that the position of the rightmost element of s'_j is minimized.

Suppose for contradiction that $d(L) < d(s_j)$. By $d(R) > d(L \cup s_j)$, we have $d(R) > d(L \cup s_j) > d(L)$. It follows that $d(s_j \cup R) > d(L)$, so $d(s_j \cup R) > d(L \cup s_j \cup R) = d(s'_j)$, a

contradiction. Thus, we have $d(L) \geq d(s_j)$. By $d(R) > d(L \cup s_j)$ and $d(L) \geq d(s_j)$, we have $d(R) > d(L \cup s_j) \geq d(s_j)$. Let $\Delta_1 = d((L \cup s_j) \cup R) - d(L \cup s_j)$ and $\Delta_2 = d(s_j \cup R) - d(s_j)$. By simple algebra, we can see that $\Delta_1 = \frac{|R| \cdot (d(R) - d(L \cup s_j))}{|L \cup s_j \cup R|}$ and $\Delta_2 = \frac{|R| \cdot (d(R) - d(s_j))}{|s_j \cup R|}$. Since $d(R) > d(L \cup s_j) \geq d(s_j)$ and $|L \cup s_j \cup R| > |s_j \cup R|$, it follows that Δ_1 must be less than Δ_2, i.e.,

$$d(s'_j) - d(L \cup s_j) < d(s_j \cup R) - d(s_j). \tag{2.1}$$

Since $Sol_{p,j}$ is a solution to the DSMSD problem instance $(A[1,p], j, l)$ and $\{s'_1, \ldots, s'_{j-1}, L \cup s_j\}$ is a set of j disjoint segments of $A[1,p]$ of length between l and $2l-1$, we have

$$\sum_{1 \leq i \leq j-1} d(s'_i) + d(L \cup s_j) \leq \sum_{1 \leq i \leq j} d(s_i). \tag{2.2}$$

By (1) and (2), we have $\sum_{1 \leq i \leq j-1} d(s'_i) + d(s'_j) < \sum_{1 \leq i \leq j-1} d(s_i) + d(s_j \cup R)$. It follows that $Sol_{q,j}$ is not a solution to the DSMSD problem instance $(A[1,q], j, l)$, a contradiction. □

The Chung-Lu Algorithm [25]. *Given a sequence A of n number pairs (v_i, w_i) with $w_i > 0$ and two positive numbers $l \leq u$, the Chung-Lu algorithm can find a maximum-density segment of A with length between l and u in an online manner in $O(n)$ time. An algorithm is said to run in an online manner if and only if it can process its input piece-by-piece and maintain a solution for the pieces processed so far.*

Note that the data structures used by the Chung-Lu algorithm [25] are simple and concise, and can be implemented in a reasonable amount of efforts. As a matter of fact, Chung and Lu [25] solved a generalized version of the maximum-density problem considered previously by [53]. For this generalized version, the only rival is the algorithm proposed in [41]. We believe that the Chung-Lu algorithm would speed up in practice.

For technical reasons, we define $D_j[i]$, $Seg_j[i]$, and $Pos_j[i]$ to be $-\infty$, NIL, and 0, respectively if $i < jl$. We first compute $D_j[tl]$, $Seg_j[tl]$, and $Pos_j[tl]$ for all $t \in \{1, 2, \ldots, \frac{n}{l}\}$ and then for all intermediate positions $i \in \bigcup_{0 \leq t \leq \frac{n}{l}-1} \{tl+1, tl+2, \ldots, (t+1)l-1\}$. The reason for this is that for all intermediate positions i with $tl < i < (t+1)l$, according to Lemma 2.6, we have $Pos_j[tl] \leq Pos_j[i] \leq Pos_j[(t+1)l]$. Therefore, if we first compute $Pos_j[tl]$ and $Pos_j[(t+1)l]$, then the following computation of $Pos_j[i]$ for $i \in \{tl+1, tl+2, \ldots, (t+1)l-1\}$ becomes faster because values lower than $Pos_j[tl]$ or greater than $Pos_j[(t+1)l]$ are not considered.

Lemma 2.7: If we already know $D_{j-1}[1 \ldots n]$, $Seg_{j-1}[1 \ldots n]$, and $Pos_{j-1}[1 \ldots n]$, then computing $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ for all $i \in \{1, 2, \ldots, \frac{n}{l}\}$ can be done in $O(n)$ time.

Proof: The pseudo code is given in Figure 2.3. The procedure consists of $\frac{n}{l}$ iterations. In the i^{th} iteration, we compute $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$. We prove that each iteration takes at most $O(l)$ time, so the total time is $O(n)$. For each $i \in \{1, 2, \ldots, j-1\}$, the i^{th} iteration can be done in $O(1)$ time by setting $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ to $-\infty$, NIL, and 0, respectively. The j^{th} iteration can also be completed in $O(1)$ time by setting $D_j[jl]$, $Seg_j[jl]$, and $Pos_j[jl]$ to $D_{j-1}[(j-1)l] + d(A[(j-1)l+1, jl])$, $A[(j-1)l+1, jl]$, and $(j-1)l+1$, respectively. Suppose now we are in the i^{th} iteration, where $i > j$. We first find the maximum-density segments s_t of $A[t, il]$ with length between l and $2l-1$ for all $t \in \{(i-1)l - 2l + 1, (i-1)l - 2l + 2, \ldots, il\}$ in $O(l)$ time by taking $A[t]$ as a number pair $(A[t], 1)$ and using the Chung-Lu algorithm to process A from position il to position $(i-1)l - 2l + 1$.

Let t' be the largest t such that $D_{j-1}[t-1] + d(s_t)$ is maximized. Then there are two cases to consider. Case 1: $Pos_j[i] \leq (i-1)l - 2l$. In this case, it is clear that $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ are equal to $D_j[(i-1)l]$, $Seg_j[(i-1)l]$, and $Pos_j[(i-1)l]$, respectively. Case 2: $Pos_j[i] \geq (i-1)l - 2l + 1$. In this case, $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ are set to $D_{j-1}[t'-1] + s_{t'}$, $s_{t'}$, and t', respectively. We distinguish between Case 1 and Case 2 by comparing $D_j[(i-1)l]$ with $D_{j-1}[t'-1] + s_{t'}$. If $D_j[(i-1)l] > D_{j-1}[t'-1] + s_{t'}$, then it is Case 1; otherwise it is Case 2. □

Now we begin to describe how to compute $D_j[i]$, $Seg_j[i]$, and $Pos_j[i]$ for all $i \in \bigcup_{0 \leq t \leq \frac{n}{l}-1} \{tl + 1, tl + 2, \ldots, (t+1)l - 1\}$ in $O(n \log l)$ time.

Lemma 2.8: If the procedure $\mathrm{DC}(p, q, j, Pos_j[p], Pos_j[q])$ can be implemented to run in $O((m' + n') \log n')$ time, where $n' = q - p + 1$ and $m' = Pos_j[q] - Pos_j[p] + 1$, then $D_j[i]$, $Seg_j[i]$, and $Pos_j[i]$ for all $i \in \bigcup_{0 \leq t \leq \frac{n}{l}-1} \{tl + 1, tl + 2, \ldots, (t+1)l - 1\}$ can be computed in $O(n \log l)$ time.

Proof: Let $m_t = Pos_j[(t+1)l] - Pos_j[tl] + 1$ for $t = 0, 1, \ldots, \frac{n}{l} - 1$. $\sum_{0 \leq t \leq \frac{n}{l}-1} m_t = \sum_{0 \leq t \leq \frac{n}{l}-1} (Pos_j[(t+1)l] - Pos_j[tl] + 1) = Pos_j[n] - Pos_j[0] + \frac{n}{l} = O(n)$. Since $(t+1)l - tl = l$, $D_j[i]$, $Seg_j[i]$, and $Pos_j[i]$ for all $i \in \{tl + 1, tl + 2, \ldots, (t+1)l - 1\}$ can be computed in $O((m_t + l) \log l)$ time by calling $\mathrm{DC}(tl, (t+1)l, j, Pos_j[tl], Pos_j[(t+1)l])$. The total complexity

for computing $D_j[i]$, $Seg_j[i]$, and $Pos_j[i]$ for all $i \in \bigcup_{0 \le t \le \frac{n}{l}-1}\{tl+1, tl+2, \ldots, (t+1)l-1\}$ is therefore $O(\sum_{0 \le t \le \frac{n}{l}-1}(m_t + l)\log l) = O(n \log l)$. □

Procedure PREPARATION

Input: j.

Goal: Compute $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ for all $i \in \{1, 2, \ldots, \frac{n}{l}\}$.

(* $D_{j-1}[1 \ldots n]$, $Seg_{j-1}[1 \ldots n]$, and $Pos_{j-1}[1 \ldots n]$ are known. *)

for $i \leftarrow 1$ to $\frac{n}{l}$ **do**
  if $i < j$ **then**
    $D_j[il] \leftarrow -\infty$, $Seg_j[il] \leftarrow$ NIL, and $Pos_j[il] \leftarrow 0$.
  else if $i = j$ **then**
    $D_j[jl] \leftarrow D_{j-1}[(j-1)l] + d(A[(j-1)l+1, jl])$.
    $Seg_j[jl] \leftarrow A[(j-1)l+1, jl]$.
    $Pos_j[jl] \leftarrow (j-1)l+1$.
  else
    Find the maximum-density segments s_t of $A[t, il]$ with length between l and $2l-1$
      for all $t \in \{(i-1)l - 2l + 1, (i-1)l - 2l + 2, \ldots, il\}$.
    $t' \leftarrow$ the largest t such that $D_{j-1}[t-1] + d(s_t)$ is maximized.
    if $D_j[(i-1)l] > D_{j-1}[t'-1] + s_{t'}$ **then**
      $D_j[il] \leftarrow D_j[(i-1)l]$, $Seg_j[il] \leftarrow Seg_j[(i-1)l]$, and $Pos_j[il] \leftarrow Pos_j[(i-1)l]$.
    else
      $D_j[il] \leftarrow D_{j-1}[t'-1] + s_{t'}$, $Seg_j[il] \leftarrow s_{t'}$, and $Pos_j[il] \leftarrow t'$.
end for

Figure 2.3: Computing $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ for all $i \in \{1, 2, \ldots, \frac{n}{l}\}$.

Lemma 2.9: If we already know $D_{j-1}[1 \ldots n]$, $Seg_{j-1}[1 \ldots n]$, and $Pos_{j-1}[1 \ldots n]$, then the procedure DC$(p, q, j, Pos_j[p], Pos_j[q])$ can be implemented to run in $O((m' + n')\log n')$ time, where $n' = q - p + 1$ and $m' = Pos_j[q] - Pos_j[p] + 1$.

Proof: The pseudo code is given in Figure 2.4. Let $c = \lfloor (p+q)/2 \rfloor$. We first compute $D_j[c]$, $Seg_j[c]$, and $Pos_j[c]$ in $O(n'+m')$ time (to be proved below) and then recursively call DC$(p, c, j, Pos_j[p], Pos_j[c])$ and DC$(c, q, j, Pos_j[c], Pos_j[q])$. Let $T(n', m')$ be the running time of DC$(p, q, j, Pos_j[p], Pos_j[q])$. Since $Pos_j[c]$ is between $Pos_j[p]$ and $Pos_j[q]$ by Lemma 2.6, we have $T(n', m') \leq T(\lfloor \frac{n'}{2} \rfloor - 1, x) + T(\lceil \frac{n'}{2} \rceil - 1, m' - x + 1)$ for some integer x in $[1, m']$. It follows that $T(n', m') = O((m'+n') \log n')$.

It remains to explain how to compute $D_j[c]$, $Seg_j[c]$, and $Pos_j[c]$ in $O(n'+m')$ time. Note that since $q - p \leq l$, we have $Pos_j[q] \leq p+1$. If $c < jl$, then we directly set $D_j[c]$, $Seg_j[c]$, and $Pos_j[c]$ to $-\infty$, NIL, and 0, respectively. Otherwise, we do the following. First we compute the maximum-density segments s_t of $A[t, c]$ of length between l and $2l-1$ with the position of the rightmost element not in $(Pos_j[q], p+1)$ for all $t \in \{Pos_j[p], Pos_j[p]+1, \ldots, Pos_j[q]\}$ in $O(n'+m')$ time by taking $A[i]$ as a number pair $(A[i], 1)$ for i not in $[Pos_j[q]+1, p+1]$ and the whole segment $A[Pos_j[q]+1, p+1]$ as a number pair $(\sum_{Pos_j[q]+1 \leq i \leq p+1} A[i], p - Pos_j[q] + 1)$ and using the Chung-Lu algorithm to process $A[Pos_j[p], c]$ from right to left.

Let t' be the largest t such that $D_{j-1}[t-1] + d(s_t)$ is maximized. Then there are two cases to consider. Case 1: $D_j[p] > D_{j-1}[t'-1] + d(s_{t'})$ or both $D_j[p] = D_{j-1}[t'-1] + d(s_{t'})$ and $t' = Pos_j[p]$. In this case, $D_j[c]$, $Seg_j[c]$, and $Pos_j[c]$ are set to $D_j[p]$, $Seg_j[p]$, and $Pos_j[p]$, respectively. Case 2: $D_j[p] < D_{j-1}[t'-1] + d(s_{t'})$ or both $D_j[p] = D_{j-1}[t'-1] + d(s_{t'})$ and $t' > Pos_j[p]$. In this case, $D_j[c]$, $Seg_j[c]$, and $Pos_j[c]$ are set to $D_{j-1}[t'-1] + d(s_{t'})$, $s_{t'}$, and t', respectively. □

The next theorem summarizes the results of this section.

Theorem 2.4: Given a DSMSD problem instance $(A[1 \ldots n], k, l)$, we can find a solution in $O(nk \log l)$ time.

Proof: By Lemmas 2.7, 2.8, and 2.9, computing D_j, Seg_j, and Pos_j can be done in $O(n \log l)$ time in the j^{th} iteration for $j = 1, \ldots, k$. Thus, computing D_j, Seg_j, and Pos_j for all j in $[1, k]$ can be done in $O(nk \log l)$ time. After Seg_j and Pos_j are found for $j = 1, \ldots, k$, a solution to the DSMSD problem instance $(A[1 \ldots n], n, k)$ can be found in $O(k)$ time by Lemma 2.5. □

Procedure DC

Input: $p, q, j, Pos_j[p]$, and $Pos_j[q]$, where $q - p \leq l$.

Goal: Compute $D_j[i]$, $Seg_j[i]$, and $Pos_j[i]$ for all $i \in \{p+1, p+2, \ldots, q-1\}$.

(*$D_{j-1}[1 \ldots n]$, $Seg_{j-1}[1 \ldots n]$, and $Pos_{j-1}[1 \ldots n]$ are known*)

1 $c \leftarrow \lfloor (p+q)/2 \rfloor$.
2 **if** $c < jl$ **then**
3 $D_j[c] \leftarrow -\infty$, $Seg_j[c] \leftarrow$ NIL, and $Pos_j[c] \leftarrow 0$.
4 **else**
5 Find the maximum-density segments s_t of $A[t, c]$ of length between l and $2l-1$
6 with the position of the rightmost element not in $(Pos_j[q], p+1)$
7 for all $t \in \{Pos_j[p], Pos_j[p]+1, \ldots, Pos_j[q]\}$.
8 $t' \leftarrow$ the largest t such that $D_{j-1}[t-1] + d(s_t)$ is maximized.
12 **if** $D_j[p] > D_{j-1}[t'-1] + d(s_{t'})$ **then**
13 $D_j[c] \leftarrow D_j[p]$, $Seg_j[c] \leftarrow Seg_j[p]$, and $Pos_j[c] \leftarrow Pos_j[p]$.
14 **else if** $D_j[p] = D_{j-1}[t'-1] + d(s_{t'})$ and $t' = Pos_j[p]$ **then**
16 $D_j[c] \leftarrow D_j[p]$, $Seg_j[c] \leftarrow Seg_j[p]$, and $Pos_j[c] \leftarrow Pos_j[p]$.
17 **else**
18 $D_j[c] \leftarrow D_{j-1}[t'-1] + d(s_{t'})$, $Seg_j[c] \leftarrow s_{t'}$, and $Pos_j[c] \leftarrow t'$.
19 **if** $c - p > 2$ **then**
20 Call DC$(p, c, j, Pos_j[p], Pos_j[c])$.
21 **if** $q - c > 2$ **then**
22 Call DC$(c, q, j, Pos_j[c], Pos_j[q])$.
23 **return**

Figure 2.4: Computing $D_j[il]$, $Seg_j[il]$, and $Pos_j[il]$ for all $i \in \{p+1, p+2, \ldots, q-1\}$.

2.5 The Complete Algorithm

We now explain how to find a solution to the DSMSD problem instance $(A[1 \ldots n], k, l)$ in $O(n + k^2 l \log l)$ time. If $n < 10kl$, by Theorem 2.4, we can find a solution in $O(nk \log l) =$

$O(10k^2l \log l) = O(n + k^2l \log l)$ time. Otherwise, by Theorem 2.3, we can first reduce the original DSMSD problem instance $(A[1 \ldots n], k, l)$ to a new DSMSD problem instance (A', k, l) in $O(n + k \log k)$ time, where A' is of length less than $21kl$. Then, by Theorem 2.4, a solution to the problem instance (A', k, l) can be found in $O(21k^2l \log l) = O(k^2l \log l)$ time. The total time is $O(n + k \log k + k^2l \log l) = O(n + k^2l \log l)$.

Theorem 2.5: Given a DSMSD problem instance $(A[1 \ldots n], k, l)$, we can find a solution in $O(n + k^2l \log l)$ time.

2.6 Notes

In this chapter we study the problem of finding k disjoint segments, each of length at least l, such that the sum of their densities is maximized and give an $O(n + k^2l \log l)$-time algorithm for it. To the best of our knowledge, there is no nontrivial lower bound proved so far for this problem. Thus there is still a large gap between the trivial lower bound of $O(n)$ and the upper bound of $O(n + k^2l \log l)$. Bridging this gap remains an open problem. Moreover, it is also interesting to see whether our techniques could accelerate the algorithms for the other problems considered in [14]. Finally, none of the methods for this problem, including ours, has yet been implemented. It might just as well carry out some empirical study for comparing those available methods in practice.

Chapter 3

Length-Constrained Max-Eccentricity Segments

Given a sequence of n real numbers $A = (a_1, a_2, \ldots, a_n)$ and an integers L, define the eccentricity of a segment $A[i, j]$ to be $ecc(i, j) = \frac{sum(i,j)}{\sqrt{length(i,j)}}$. The problem is to find a segment $A[i, j]$ maximizing the eccentricity $ecc(i, j)$ subject to $length(i, j) \geq L$. For the case where there is no length constraint, i.e. $L = 1$, Lipson *et al.* [56] proposed an approximation scheme. Given any $\epsilon \in (0, 1/5]$, their approximation scheme runs in $O(n\epsilon^{-2})$ time and outputs a segment $A[i, j]$ such that $ecc(i, j)$ is at least $\text{Opt}/\alpha(\epsilon)$, where Opt is the maximum possible eccentricity and $\alpha(\epsilon) = (1 - \sqrt{2\epsilon(2 + \epsilon)})^{-1}$. There was no known subquadratic time exact algorithm before even for the case where there is no length constraint. In this chapter, we propose an exact algorithm which copes with the general case and runs in $O(n\frac{T(L^{1/2})}{L^{1/2}})$ time, where $T(n')$ is the time required to solve the all-pairs shortest paths problem on a graph of n' nodes. By the latest result of Chan [19], $T(n') = O(n'^3 \frac{(\log\log n')^3}{(\log n')^2})$, so our algorithm runs in subquadratic time $O(nL\frac{(\log\log L)^3}{(\log L)^2})$. It should be noted that our result immediately implies an $O(n)$-time exact algorithm for the case where there is no length constraint.

3.1 Preliminaries

In the following, we review some definitions and theorems. For more details, readers can refer to [15, 16, 31].

Definition 3.1: A function $f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ is said to be *quasiconvex* if and only if for all points $u, v \in \mathbb{R}^+ \times \mathbb{R}$ and all $\lambda \in [0, 1]$, we have $f(\lambda \cdot u + (1 - \lambda) \cdot v) \leq \max\{f(u), f(v)\}$.

Lemma 3.1: [16] Define $f : \mathbb{R}^+ \times \mathbb{R}$ by letting

$$f(\ell, s) = \begin{cases} \frac{s}{\sqrt{\ell}} & \text{if } s \geq 0; \\ 0 & \text{otherwise.} \end{cases}$$

Then f is quasiconvex.

Theorem 3.1: [15] Given a sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$, a length lower bound L, and a quasiconvex score function $f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$, there exists an algorithm, denoted by MSS(A, L, f), which can find a segment $A[i, j]$ that maximizes $f(length(i, j), sum(i, j))$ subject to $length(i, j) \geq L$ in $O(n)$ time.

By the fact $f(\ell, s) = s$ is quasiconvex and Theorem 3.1, we have the following corollary, which was also proved in [31, 53].

Corollary 3.1: There exists an $O(n)$-time algorithm for finding a segment $A[i, j]$ maximizing $sum(i, j)$ subject to $length(i, j) \geq L$.

We next prove that if all segments with lengths $\geq L$ have negative sums, then the optimal solution must have length less than $2L$.

Lemma 3.2: If $sum(p, q) < 0$ holds for each segment $A[p, q]$ of length at least L, then $length(p^*, q^*) < 2L$, where $(p^*, q^*) = \arg \max_{length(p,q) \geq L} ecc(p, q)$.

Proof: Let $(p^*, q^*) = \arg \max_{length(p,q) \geq L} ecc(p, q)$. Suppose for contradiction that $length(p^*, q^*) \geq 2L$. Let $c_1 = \lfloor (p^*+q^*)/2 \rfloor$ and $c_2 = c_1+1$. Then we have $length(p^*, c_1) \geq L$ and $length(c_2, q^*) \geq L$. Since $\frac{sum(p^*,q^*)}{length(p^*,q^*)} = \frac{sum(p^*,c_1)+sum(c_2,q^*)}{length(p^*,c_1)+length(c_2,q^*)}$, we have

$$\frac{sum(p^*, q^*)}{length(p^*, q^*)} \leq \frac{sum(p^*, c_1)}{length(p^*, c_1)} \text{ or } \frac{sum(p^*, q^*)}{length(p^*, q^*)} \leq \frac{sum(c_2, q^*)}{length(c_2, q^*)}.$$

Without loss of generality, we assume $\frac{sum(p^*,q^*)}{length(p^*,q^*)} \leq \frac{sum(p^*,c_1)}{length(p^*,c_1)}$. Since $\frac{sum(p^*,q^*)}{length(p^*,q^*)} \leq \frac{sum(p^*,c_1)}{length(p^*,c_1)} < 0$ and $\sqrt{length(p^*, q^*)} > \sqrt{length(p^*, c_1)}$, we have

$$\frac{\sqrt{length(p^*,q^*)}\cdot sum(p^*,q^*)}{length(p^*,q^*)} < \frac{\sqrt{length(p^*,c_1)}\cdot sum(p^*,c_1)}{length(p^*,c_1)}$$
$$\Leftrightarrow \quad \frac{sum(p^*,q^*)}{\sqrt{length(p^*,q^*)}} < \frac{sum(p^*,c_1)}{\sqrt{length(p^*,c_1)}}$$
$$\Leftrightarrow \quad ecc(p*,q*) < ecc(p^*,c_1).$$

It contradicts $(p^*,q^*) = \arg\max_{length(p,q)\geq L} ecc(p,q)$. □

3.2 Subroutine

In the following we give a new algorithm for the Min-Plus Convolution Problem, which serves as a subroutine in our algorithm for the Length-Constrained Max-Eccentricity Segments problem. The min-plus convolution of two vectors $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$ is a vector $\mathbf{z} = (z_0, z_1, \ldots, z_{n-1})$ such that $z_k = \min_{i=0}^{k}\{x_i + y_{k-i}\}$ for $k = 0, 1, \ldots, n-1$. Given two vectors $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$, the Min-Plus Convolution Problem is to compute the min-plus convolution $\mathbf{z}$ of $\mathbf{x}$ and $\mathbf{y}$. In the following, we give an $O(n^{1/2}T(n^{1/2}))$-time algorithm for the Min-Plus Convolution Problem, where $T(n)$ is the time required to solve the all-pairs shortest paths problem on a graph of n nodes. To date, the best algorithm for computing the all-pairs shortest paths problem on a graph of n nodes runs in $O(n^3\frac{(\log\log n)^3}{(\log n)^2})$ time [19]. Thus, our work implies an $O(n^2\frac{(\log\log n)^3}{(\log n)^2})$-time algorithm for the min-plus convolution problem.

Definition 3.2: The min-plus product BC of a $d \times n'$ matrix $B = [b_{i,j}]$ and an $n' \times d$ matrix $C = [c_{i,j}]$ is a $d \times d$ matrix $D = [d_{i,j}]$ where $d_{i,j} = \min_{k=0}^{n'-1}\{b_{i,k} + c_{k,j}\}$.

Note that the notion of "min-plus product" is different from the notion of "min-plus convolution". It is well known [1] that the time complexity of computing the min-plus product of two $n' \times n'$ matrices is asymptotically equal to that of computing all pairs shortest paths for a graph with n' vertices. The next lemma was proved by Takaoka in [72]. The proof of the next lemma was also given in [72], and we include it here for completeness.

Lemma 3.3: [72] Given a $T(n')$-time algorithm for computing the min-plus product of any two $n' \times n'$ matrices, the computation of the min-plus product of B and C, where B is a $d \times n'$ matrix and C is an $n' \times d$ matrix, can be done in $O(\frac{n'}{d}T(d))$ time if $d \leq n'$.

Proof: For simplicity we assume that d divides n. We first split B into n'/d matrices $B_1, \ldots, B_{n'/d}$ of dimension $d \times d$ and C into n'/d matrices $C_1, \ldots, C_{n'/d}$ of dimension $d \times d$. Then we can compute $\{B_1C_1, B_2C_2, \ldots, B_{n'/d}C_{n'/d}\}$ in $O(dT(n'/d))$ time by the given algorithm. The (i,j)-th entry of the min-plus product of B and C is $\min_{k=1}^{n'/d}\{$the (i,j)-th entry of $B_kC_k\}$. □

Our new algorithm for the MIN-PLUS CONVOLUTION PROBLEM is as follows.

Algorithm MINPLUSCONVOLUTION

Input: $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$.

Output: $\mathbf{z} = (z_0, z_1, \ldots, z_{n-1})$ such that $z_k = \min_{i=0}^{k}\{x_i + y_{k-i}\}$ for $k = 0, 1, \ldots, n-1$.

1: Construct an $\lceil n^{1/2} \rceil \times (2n-1)$ matrix $B = [b_{i,j}]$ such that the i^{th} row of B
   is equal to $(\infty, \ldots, \infty, x_0, x_1, \ldots, x_{n-1}, \overbrace{\infty, \ldots, \infty}^{i \times \lceil n^{1/2} \rceil})$ for $i = 0, 1, \ldots, \lceil n^{1/2} \rceil - 1$.

2: Construct a $(2n-1) \times \lceil n^{1/2} \rceil$ matrix $C = [c_{i,j}]$ such that the transpose
   of the j^{th} column of C is equal to $(\overbrace{\infty, \ldots, \infty}^{j}, y_{n-1}, y_{n-2}, \ldots, y_0, \infty, \ldots, \infty)$
   for $j = 0, 1, \ldots, \lceil n^{1/2} \rceil - 1$.

3: Let $D = [d_{i,j}]$ be the min-plus product of B and C.

4: **for** $k \leftarrow 0$ to $n-1$ **do**
   Find i, j such that $k = i \times \lceil n^{1/2} \rceil + j$, where $0 \leq j < \lceil n^{1/2} \rceil$.
   Set z_k to $d_{i,j}$.
   end for

5: **return** $\mathbf{z} = (z_0, z_1, \ldots, z_{n-1})$.

The following lemma ensures the correctness.

Lemma 3.4: In MinPlusConvolution, $d_{i,j} = \min_{t=0}^{i\times\lceil n^{1/2}\rceil+j}\{x_t + y_{i\times\lceil n^{1/2}\rceil+j-t}\}$ if $0 \leq i \times \lceil n^{1/2}\rceil + j \leq n-1$.

Proof:

$$\begin{aligned}
d_{i,j} &= \min_{t=0}^{2n-1}\{b_{i,t} + c_{t,j}\} \\
&= \min\{\min_{t=0}^{n-2-i\times\lceil n^{1/2}\rceil}\{b_{i,t} + c_{t,j}\}, \min_{t=n-1-i\times\lceil n^{1/2}\rceil}^{n+j-1}\{b_{i,t} + c_{t,j}\}, \min_{t=n+j}^{2n-1}\{b_{i,t} + c_{t,j}\}\} \\
&= \min\{\min_{t=0}^{n-2-i\times\lceil n^{1/2}\rceil}\{\infty + c_{t,j}\}, \min_{t=n-1-i\times\lceil n^{1/2}\rceil}^{n+j-1}\{b_{i,t} + c_{t,j}\}, \min_{t=n+j}^{2n-1}\{b_{i,t} + \infty\}\} \\
&= \min_{t=n-1-i\times\lceil n^{1/2}\rceil}^{n+j-1}\{b_{i,t} + c_{t,j}\} \\
&= \min\{x_0 + y_{i\times\lceil n^{1/2}\rceil+j}, x_1 + y_{i\times\lceil n^{1/2}\rceil+j-1} + \cdots + x_{i\times\lceil n^{1/2}\rceil+j} + y_0\} \\
&= \min_{t=0}^{i\times\lceil n^{1/2}\rceil+j}\{x_t + y_{i\times\lceil n^{1/2}\rceil+j-t}\}
\end{aligned}$$

□

We now analyze the time complexity. Let $T(n)$ denote the time required to compute the min-plus product of two $n \times n$ matrices. Steps 1 and 2 take $O(n^{3/2})$ time, and by Lemma 3.3, Step 3 takes $O(\frac{2n-1}{n^{1/2}}T(\lceil n^{1/2}\rceil)) = O(n^{1/2}T(n^{1/2}))$ time. Steps 4 and 5 take $O(n)$ time. Therefore, the total running time is $O(n^{1/2}T(n^{1/2}) + n^{3/2})$. Since $T(n) = \Omega(n^2)$, we have $O(n^{1/2}T(n^{1/2}) + n^{3/2}) = O(n^{1/2}T(n^{1/2}))$. Theorem 3.2 summarizes our results for the Min-Plus Convolution Problem.

Theorem 3.2: The running time of Algorithm MinPlusConvolution is $O(n^{1/2}T(n^{1/2}))$, where $T(n)$ is the time required to compute the min-plus product of two $n \times n$ matrices.

The next Lemma was proved by Bergkvist and Damaschke in [14].

Lemma 3.5: [14] Given a sequence $A = (a_1, a_2, \ldots, a_n)$, the Maximum Consecutive Sums Problem is to compute a sequence $(w_1, w_2, \ldots, w_n)$ where $w_i = \max\{\sum_{j=p}^{q} a_j : length(p, q) = i\}$ for each $i = 1, 2, \ldots, n$. The Maximum Consecutive Sums Problem can be reduced to the Min-Plus Convolution Problem in linear time.

Corollary 3.2: Given a sequence $A = (a_1, a_2, \ldots, a_n)$, we can compute in $O(n^{1/2}T(n^{1/2}))$ time a sequence $(w_1, w_2, \ldots, w_n)$ such that $w_i = \max\{\sum_{j=p}^{q} a_j : \mathit{length}(p, q) = i\}$ for each $i = 1, 2, \ldots, n$ by making use of the Alogirthm MinPlusConvolution, where $T(n)$ is the time required to compute the min-plus product of two $n \times n$ matrices.

Proof: Immediately from Theorem 3.2 and Lemma 3.5. □

3.3 Algorithm

We next show how to solve the Length-Constrained Max-Eccentricity Segments problem in $O(n\frac{T(L^{1/2})}{L^{1/2}})$ time, where $T(n')$ is the time required to compute the min-plus product of two $n' \times n'$ matrices. To avoid notational overload, we assume that $4L$ divides n. By Corollary 3.1, we can find a segment $A[i, j]$ maximizing $\mathit{sum}(i, j)$ subject to $\mathit{length}(i, j) \geq L$ in linear time. Then there are three cases to consider: (1) $\mathit{sum}(i, j) = 0$; (2) $\mathit{sum}(i, j) > 0$; (3) $\mathit{sum}(i, j) < 0$. If it is Case 1, then $A[i, j]$ must be an optimal solution, and we are done. If it is Case 2, then we know there is at least one segment satisfying the length constraint with positive sum. Define $f(\ell, s)$ by

$$f(\ell, s) = \begin{cases} \frac{s}{\sqrt{\ell}} & \text{if } s \geq 0; \\ 0 & \text{otherwise,} \end{cases}$$

By Lemma 3.1, f is quasiconvex, so we can call MSS(A, L, f) to find the segment $A[i', j']$ maximizing $f(i', j')$ subject to $\mathit{length}(i', j') \geq L$ in linear time. Clearly, $A[i', j']$ is an optimal solution, and we are done. If it is Case 3, i.e., all segments satisfying the length constraint have negative sums, we do the following. First, by letting A_k be the segment $A[2kL + 1, 2kL + 4L]$ for each $k = 0, 1, \ldots, \frac{n}{2L} - 2$, we can divide the whole sequence $A[1, n]$ into $\frac{n}{2L} - 1$ segments, each of length $4L$. See Figure 3.1 for illustration. By making use of Corollary 3.2, we are able to compute the segment $A[i_k, j_k] \subseteq A_k$ maximizing the eccentricity subject to the length constraint in $O(L^{1/2}T(L^{1/2}))$ time for each $k = 0, 1, \ldots, \frac{n}{2L} - 2$. According to Lemma 3.2, some $A[i_k, j_k]$ must be the optimal solution. The detailed algorithm is given below.

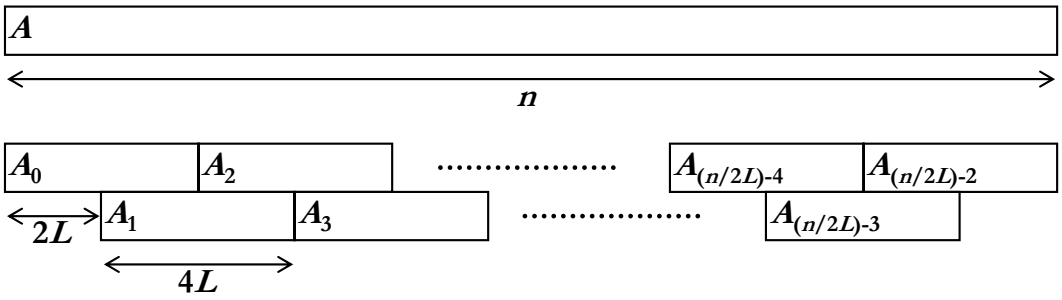


Figure 3.1: Illustration of $A_i, i = 0, 1, 2, ..., \frac{n}{2L} - 2$.

Algorithm ComputeMES

Input: A sequence A of n numbers and a length lower bound L with $1 \leq L \leq n$.

Output: A segment $A[i, j]$ maximizing $ecc(i, j)$ subject to $length(i, j) \geq L$.

1: $(i, j) \leftarrow \arg \max_{length(p,q) \geq L} sum(p, q)$.

2: **if** $sum(i, j) = 0$ **then return** $A[i, j]$,

3: **if** $sum(i, j) > 0$ **then**

   1: Define $f : \mathbb{R}^+ \times \mathbb{R}$ by letting $f(\ell, s) = \frac{s}{\sqrt{\ell}}$ if $s \geq 0$ and 0, otherwise.

   2: **return** MSS(A, L, f).

4: **for** $k \leftarrow 0$ to $\frac{n}{2L} - 2$ **do**

   1. Compute $(w_1, \ldots, w_{4L})$, where $w_j = \max\{sum(p, q) |\ length(p, q) = j$ and $2kL+1 \leq p \leq q \leq 2kL + 4L\}$ for each $j = 1, \ldots, 4L$.
   2. Compute $ecc_j = \frac{w_j}{\sqrt{j}}$ for each $j = L, L + 1, \ldots, 2L - 1$.
   3. $j_k \leftarrow \arg\max_{j=L}^{2L-1} ecc_j$.
   4. $i_k \leftarrow \arg\max_{i=2kL+1}^{2kL+4L-j_k+1} ecc(i, i + j_k - 1)$.

5: **return** the max-eccentricity segment in $\{A[i_0, j_0], A[i_1, j_1], \ldots, A[i_{\frac{n}{2L}-2}, j_{\frac{n}{2L}-2}]\}$.

Theorem 3.3: Algorithm ComputeMES solves the Length-Constrained Max-Eccentricity Segments problem in $O(n\frac{T(L^{1/2})}{L^{1/2}})$ time, where $T(n')$ is the time required to compute the min-plus product of two $n' \times n'$ matrices.

Proof: We begin by considering the correctness. Let $(i,j) = \arg \max_{length(p,q) \geq L} sum(p,q)$ and $(i^*,j^*) = \arg \max_{length(p,q) \geq L} ecc(p,q)$.

In the case where $sum(i,j) = 0$, we have $ecc(p,q) = \frac{sum(p,q)}{\sqrt{length(p,q)}} \leq 0$ for all (p,q) with $length(p,q) \geq L$. It follows that $0 \geq ecc(i^*,j^*) \geq ecc(i,j) = 0$, so $A[i,j]$ is an optimal solution.

In the case where $sum(i,j) > 0$, we have $ecc(i^*,j^*) \geq ecc(i,j) > 0$. Let $f(\ell,s) = \frac{s}{\sqrt{\ell}}$ if $s \geq 0$ and 0, otherwise. By Lemma 3.1, f is quasiconvex, so the call to MSS(A,L,f) returns a segment $A[i',j']$ maximizing $f(i',j')$ subject to $length(i',j') \geq L$. We next prove that $ecc(i',j') \geq ecc(i^*,j^*)$, so $A[i',j']$ is an optimal solution. Note that for any segment $A[p,q]$, we have $f(p,q) = ecc(p,q)$ as long as $ecc(p,q) \geq 0$ or $f(p,q) > 0$. Since $ecc(i^*,j^*) > 0$, we have $ecc(i^*,j^*) = f(i^*,j^*) > 0$. It follows that $f(i',j') \geq f(i^*,j^*) > 0$, which implies that $ecc(i',j') = f(i',j')$. Therefore, $ecc(i',j') = f(i',j') \geq f(i^*,j^*) = ecc(i^*,j^*)$.

In the case where $sum(i,j) < 0$, we have $length(i^*,j^*) < 2L$ by Lemma 3.2. It follows that $A[i^*,j^*]$ must be contained in $A[2kL+1, 2kL+4L]$ for some $k \in \{0,1,\ldots,\frac{n}{2L}-2\}$ and thus the segment returned at Step 5 must be an optimal solution.

We next analyze the running time. By Corollary 3.1, Step 1 takes $O(n)$ time. Step 2 takes constant time. By Theorem 3.1, Step 3 takes $O(n)$ time. By Corollary 3.2, each iteration of the loop at Step 4 takes $O(L^{1/2}T(L^{1/2}) + L)$ time. Thus Step 4 takes $O(\frac{n}{L}L^{1/2}T(L^{1/2}) + n) = O(n\frac{T(L^{1/2})}{L^{1/2}})$ time. Step 5 takes $O(n/L)$ time. Therefore the total running time is $O(n\frac{T(L^{1/2})}{L^{1/2}})$. □

3.4 Notes

To the best of our knowledge, there is not any non-trivial lower bound for the Length-Constrained Max-Eccentricity Segments problem proved so far. Thus, there is still a large gap between the trivial lower bound of $O(n)$ and the upper bound of $O(nL\frac{(\log\log L)^3}{(\log L)^2})$ for the Length-Constrained Max-Eccentricity Segments problem. Bridging this gap remains an open problem.

Chapter 4

Sum-Constrained Max-Density Intervals

Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2) \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, define the *aberrance* of an index interval $I = [i, j] = \{i, i+1, \ldots, j\}$ to be $aberr(i, j) = \frac{|\sum_{k=i}^{j} s_k|}{\sqrt{\sum_{k=i}^{j} \ell_k}}$. Consider the following problems arising in association rule mining [39, 40], computational biology [3, 15, 20, 24, 25, 31, 41, 44, 48, 51, 53, 73], and statistics [28].

- Sum-Constrained Max-Density Interval (SDI) Problem: Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a sum lower bound L, an index interval $I = [i, j]$ is said to be *endorsed* if and only if $sum(i, j) \geq L$. The problem is to find an endorsed interval $I = [i, j]$ maximizing the density $d(i, j)$. Bernholt *et al.* [15]'s results imply an $O(n \log n)$-time algorithm for this problem, and we give an $O(n)$-time algorithm in this chapter.

- Density-Constrained Max-Sum Interval (DSI) Problem: Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a density lower bound L, find an index interval $I = [i, j]$ maximizing the sum $sum(i, j)$ subject to $d(i, j) \geq L$. Bernholt *et al.* [15]'s results imply an $O(n \log n)$-time algorithm for this problem, and recently, Cheng *et al.* [24] obtained an $O(n)$-time algorithm.

- Length-Constrained Max-Density Interval (LDI) Problem: Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a length

lower bound L, find an index interval $I = [i, j]$ maximizing the density $d(i, j)$ subject to $length(i, j) \geq L$. This problem was studied in [15, 25, 39, 41, 44, 48, 51, 53] and can be solved in $O(n)$ time [15, 25, 39, 41, 51].

- Density-Constrained Max-Length Interval (DLI) Problem: Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a density lower bound L, find an index interval $I = [i, j]$ maximizing the length $length(i, j)$ subject to $d(i, j) \geq L$. This problem was studied in [3, 15, 20, 39, 73] and can be solved in $O(n)$ time [39].

- Length-Constrained Max-Aberrance Interval (LAI) Problem: Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, a length lower bound L, and a length upper bound U, find an index interval $I = [i, j]$ maximizing the aberrance $aberr(i, j)$ subject to $L \leq length(i, j) \leq U$. Bernholt *et al.* [15] proposed an $O(n)$-time for this problem.

- Length-Constrained Max-Sum Interval (LSI) Problem: Given a sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a length lower bound L, find an index interval $I = [i, j]$ maximizing the sum $sum(i, j)$ subject to $length(i, j) \geq L$. This problem was solvable in $O(n)$ time by algorithms in [15, 31, 53].

4.1 Preliminaries

For ease of exposition, we assume the sum lower bound $L \geq 0$ in subsequent discussion. The restriction can be overcame as follows. If $L < 0$ and $s_i \geq 0$ for some i, then it is safe to reset L to 0. Otherwise, if $L < 0$ and $s_i < 0$ for all i, let $D' = ((s'_1, \ell'_1), (s'_2, \ell'_2), \ldots, (s'_n, \ell'_n)) = ((\ell_1, -s_1), (\ell_2, -s_2), \ldots, (\ell_n, -s_n))$ and $U = -L$. The problem is then reduced to finding an index interval $I = [i, j]$ that maximizes $\frac{\sum_{k=i}^{j} s'_k}{\sum_{k=i}^{j} \ell'_k}$ subject to $\sum_{k=i}^{j} \ell'_k \leq U$, which is solvable in $O(n)$ time in an online manner by Chung and Lu's algorithm [25].

Let $S = (s_1, s_2, \ldots, s_n)$ and $P_S[0..n]$ be the prefix-sum array of S, where $P_S[0] = 0$ and $P_S[i] = P_S[i-1] + s_i$ for each $i = 1, 2, \ldots, n$. Let $\pounds = (\ell_1, \ell_2, \ldots, \ell_n)$ and $P_{\pounds}[0..n]$ be the prefix-sum array of $\pounds$, where $P_{\pounds}[0] = 0$ and $P_{\pounds}[i] = P_{\pounds}[i-1] + \ell_i$ for each $i = 1, 2, \ldots, n$. Both P_S and

$P_{\pounds}$ can be computed in $O(n)$ time in an online manner. Note that $sum(i,j) = P_S[i] - P_S[j-1]$, $length(i,j) = P_{\pounds}[i] - P_{\pounds}[j-1]$, and $d(i,j) = \frac{P_S[i]-P_S[j-1]}{P_{\pounds}[i]-P_{\pounds}[j-1]}$. Therefore, by keeping P_S and $P_{\pounds}$, each computation of the sum, length, or density of an index interval can be done in constant time. We next introduce the notion of partners. For technical reasons, we define $sum(0,p) = L$ and $d(0,p) = -\infty$ for all indices p.

Definition 4.1: Given an index q, an nonnegative integer p is said to be a *partner* of q if and only if $sum(p,q) \geq L$.

Definition 4.2: Given an index q, an integer p is said to be the *best partner* π_q of q if and only if p is the largest partner of q such that $d(p,q) = \max\{d(i,q) : i \text{ is a partner of } q\}$.

Definition 4.3: Denote by r_q the right most partner (i.e., the largest partner) of index q. Define the *ideal right most partner* of index q as $\hat{r}_q = \max_{1 \leq p \leq q} r_p$.

Definition 4.4: An index q is said to be a *good index* if and only if $\hat{r}_q = r_q$.

Let q^* be an index that maximizes $d(\pi_{q^*}, q^*)$. If $\pi_{q^*} = 0$, then there is no endorsed interval; otherwise, index interval $[\pi_{q^*}, q^*]$ is a maximum-density endorsed interval. Therefore, to solve the SDI problem, it suffices to find index q^*. The next lemma enable us to eliminate bad indices from consideration.

Lemma 4.1: If q is a bad index, then index interval $[\pi_q, q]$ is not a maximum-density endorsed interval.

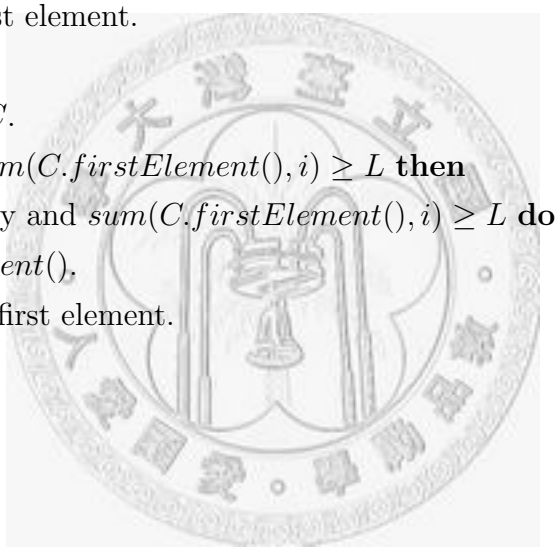
Proof: If q is a bad index, then there exist $p < q$ such that $r_q < \hat{r}_q = r_p$. Since $sum(p, \hat{r}_q) \geq L$ and $sum(q, \hat{r}_q) < L$, we have $s_p > s_q$. It follows that if index interval $[\pi_q, q]$ is an endorsed interval, then index interval $[\pi_q, p]$ is an endorsed interval and $d(\pi_q, p) > d(\pi_q, q)$. Thus, index interval $[\pi_q, q]$ is impossible to be a maximum-density endorsed interval. □

4.2 Subroutine

We next give a subroutine to compute r_q for all good indices q in $O(n)$ time. The pseudocode is given below, where the goal is to fill in an array $R[1..n]$, initialized with -1's, such that $R[i] = r_i$ for all good indices i and $R[i] = -1$ for all bad indices i at the end.

Subroutine RMP

1: Create an array $R[1..n]$ initialized with -1's.
2: $\hat{R} \leftarrow 0$.
3: Create an empty list C;
4: **for** $i \leftarrow 1$ to n **do**
5: **while** C is not empty and $sum(C.lastElement() + 1, i) \leq 0$ **do**
6: Delete from C its last element.
7: **end while**
8: Insert i at the end of C.
9: **if** $sum(\hat{R}, i) \geq L$ or $sum(C.firstElement(), i) \geq L$ **then**
10: **while** C is not empty and $sum(C.firstElement(), i) \geq L$ **do**
11: $\hat{R} \leftarrow C.firstElement()$.
12: Delete from C its first element.
13: **end while**
14: $R[i] \leftarrow \hat{R}$.
15: **end if**
16: **end for**
17: **return** $R[1..n]$.



The Subroutine RMP consists of n iterations, in the i^{th} iteration, $R[i]$ is reset to r_i if i is an good index. To accomplish this task efficiently, we maintain a list C and a variable $\hat{R}$ such that at the end of the i^{th} iteration the following conditions hold.

1. For each two adjacent elements $p < q$ in C, $sum(p + 1, q) > 0$.
2. For any index $q \geq i$, if $r_q \in [\hat{R}, i]$, then $r_q \in C \cup \{\hat{R}\}$.
3. $\hat{R} = \hat{r}_i$.

It is clear that $R[1] = r_1 = 0$ and the three conditions hold at the end of the first iteration of the **for**-loop. Suppose that the three conditions hold at the end of the $(i-1)^{th}$ iteration of the **for**-loop. We prove that $R[i]$ is reset to r_i in the i^{th} iteration of the **for**-loop if and

only if i is a good index and the three conditions hold at the end of the i^{th} iteration of the **for**-loop. Consider the moment immediately before the execution of line 9 in the i^{th} iteration of the **for**-loop. It is clear that condition 1 still holds at this moment. We next prove that condition 2 also holds at this moment. Suppose for contradiction that some r_q, where $q \geq i$, is removed from C during the execution of the **while**-loop of lines $4 \sim 7$. A necessary condition for r_q to be deleted is $sum(r_q+1, i) \leq 0$. It follows that $sum(i, q) \geq sum(r_q, q) \geq L$, so r_q is not the right most partner of q, a contradiction. Therefore, we have conditions 1 and 2 hold and $\hat{R} = \hat{r}_{i-1}$ just before the execution of line 9. We next examine the execution of lines $9 \sim 15$. Consider the following three cases.

Case 1: $r_i \notin C \cup \{\hat{R}\}$ just before the execution of line 9. By condition 2, we have $r_i < \hat{r}_{i-1}$, so index i is not good and $\hat{r}_{i-1} = \hat{r}_i$. Thus, condition 3 holds just before the execution of line 9. In this case we fail the test condition in line 9 so lines $10 \sim 14$ are not executed. Therefore, $R[i]$ is not reset, and conditions $1 \sim 3$ hold at the end of the i^{th} iteration of the **for**-loop.

Case 2: $r_i = \hat{R}$ just before the execution of line 9. It follows that index i is good and $\hat{r}_{i-1} = r_i = \hat{r}_i$. Thus, condition 3 holds just before the execution of line 9. The body of the **while**-loop of lines $10 \sim 13$ is not executed in this case. Therefore, $R[i]$ is reset to $\hat{R} = r_i$ in line 14, and conditions $1 \sim 3$ hold at the end of the i^{th} iteration of the **for**-loop.

Case 3: $r_i \in C$ just before the execution of line 9. It follows that $r_i > \hat{r}_{i-1}$, so $\hat{r}_i = r_i$ and index i is good. By conditions 1, for all $c \in C$ before r_i, $sum(c, i) > sum(r_i, i) \geq L$. Moreover, since r_i is the right most partner of i and indices in C are in increasing order, for all $c' \in C$ after r_i, $sum(c', i) < L$. Therefore, we have $\hat{R} = r_i$ holds after the execution of the **while**-loop of lines $10 \sim 13$. It follows that condition 3 holds at the end of the i^{th} iteration of the **for**-loop since $r_i = \hat{r}_i$. It is clear that deleting from C a prefix has no harm to condition 1, so it remains to prove that condition 2 still holds. Suppose for contradiction that condition 2 does not hold at the end of the i^{th} iteration of the **for**-loop. It follows that some r_q with $r_i < r_q \leq i$ is removed from C, which leads to a contradiction because r_i is the largest index removed from C in the **while**-loop of lines $10 \sim 13$.

We next analyze the running time. In each iteration of the **while**-loop of lines $5 \sim 7$ and the **while**-loop of lines $10 \sim 13$, there is one index in C removed. Since each index is inserted into C at most once, the time spent on these two **while**-loops is bounded by $O(n)$. To summarize,

we have the following lemma.

Lemma 4.2: Subroutine RMP computes in $O(n)$ time an array $R[1..n]$ such that $R[i]$ is the right most partner of i if i is a good index and $R[i] = -1$ if i is a bad index.

Denote by $\phi(x, y)$ the largest z in $[x, y]$ that minimizes $d(x, z)$. We next describe a subroutine which finds the largest $p \in [l, u]$ that maximizes $d(p, q)$ given $0 \leq l \leq u \leq q$. The pseudocode is given below, where we initialize a variable p with value l and then repeatedly resetting p to $\phi(p, u-1)+1$ until $p = u$ or the lowest density interval starting from p and ending before u has the same density as interval $[p, q]$. The pseudocode is given below.

Subroutine BEST(l, u, q)
1: $p \leftarrow l$.
2: **while** $p < u$ and $d(p, \phi(p, u-1)) \leq d(p, q)$ **do**
3: $\quad p \leftarrow \phi(p, u-1)+1$.
4: **end while**
5: **return** p.

Lemma 4.3: [25] The call to BEST(l, u, q) returns the largest $p \in [l, u]$ that maximizes $d(p, q)$ if $0 \leq l \leq u \leq q$.

Lemma 4.4: Let p be the return value of the call to BEST(l, r_q, q). Then $p = \pi_q$ if $\pi_q \in [l, r_q]$ and $0 \leq l \leq r_q$.

Proof: Suppose that p is not a partner of q, i.e., $sum(p, q) < L \leq sum(r_q, q)$. It follows that $d(p, q) < d(r_q, q)$, which contradicts Lemma 4.3. Thus, p must be a partner of q. Suppose for contradiction that $p \neq \pi_q$. Since p is a partner of q, by the definition of best partners, we have either $(d(p, q) < d(\pi_q, q))$ or $(d(p, q) = d(\pi_q, q)$ and $p < \pi_q)$, which contradicts Lemma 4.3. □

Chung and Lu [25] also gave efficient implementations of Subroutine BEST in their paper, which directly implies the next lemma.

Lemma 4.5: [25] A sequence of consecutive calls to Subroutine BEST, say BEST(l_1, u_1, q_1), BEST(l_2, u_2, q_2),..., BEST(l_k, u_k, q_k), can be completed in total $O(u_k + k)$ time provided that $l_1 = 0$, $l_i =$ BEST$(l_{i-1}, u_{i-1}, q_{i-1})$ for each $i = 2, 3, \ldots, k$, $u_1 \leq u_2 \leq \cdots \leq u_k$, and $u_i \leq q_i$ for each $i = 1, 2, \ldots, k$.

4.3 Algorithm

Our algorithm for the SDI problem is as follows. First, we initialize a variable l with value 0 and call Subroutine RMP to compute an array $R[1..n]$ such that $R[i] = r_i$ if i is a good index and $R[i] = -1$ if i is a bad index. Then, for each good index q, taken in increasing order, call BEST$(l, R[q], q)$ to compute the largest $l_q \in [l, R[q]]$ that maximizes $d(l_q, q)$ and reset variable l to l_q. Finally, the index interval $[l_q, q]$ that maximizes $d(l_q, q)$ is returned. The pseudocode is given below.

Algorithm ComputeHCI

Input: A sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a sum lower bound L.

Output: An index interval $I = [i, j]$ maximizing $d(i, j)$ subject to $sum(i, j) \geq L$.

```
1: l ← 0.
2: R ← call Subroutine RMP.
3: c_max ← −∞.
4: (α, β) = (0, 0);
5: for q ← 1 to n do
6:    if R[q] ≠ −1 then
7:       l ← BEST(l, R[q], q).
8:       if d(l, q) > c_max then
9:          c_max ← d(l, q).
10:         (α, β) ← (l, q).
11:      end if
12:   end if
13: end for
14: return [α, β].
```

Theorem 4.1: Algorithm ComputeHCI solves the SDI problem in $O(n)$ time.

Proof: We first prove the correctness. Let $Q = \{q_1, q_2, \ldots, q_k\}$ be the set of good indices, where $q_1 < q_2 \cdots < q_k$ and $k = |Q|$. Let $l_{q_0} = 0$ and l_{q_i} be the return value of the call to Subroutine BEST in the q_i^{th} iteration of the **for**-loop, $i = 1, 2, \ldots, k$. Note that l_{q_i} is the largest integer in $[l_{q_{i-1}}, r_{q_i}]$ that maximizes $d(l_{q_i}, q)$ for all i with $1 \leq i \leq k$. Let q_{i^*} be the good

index that maximizes $d(\pi_{q_{i^*}}, q_{i^*})$. By Lemma 4.1, it suffices to prove that $\pi_{q_{i^*}} = l_{q_{i^*}}$. To prove $\pi_{q_{i^*}} = l_{q_{i^*}}$, by Lemma 4.4, it suffices to prove that $\pi_{q_{i^*}} \in [l_{q_{i^*-1}}, r_{q_{i^*}}]$, i.e., $\pi_{q_{i^*}} \geq l_{q_{i^*-1}}$. Suppose for contradiction that $\pi_{q_{i^*}} < l_{q_{i^*-1}}$. Let $q_t \leq q_{i^*-1}$ be the first good index such that $\pi_{q_{i^*}} < l_{q_t}$. Then we have $\pi_{q_{i^*}} \in [l_{q_{t-1}}, r_{q_t}]$ and by Lemma 4.3, l_{q_t} is the largest index in $[l_{q_{t-1}}, r_{q_t}]$ that maximizes $d(l_{q_t}, q)$. It follows that

$$d(\pi_{q_{i^*}}, l_{q_t} - 1) \leq d(l_{q_t}, q_t). \tag{4.1}$$

Since $d(l_{q_t}, q_t) \geq d(r_{q_t}, q_t) \geq \frac{L}{length(r_{q_t}, q_t)} \geq 0$ and $length(l_{q_t}, q_t) \geq length(r_{q_t}, q_t)$, we have $sum(l_{q_t}, q_t) \geq sum(r_{q_t}, q_t) \geq L$. Therefore, l_{q_t} is a partner of q_t and we have

$$d(l_{q_t}, q_t) \leq d(\pi_{q_t}, q_t). \tag{4.2}$$

Following from inequality (1), we have $d(l_{q_t}, q_t) \leq d(q_t + 1, q_{i^*})$ for otherwise $d(\pi_{q_t}, q_t) \leq d(\pi_{q_{i^*}}, q_{i^*}) < d(l_{q_t}, q_t)$, which contradicts inequality (2). Combining inequality (1) with $d(l_{q_t}, q_t) \leq d(q_t + 1, q_{i^*})$, we have $d(\pi_{q_{i^*}}, l_{q_t} - 1) \leq d(l_{q_t}, q_t) \leq d(q_t + 1, q_{i^*})$. It follows that

$$d(\pi_{q_{i^*}}, q_{i^*}) \leq d(l_{q_t}, q_{i^*}). \tag{4.3}$$

By inequality (3), $0 \leq \frac{L}{length(r_{q_{i^*}}, q_{i^*})} \leq d(r_{q_{i^*}}, q_{i^*})$, and $d(r_{q_{i^*}}, q_{i^*}) \leq d(\pi_{q_{i^*}}, q_{i^*})$, we have

$$0 \leq d(r_{q_{i^*}}, q_{i^*}) \leq d(l_{q_t}, q_{i^*}). \tag{4.4}$$

Since $l_{q_t} \leq r_{q_t} \leq r_{q_{i^*}} \leq q_{i^*}$, we have $length(r_{q_{i^*}}, q_{i^*}) \leq length(l_{q_t}, q_{i^*})$. By inequality (4) and $length(r_{q_{i^*}}, q_{i^*}) \leq length(l_{q_t}, q_{i^*})$, we have

$$L \leq sum(r_{q_{i^*}}, q_{i^*}) \leq sum(l_{q_t}, q_{i^*}). \tag{4.5}$$

By inequalities (3) and (5), l_{q_t} is a partner of q_{i^*} and $d(\pi_{q_{i^*}}, q_{i^*}) \leq d(l_{q_t}, q_{i^*})$, which contradicts the definition of best partners.

We now analyze the time complexity. By Lemma 4.2, the call to RMP takes $O(n)$ time. By the definition of good indices, we have $R[q_1] \leq R[q_2] \leq \cdots \leq R[q_k]$. Because we also have $l_{q_i} = \text{BEST}(l_{q_{i-1}}, R[q_i], q_i)$ for each $i = 1, 2, \ldots, k$, by Lemma 4.5, the calls to Subroutine BEST, i.e., $\text{BEST}(l_{q_0}, R[q_1], q_1)$, $\text{BEST}(l_{q_1}, R[q_2], q_2)$,..., and $\text{BEST}(l_{q_{k-1}}, R[q_k], q_k)$, totally take $O(R[q_k] + k) = O(n)$ time. □

Finally, we modify Algorithm ComputeHCR such that it runs in an online manner. The modified version is given below, where we maintain an integer pair (α, β) such that after processing the q^{th} number pair in D at the q^{th} iteration of the **for**-loop, the integer interval $[\alpha, \beta]$ is a maximum-density endorsed interval for the subsequence $((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_q, \ell_q))$, if any. The correctness is easy to verify by noting that $r = R[q]$ holds at the end of the q^{th} iteration of the **for**-loop for each $q = 1, 2, \ldots, n$.

Theorem 4.2: Algorithm OnlineComputeHCR solves the SDI problem in $O(n)$ time in an online manner.

Algorithm OnlineComputeHCI

Input: A sequence $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ of number pairs, where $\ell_i > 0$ for all i, and a sum lower bound L.

Goal: Maintaining an integer pair (α, β) such that after processing the q^{th} number pair in D at the q^{th} iteration of the **for**-loop, the integer interval $[\alpha, \beta]$ is a maximum-density endorsed interval for the subsequence $((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_q, \ell_q))$, if any.

1: $l \leftarrow 0$; $r \leftarrow -1$; $\hat{r} \leftarrow 0$; $c_{max} \leftarrow -\infty$; $(\alpha, \beta) \leftarrow (0, 0)$.
2: Create an empty list C.
3: **for** $q \leftarrow 1$ to n **do**
4: $r \leftarrow -1$.
5: **while** C is not empty and $sum(C.lastElement() + 1, q) \leq 0$ **do**
6: Delete from C its last element.
7: **end while**
8: Insert q at the end of C.
9: **if** $sum(\hat{r}, q) \geq L$ or $sum(C.firstElement(), q) \geq L$ **then**
10: **while** C is not empty and $sum(C.firstElement(), q) \geq L$ **do**
11: $\hat{r} \leftarrow C.firstElement()$.
12: Delete from C its first element.
13: **end while**
14: $r \leftarrow \hat{r}$.
15: **end if**
16: **if** $r \neq -1$ **then**
17: $l \leftarrow \text{BEST}(l, r, q)$.
18: **if** $d(l, q) > c_{max}$ **then**
19: $c_{max} \leftarrow d(l, q)$.
20: $(\alpha, \beta) \leftarrow (l, q)$.
21: **end if**
22: **end if**
23: **end for**

Chapter 5

Density Finding

Given a sequence of number pairs $D = ((s_1, \ell_1), (s_2, \ell_2), \ldots, (s_n, \ell_n))$ where $\ell_i > 0$ for $i = 1, 2, \ldots, n$, two positive numbers L, U with $L \leq U$, and a real number δ, define the sum, length, and density of an index interval $[i, j]$ to be $sum(i, j) = \sum_{r=i}^{j} s_r$, $length(i, j) = \sum_{r=i}^{j} \ell_r$ and $d(i, j) = \frac{sum(i,j)}{length(i,j)}$, respectively. An index interval $[i, j]$ is said to be *feasible* if and only if $L \leq length(i, j) \leq U$. The Density Finding Problem is to compute the density of the feasible index interval $[i, j]$ which minimizes $|d(i, j) - \delta|$. Lee *et al.* [51] proved that the Density Finding Problem has a lower bound of $\Omega(n \log n)$ in the algebraic decision tree model and provided an $O(n \log^2 m)$ algorithm for it, where $m = \min(\lfloor \frac{U-L}{\ell_{min}} \rfloor, n)$ and $\ell_{min} = \min_{1 \leq r \leq n} \ell_r$. In this chapter, we show how to solve the Density Finding Problem problem in optimal $O(n \log n)$ time.

5.1 Reduction to the Slope Finding Problem

Let $P, Q \subseteq \mathbb{R}^2$ be two n-point multisets and $Ar \geq b$ be a set of λ inequalities on x and y, where $A \in \mathbb{R}^{\lambda \times 2}$, $r = [\begin{smallmatrix} x \\ y \end{smallmatrix}]$, and $b \in \mathbb{R}^{\lambda}$. Define the *constrained Minkowski sum* $(P \oplus Q)_{Ar \geq b}$ as the multiset

$$\{(p + q) : p \in P, q \in Q, A(p + q) \geq b\}.$$

Given P, Q, $Ar \geq b$, and a real number δ, the Slope Finding Problem is to compute the value of $\frac{y^*}{x^*}$ where $(x^*, y^*) = \arg \min_{(x,y) \in (P \oplus Q)_{Ar \geq b}} \{|\frac{y}{x} - \delta|\}$. The Density Finding Problem can be reduced to the Slope Finding Problem in linear time as follows. Let

$length(1,0) = 0$ and $sum(1,0) = 0$. Compute in $O(n)$ time the following two point sets: $P = \{(length(1,i), sum(1,i)) : 0 \leq i \leq n\}$ and $Q = \{(-length(1,i), -sum(1,i)) : 0 \leq i \leq n)\}$. Note that each feasible index interval $[i,j]$ of S corresponds to a point $(length(1,j) - length(1,i-1), sum(1,j) - sum(1,i-1))$ in $(P \oplus Q)_{L \leq x \leq U}$. Thus, the DENSITY FINDING PROBLEM is equivalent to computing the value of $\frac{y^*}{x^*}$ where $(x^*, y^*) = \arg \min_{(x,y) \in (P \oplus Q)_{L \leq x \leq U}} \{|\frac{y}{x} - \delta|\}$.

Theorem 5.1: The DENSITY FINDING PROBLEM is linear time reducible to the SLOPE FINDING PROBLEM with two linear constraints.

5.2 The Algorithm for the Slope Finding Problem

In this section, we give an $O(\lambda \log \lambda + \lambda \cdot n \log n)$-time algorithm for the SLOPE FINDING PROBLEM. By Theorem 5.1, it follows that the DENSITY FINDING PROBLEM is solvable in $O(n \log n)$ time. For simplicity, we assume $\delta = 0$. Otherwise, we can first perform the following transformation and then reset δ to 0.

1. Transform the point set $P = \{(x_{1,1}, y_{1,1}), (x_{1,2}, y_{1,2}), \ldots, (x_{1,n}, y_{1,n})\}$ into $\{(x_{1,1}, -\delta x_{1,1} + y_{1,1}), (x_{1,2}, -\delta x_{1,2} + y_{1,2}), \ldots, (x_{n,1}, -\delta x_{n,1} + y_{n,1})\}$
2. Transform the point set $Q = \{(x_{2,1}, y_{2,1}), (x_{2,2}, y_{2,2}), \ldots, (x_{2,n}, y_{2,n})\}$ into $\{(x_{1,2}, -\delta x_{1,2} + y_{1,2}), (x_{2,2}, -\delta x_{2,2} + y_{2,2}), \ldots, (x_{n,2}, -\delta x_{n,2} + y_{n,2})\}$.
3. Transform the i^{th} constraint $L_i : a_i x + b_i y \geq c_i$ into $(a_i + \delta b_i)x + b_i y \geq c_i$ for each $i = 1, 2, \ldots, \lambda$.

A function $f : D \to (\mathbb{R} \cup \infty)$ defined on a convex subset D of $\mathbb{R}^2$ is *quasiconcave* if whenever $v_1, v_2 \in D$ and $\gamma \in [0,1]$ then

$$f(\gamma \cdot v_1 + (1-\gamma) \cdot v_2) \geq \min\{f(v_2), f(v_2)\}.$$

For technical reasons, we define $\frac{y}{x} = \infty$ if $x = 0$ and $y \in \mathbb{R}$. Let $D_1 = \{(x,y) \in \mathbb{R}^2 : x \geq 0, y \geq 0\}$, $D_2 = \{(x,y) \in \mathbb{R}^2 : x \leq 0, y \geq 0\}$, $D_3 = \{(x,y) \in \mathbb{R}^2 : x \leq 0, y \leq 0\}$, and $D_4 = \{(x,y) \in \mathbb{R}^2 : x \geq 0, y \leq 0\}$. We next prove that the function $f : \mathbb{R}^2 \to (\mathbb{R} \cup \infty)$ defined by letting $f(x,y) = |\frac{y}{x}|$ is quasiconcave when we restrict its domain to some D_i.

Lemma 5.1: Let $D_1 = \{(x, y) \in \mathbb{R}^2 : x \geq 0, y \geq 0\}$, $D_2 = \{(x, y) \in \mathbb{R}^2 : x \leq 0, y \geq 0\}$, $D_3 = \{(x, y) \in \mathbb{R}^2 : x \leq 0, y \leq 0\}$, and $D_4 = \{(x, y) \in \mathbb{R}^2 : x \geq 0, y \leq 0\}$. Define function $f_i : D_i \to \mathbb{R}$ by letting $f_i(x, y) = |\frac{x}{y}|$ for each $i = 1, 2, 3, 4$. Then we have function f_i is quasiconcave for each $i = 1, 2, 3, 4$.

Proof: We only prove that f_1 is quasiconcave. The proofs for f_2, f_3, and f_4 can be derived in a similar way. Let $v_1 = (x_1, y_1) \in D_1$, $v_2 = (x_2, y_2) \in D_1$, and $x_1 \geq x_2$. Without loss of generality we may assume $x_1 > 0$ and $v_\gamma \notin \{v_1, v_2\}$. Consider the following two cases.

Case 1: $\frac{y_1}{x_1} \leq \frac{y_2}{x_2}$. Let $v' = (x', y')$ be the point which satisfies $\frac{y_1}{x_1} = \frac{y'}{x'}$ and $x' = x_\gamma = \gamma x_1 + (1 - \gamma)x_2$. By $x_1 > 0$, $x_2 \geq 0$, and $\frac{y_1}{x_1} \leq \frac{y_2}{x_2}$, we have $y_2 \geq \frac{x_2}{x_1} y_1$. It follows that $y' = (\gamma x_1 + (1 - \gamma)x_2)\frac{y_1}{x_1} = \gamma y_1 + (1 - \gamma)\frac{x_2}{x_1} y_1 \leq \gamma y_1 + (1 - \gamma)y_2 = y_\gamma$. By $0 < x' = x_\gamma$ and $0 \leq y' \leq y_\gamma$, we have $|\frac{y_\gamma}{x_\gamma}| = \frac{y_\gamma}{x_\gamma} \geq \frac{y'}{x'} = \frac{y_1}{x_1} = |\frac{y_1}{x_1}| \geq \min\{|\frac{y_1}{x_1}|, |\frac{y_2}{x_2}|\}$.

Case 2: $\frac{y_1}{x_1} > \frac{y_2}{x_2}$. Let $v'' = (x'', y'')$ be the point which satisfies $\frac{y_2}{x_2} = \frac{y''}{x''}$ and $x'' = x_\gamma = \gamma x_1 + (1 - \gamma)x_2$. By $x_1 > 0$ and $\frac{y_1}{x_1} > \frac{y_2}{x_2}$, we have $y_1 > \frac{x_1}{x_2} y_2$. It follows that $y'' = (\gamma x_1 + (1 - \gamma)x_2)\frac{y_2}{x_2} = \gamma \frac{x_1}{x_2} y_2 + (1 - \gamma)y_2 < y_1 + (1 - \gamma)y_2 = y_\gamma$. By $0 < x'' = x_\gamma$ and $0 \leq y'' \leq y_\gamma$, we have $|\frac{y_\gamma}{x_\gamma}| = \frac{y_\gamma}{x_\gamma} > \frac{y''}{x''} = \frac{y_2}{x_2} = |\frac{y_2}{x_2}| \geq \min\{|\frac{y_1}{x_1}|, |\frac{y_2}{x_2}|\}$. □

The next theorem was proved by Bernholt *et al.* [15].

Theorem 5.2: Given a set of λ linear inequalities $Ar \geq b$ and two n-point multisets $P, Q \subseteq \mathbb{R}^2$, one can compute the vertices of the convex hull of $(P \oplus Q)_{Ar \geq b}$ in $O(\lambda \log \lambda + \lambda \cdot n \log n)$ time.

Let R_i be the vertices of the convex hull of $(P \oplus Q)_{Ar \geq b} \cap D_i$ for each $i = 1, 2, 3, 4$. It should be noted that

$$\begin{cases} (P \oplus Q)_{Ar \geq b} \cap D_1 = (P \oplus Q)_{Ar \geq b, x \geq 0, y \geq 0}; \\ (P \oplus Q)_{Ar \geq b} \cap D_2 = (P \oplus Q)_{Ar \geq b, x \leq 0, y \geq 0}; \\ (P \oplus Q)_{Ar \geq b} \cap D_3 = (P \oplus Q)_{Ar \geq b, x \leq 0, y \leq 0}; \\ (P \oplus Q)_{Ar \geq b} \cap D_4 = (P \oplus Q)_{Ar \geq b, x \geq 0, y \leq 0}. \end{cases}$$

Thus, by Theorem 5.2, each R_i is computable in $O((\lambda + 2)\log(\lambda + 2) + (\lambda + 2) \cdot n \log n) = O(\lambda \log \lambda + \lambda \cdot n \log n)$ time. Let $sol_i = \min\{|\frac{y}{x}| : (x, y) \in (P \oplus Q)_{Ar \geq b} \cap D_i\}$ for each $i = 1, 2, 3, 4$. By Lemma 5.1, we have $sol_i = \min\{|\frac{y}{x}| : (x, y) \in R_i\}$ for each i. Note that the size of each R_i is

bounded above by $O(\lambda + n)$. Therefore, each sol_i is computable in $O(\lambda + n)$ time by examining all points in R_i. Finally, we have the solution is $\min_{i=1}^{4} sol_i$.

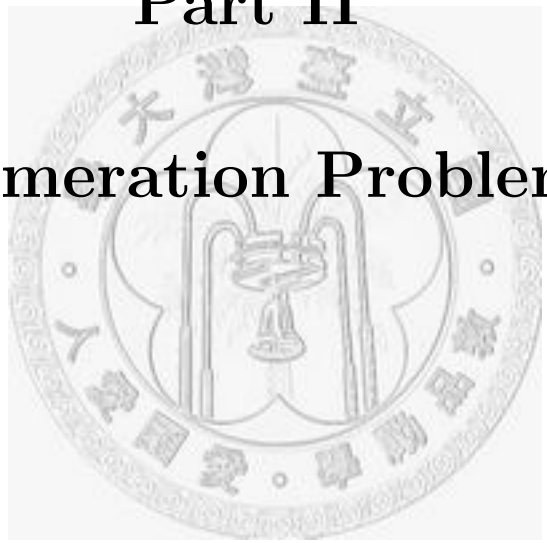
Theorem 5.3: The SLOPE FINDING PROBLEM is solvable in $O(\lambda \log \lambda + \lambda \cdot n \log n)$ time.

Corollary 5.1: The DENSITY FINDING PROBLEM is solvable in $O(n \log n)$ time.

Proof: Immediately from Theorem 5.1 and Theorem 5.3. □

Part II

Enumeration Problems



Chapter 6

Length-Constrained k Max-Sum Segments

Given a sequence $A = (a_1, a_2, \ldots, a_n)$ of numbers, a positive integer k, and an interval $[L, U]$, the Length-Constrained k Max-Sum Segments Problem is to find the k max-sum segments among all segments of A with lengths in the interval $[L, U]$.

For the case where k is fixed to 1, this problem is studied in [31, 53] and can be solved in linear time [31]. For the case where L is fixed to 1 and U is fixed to n, this problem is well studied in [5, 6, 11, 18, 23, 30, 54, 55] and can be solved in $O(n + k)$ time [18, 30].

In this chapter we give an $O(n+k)$-time algorithm for the Length-Constrained k Max-Sum Segments Problem. In addition, we show that our algorithms can be used as a basis for delivering more efficient algorithms for some related problems such as finding k length-constrained segments satisfying a density lower bound and finding area-constrained k max-sum subarrays.

6.1 Preliminaries

To achieve the time bound of $O(n+k)$, we make use of the range max-sum query (RMSQ) [21] and Frederickson's [34] algorithm for finding the maximum k elements in a heap-ordered tree.

In the Range Max-Sum Segment Query (RMSQ) problem, a sequence $A = (a_1, a_2, \ldots, a_n)$ of n numbers is given to be preprocessed such that any range max-sum segment query can be

answered quickly. A range max-sum segment query specifies two intervals $[i, j]$ and $[k, l]$, and the goal is to find a pair of indices (x, y) with $i \leqslant x \leqslant j$ and $k \leqslant y \leqslant \ell$ that maximizes $S[x, y]$.

Theorem 6.1: [Chen and Chao [21]] The RMSQ problem can be solved in $O(n)$ preprocessing time and $O(1)$ time per query.

For our purposes, a D-heap is a rooted degree-D tree in which each node contains a field *value*, satisfying the restriction that the value of any node is larger than or equal to the values of its children. Note that we do not require the tree to be balanced. Frederickson [34] proposed an algorithm for finding the k largest elements in a D-heap in $O(k)$ time. When Frederickson's algorithm traverses the heap to find the k largest nodes, it does not access a node unless it has ever accessed the node's parent. This property makes it possible to run the Frederickson's algorithm without first explicitly building the entire heap in the memory as long as we have a way to obtain the information of a node given the information of its parent.

Theorem 6.2: [Frederickson [34]] For any constant D, we can find the k largest nodes in any D-heap, in $O(k)$ time.

6.2 An $O(n + k)$-Time Algorithm

First we preprocess the input sequence $A[1..n]$ so that given any two intervals $[i, j]$ and $[k, l]$, we can find the pair (x, y), denoted RMSQ(i, j, k, l), with $i \leqslant x \leqslant j$ and $k \leqslant y \leqslant \ell$ that maximizes $S[x, y]$. By Theorem 6.1, this preprocessing can be done in $O(n)$ time. In the following, we say a segment $A[i, j]$ is legal if and only if $L \leq j - i + 1 \leq U$. Set $p_i = \max\{i - U + 1, 1\}$ and $q_i = i - L + 1$ for all $i = 1, \ldots, n$. For simplicity, we assume $p_i \leq q_i$ for all $i = 1, \ldots, n$. Then $\bigcup_{i=1}^{n} \{A[h, i] : h \in [p_i, q_i]\}$ is the set of all legal segments. Our task is to find the k max-sum segments in this set.

We now turn to define some data structures used in our algorithm. For each index i, define $H(i)$ to be a rooted ordered binary tree in which each node contains three fields *pair*, *value*, and *interval*, satisfying the following properties: (1) There are total $q_i - p_i + 1$ nodes in $H(i)$ and the interval of the root of $H(i)$ is $[p_i, q_i]$, (2) for each node u of $H(i)$, if $p < k$ then u's left child has interval $[p, k - 1]$, and if $k < q$ then u's right child has interval $[k + 1, q]$, where $[p, q]$

$= u.interval$ and (k, i) =RMSQ(p, q, i, i), and (3) for each node u of $H(i)$, if $u.interval = [p, q]$ then $u.pair = (k, i)$ and $u.value = S[k, i]$, where (k, i)=RMSQ(p, q, i, i).

Let us now return to describe our algorithm. Let $V(H(i))$ denote the set of nodes in $H(i)$. It is clear that the k largest value nodes in $\bigcup_{i=1}^{n} V(H(i))$ correspond to the k max-sum legal segments. Thus the remaining work is to find the k largest value nodes in $\bigcup_{i=1}^{n} V(H(i))$. Notice that given any node u of $H(i)$, we can always construct u's children in $O(1)$ time since we have done the RMSQ preprocessing on $A[1..n]$. Thus we only construct the root of $H(i)$ in the first instance and expand the tree as needed. Since we have known p_i and q_i for each index i and done the RMSQ preprocessing on $A[1..n]$, we can construct, in total $O(n)$ time, the root of $H(i)$ for each index i. Then we place these roots into a balanced 2-heap by the *heapify* operation [26] in $O(n)$ time. Note that each $H(i)$ is a 2-heap, so we have conceptually built a 4-heap for the set $\bigcup_{i=1}^{n} V(H(i))$. Now by Theorem 7.3, we can apply Frederickson's algorithm [34] to find the k largest value nodes in that 4-heap in $O(k)$ time. As before, except the roots of all $H(i)$, all the nodes in that 4-heap are not physically created until they are needed in running Frederickson's [34] algorithm. We summarize this section by the following theorem.

Theorem 6.3: Given a sequence $A = (a_1, \ldots, a_n)$ of numbers, a positive integer k, and an interval $[L, U]$, we can find, in $O(n + k)$ time, the k max-sum segments of A with lengths in $[L, U]$.

We prove the following stronger theorem by slightly modifying the above algorithm.

Theorem 6.4: Given a sequence of pairs of numbers $A = ((a_1, \ell_1), \ldots, (a_n, \ell_n))$, where $\ell_i > 0$ for $i = 1, \ldots, n$, a positive integer k, and an interval $[L, U]$, we can find, in $O(n + k)$ time, the k max-sum segments of A with lengths in $[L, U]$.

Proof: We show how to modify the above algorithm to achieve this theorem. In fact, we only need to change the settings of p_i's and q_i's. The remaining parts are the same. For all $i = 1, \ldots, n$, we redefine p_i to be the minimum index $1 \leq h \leq i$ such that $\pounds[h, i] \leq U$ and q_i to be the maximum index $1 \leq h' \leq i$ such that $\pounds[h', i] \geq L$. For simplicity, we assume p_i and q_i exist for all $i = 1, \ldots, n$. Since ℓ_i is positive for all $i = 1, \ldots, n$, the sequences $(p_1, \ldots, p_n)$ and $(q_1, \ldots, q_n)$ must be nondecreasing. Thus we can compute p_i and q_i for all $i = 1, \ldots, n$ by the following procedure in $O(n)$ time.

1. Set $p = 1$ and $q = 1$.

2. **for** $i \leftarrow 1$ to n **do**

   (a) **while**($\pounds[p, i] > U$ and $p \leq i$) **do** $p \leftarrow p + 1$.

   (b) **while**($\pounds[q + 1, i] \geq L$ and $q + 1 \leq i$) **do** $q \leftarrow q + 1$.

   (c) $p_i \leftarrow p$ and $q_i \leftarrow q$.

3. **return** $(p_1, \ldots, p_n)$ and $(q_1, \ldots, q_n)$.

□

6.3 Applications

6.3.1 Finding k Length-Constrained Maximum-Density Segments Satisfying a Density Lower Bound

Given a sequence of pairs of numbers $A = ((a_1, \ell_1), \ldots, (a_n, \ell_n))$, where $\ell_i > 0$ for all $i = 1, \ldots, n$, a positive integer k, an interval $[L, U]$, and a number δ, let $k_{out} = \min\{k, n_\delta\}$, where n_δ is the total number of segments of A with lengths in $[L, U]$ and densities $\geq \delta$. We show how to find k_{out} segments of A with lengths in $[L, U]$ and densities $\geq \delta$ in $O(n + k_{out})$ time. A segment $A[i, j]$ is called a legal segment if and only if the length of $A[i, j]$ is in $[L, U]$. Let δ_{max} be the density of the legal segment which has the maximum density among all legal segments. The Length-Constrained Maximum-Density Segment Problem is to find a legal segment with density equal to δ_{max}. The Length-Constrained Maximum-Density Segment Problem is well studied in [25, 41, 44, 48, 53] and can be solved in linear time by [25, 41]. Let $n_{\delta_{max}}$ be the total number of legal segments with density equal to δ_{max}. If we are not satisfied by finding only one legal segments with density equal to δ_{max}, then by first computing δ_{max} by $O(n)$-time algorithms in [25, 41] and setting δ to δ_{max}, our algorithm can list $k_{out} = \min\{k, n_{\delta_{max}}\}$ legal segments with density equal to δ_{max} in $O(n + k_{out})$ time.

Theorem 6.5: Given a sequence of pairs of numbers $A = ((a_1, \ell_1), \ldots, (a_n, \ell_n))$, where $\ell_i > 0$ for all $i = 1, \ldots, n$, a positive integer k, an interval $[L, U]$, and a number δ, let $k_{out} = \min\{k, n_\delta\}$, where n_δ is the total number of segments of A with lengths in $[L, U]$ and densities $\geq \delta$. Then we can find, in $O(n + k_{out})$ time, k_{out} segments of A with lengths in $[L, U]$ and densities $\geq \delta$.

Proof: In the following, a segment is called a legal segment if and only if its length is in $[L, U]$. Let $A' = ((a_1 - \ell_1\delta, \ell_1), (a_2 - \ell_2\delta, \ell_2), \ldots, (a_n - \ell_n\delta, \ell_n))$. Since $A[i, j]$ has density $\frac{a_i + \cdots + a_j}{\ell_i + \cdots + \ell_j} \geq \delta$ if and only if $A'[i, j]$ has sum $\sum_{i \leq h \leq j}(a_h - \ell_h\delta) \geq 0$, it suffices to show how to find k_{out} legal segments of A' with sums ≥ 0 in $O(n + k_{out})$ time. As in the proof of Theorems 6.3 and 6.4, we first implicitly construct a heap for all legal segments of A' in $O(n)$ time. Let $2^{t-1} \leq k < 2^t$. We then execute the following procedure to find k_{out} legal segments of A' with sums ≥ 0 in $O(k_{out} + 1)$ time, so the total time is $O(n + k_{out})$.

1. **for** $i \leftarrow 0$ to t **do**
   (a) $S \leftarrow$ the 2^i max-sum legal segments of A'.
   (b) **if** some segment in S has sum less than 0 **then** stop the loop.
2. **if** S contains more than k segments with sums ≥ 0 **then return** any k of them; otherwise, **return** all segments in S with sums ≥ 0.

We now prove that the procedure runs in $O(k_{out}+1)$ time. By Theorem 7.3, the i^{th} iteration of the loop in Step 1 can be done in $O(2^i)$ time. If $n_\delta = 0$, then it is clear that this procedure returns in constant time; otherwise, there are two cases to consider.

Case 1: $n_\delta \geq k$. In this case, the loop continues until the t^{th} iteration, so the total time spent on Step 1 is $O(1 + 2 + 4 + ... + 2^t) = O(2^{t+1}) = O(k) = O(\min\{k, n_\delta\}) = O(k_{out})$. After the loop stops, the size of S is at most 2^t, so Step 2 can also be done in $O(2^t) = O(k) = O(\min\{k, n_\delta\}) = O(k_{out})$ time.

Case 2: $n_\delta < k$. Let $2^{\hat{t}-1} \leq n_\delta < 2^{\hat{t}}$. In this case, the loop continues until the $\hat{t}^{th}$ iteration, so the total time spent on Step 1 is $O(1 + 2 + 4 + ... + 2^{\hat{t}}) = O(2^{\hat{t}+1}) = O(n_\delta) = O(\min\{k, n_\delta\}) = O(k_{out})$. After the loop stops, the size of S is at most $2^{\hat{t}}$, so Step 2 can also be done in $O(2^{\hat{t}}) = O(n_\delta) = O(\min\{k, n_\delta\}) = O(k_{out})$ time. □

6.3.2 Finding the Area-Constrained k Max-Sum Subarrays

Given an $n \times n$ array $A[1..n][1..n]$, define the sum and area of a subarray $A[k..l][i..j]$ to be $\sum_{p=k}^{l} \sum_{q=i}^{j} A[p][q]$ and $(l-k+1)(j-i+1)$ respectively. The k MAX-SUM SUBARRYS PROBLEM is well studied in [5, 6, 7, 11, 18, 23, 54, 55] and can be solved in $O(n^3 + k)$ time by Brodal and Jørgensen [18] and $O(n^3 \cdot \sqrt{\frac{\log \log n}{\log n}} + k \log n)$ time by Bae and Takaoka [7].

In the following, we prove that the AREA-CONSTRAINED k MAX-SUM SUBARRAYS PROBLEM can be solved in $O(n^3 + k)$ time by applying Theorem 6.4.

Theorem 6.6: Given an $n \times n$ array $A[1..n][1..n]$ and an interval $[L, U]$, we can find, in $O(n^3 + k)$ time, the k max-sum subarrays with areas in $[L, U]$.

Proof: First we have to construct strips $S_{i,j} = ((\sum_{i \le h \le j} A[h][1], j-i+1), (\sum_{i \le h \le j} A[h][2], j-i+1), \ldots, (\sum_{i \le h \le j} A[h][n], j-i+1))$ for all $1 \le i \le j \le n$ in $O(n^3)$ time. Then we construct a sequence S by concatenating these strips with pairs $(0, U+1)$. It is not hard to see that each segment of S with length in $[L, U]$ corresponds to a subarray of A with area in $[L, U]$, and vice versa. Thus we can first apply Theorem 6.4 to find k max-sum segments of S with lengths in $[L, U]$ in $O(n^3+k)$ time and then output their corresponding subarrays of A. $\square$

Chapter 7

Weight-Constrained k Longest Paths

Let $T = (V, E)$ be a tree with a length function $\ell : E \to \mathbb{R}$ and a weight function $w : E \to \mathbb{R}$. Define the length and weight of a path $P = (v_1, v_2, \ldots, v_n)$ in T to be $\sum_{1 \le i \le n-1} \ell(\overline{v_i v_{i+1}})$ and $\sum_{1 \le i \le n-1} w(\overline{v_i v_{i+1}})$ respectively. Given T, a positive integer k and two numbers L, U with $L \le U$, the Weight-Constrained k Longest Paths Problem is to find the k longest paths among all paths in T with weights in the interval $[L, U]$.

For the case where there are no weight constraints, i.e., $L = -\infty$ and $U = \infty$, Megiddo *et al.* [61] proposed an $O(|V| \log^2 |V|)$-time algorithm for finding the k^{th} longest path, and then Frederickson and Johnson [36] proposed an $O(|V| \log |V|)$-time algorithm for the same problem. Their algorithms can be easily modified to find the k longest paths in $O(|V| \log^2 |V| + k)$ and $O(|V| \log |V| + k)$ time respectively. Wu *et al.* [74] gave an $O(|V| \log^2 |V|)$-time algorithm for the case $L = -\infty$ and $k = 1$. Kim [49] gave an $O(|V| \log |V|)$-time algorithm for the case where $L = 0$, $U = \infty$, $k = 1$, $\ell(e) = 1$ $\forall e \in E$, and T has a constant degree.

In this chapter we give an $O(|V| \log |V| + k)$-time algorithm for the Weight-Constrained k Longest Paths Problem and prove the Weight-Constrained k Longest Paths Problem has a lower bound $\Omega(|V| \log |V| + k)$ in the algebraic computation tree model.

7.1 Preliminaries

To achieve the time bound of $O(|V| \log |V| + k)$, we make use of Frederickson and Johnson's [36] representation of intervertex distances of a tree, range maxima query (RMQ) [10, 33, 43], and

Frederickson's [34] algorithm for finding the maximum k elements in a heap-ordered tree. In the following, we briefly review these data structures and algorithms.

Definition 7.1: Let $T = (V, E)$ be a tree. A node $v \in V$ is said to be the *centroid* of T if and only if after removing v from T, each resulting connected component contains at most $|V|/2$ nodes.

Definition 7.2: Let $T = (V, E)$ be a tree. A triplet $(c, T_1 = (V_1, E_1), T_2 = (V_2, E_2))$ is called a *centroid decomposition* of T if it satisfies the following properties: (1) c is a centroid of T; (2) T_1 and T_2 are two subtrees of T such that $V_1 \cap V_2 = c$, $\frac{|V|+2}{3} \leq |V_1| \leq \frac{2|V|+1}{3}$, and $E_1 \cup E_2 = E$.

Notation 7.1: Let $T = (V, E)$ be a tree with a length function $\ell : E \rightarrow \mathbb{R}$ and a weight function $w : E \rightarrow \mathbb{R}$. We slightly overload the notation by letting $\ell(u, v)$ and $w(u, v)$ also denote the length and weight of the path from u to v if there is no edge from u to v.

Definition 7.3: Let $T = (V, E)$ be a tree with a length function $\ell : E \rightarrow \mathbb{R}$ and a weight function $w : E \rightarrow \mathbb{R}$. A rooted ordered binary tree $T' = (V', E', r)$ in which each node contains fields *cent*, $list_1$ and $list_2$ is called a *centroid decomposition tree* of T if it satisfies the following recursive properties: (1) If $|V| = 1$, then $|V'| = 1$, $r.cent$ = the only vertex in V, and $r.list_1 = r.list_2$=NIL; (2) if $|V| = 2$, then $|V'| = 1$, $r.cent$ = one of the vertex in V, $r.list_1 = ((v, \ell(r.cent, v), w(r.cent, v))$, and $r.list_2 = ((r.cent, 0, 0))$, where $v \in V \backslash \{r.cent\}$; (3) if $|V| > 2$, then $\exists$ centroid decomposition $(c, T_1 = (V_1, E_1), T_2 = (V_2, E_2))$ of T such that the left subtree and right subtree of r are centroid decomposition trees of T_1 and T_2 respectively, $r.cent = c$, and $r.list_j$, $j \in \{1, 2\}$, is a list of triplets $((v_i, \ell(c, v_i), w(c, v_i))|v_i \in V_j - \{c\})$ sorted on $w(c, v_i)$.

As an illustration, a tree T and its centroid decomposition tree T' are shown in Figure 7.1 and Figure 7.2 respectively.

Theorem 7.1: [Frederickson and Johnson [36]] Given a tree $T(V, E)$ with a length function $\ell : E \rightarrow \mathbb{R}$ and a weight function $w : E \rightarrow \mathbb{R}$, we can construct a centroid decomposition tree of T in $O(|V| \log |V|)$ time.

Now we describe the RANGE MAXIMA QUERY (RMQ) problem. In the RMQ problem, a list $A = (a_1, a_2, \ldots, a_n)$ of n real numbers is given to be preprocessed such that any range maxima query can be answered quickly. A range maxima query specifies an interval $[i, j]$ and the goal is to find the index k with $i \leq k \leq j$ such that a_k achieves maximum.

We first describe a simple algorithm for solving the RMQ problem in $O(n \log n)$ preprocessing time and $O(1)$ time per query. For each $1 \leq i \leq n$ and each $1 \leq j \leq \lfloor \log n \rfloor$, we precompute $M[i][j] = \arg\max_{k=i,\ldots,i+2^j-1}\{a_k\}$, i.e., the index of the maximum element in $A[i, i + 2^j - 1]$. This can be done in $O(n \log n)$ time by using dynamic programming because

$$M[i][j] = \begin{cases} M[i][j-1] & \text{if } A[M[i][j-1]] \geq A[M[i+2^{j-1}-1][j-1]]; \\ M[i+2^{j-1}-1][j-1] & \text{otherwise.} \end{cases}$$

Given a query interval $[i, j]$, let $k = \lfloor \log(j - i) \rfloor$. Because both $[i, i + 2^k - 1]$ and $[j - 2^k + 1, j]$ are subintervals of $[i, j]$ and $[i, i + 2^k - 1] \cup [j - 2^k + 1, j] = [i, j]$, the index of the maximum element in $A[i, j]$ is $\arg\max_{k \in \{M[i][i+2^k-1], M[j-2^k+1][j]\}}\{A[k]\}$.

We now sketch an algorithm for solving the RMQ problem in $O(n)$ preprocessing time and $O(1)$ time per query. This algorithm was given by Bender and Farach-Coltongiven [10], and they showed that the RMQ problem is linearly equivalent to the RMQ$\pm$1 problem which is the same as the RMQ problem except that the adjacent elements of the input list differ by exactly one. Thus, in the following we focus on the RMQ$\pm$1 problem. Let $A = (a_1, a_2, \ldots, a_n)$ be an instance to the RMQ$\pm$1 problem.[1] The algorithm starts by dividing the list A into $2n/\log n$ shorter sublists $A[1, \frac{\log n}{2}], A[\frac{\log n}{2} + 1, \log n], \ldots, A[n - \frac{\log n}{2} + 1, n]$, each of length $\frac{\log n}{2}$. Each sublist $A[\frac{(i-1)\log n}{2} + 1, \frac{i \log n}{2}]$ is represented by the maximum element r_i in it. They then run the simple RMQ algorithm described in the beginning on these $O(n/\log n)$ representatives in $O(\frac{n}{\log n} \log(\frac{n}{\log n})) = O(n)$ preprocessing time. By the property that adjacent elements in the list A differs by exactly one, they use a table-lookup technique to precompute the indices of the maximum elements in all sublists of A with lengths $\leq \frac{2n}{\log n}$ in $O(n)$ time. Given a query interval $[i, j]$, let $i' = \lceil \frac{2i}{\log n} \rceil$ and $j' = \lfloor \frac{2j}{\log n} \rfloor$. Let r_k be the maximum of $\{r_{i'+1} r_{i'+2}, \ldots, r_{j'-1}\}$, a_{i^*} be the maximum element in $A[i, \frac{i' \log n}{2}]$, and a_{j^*} be the maximum element in $A[\frac{j' \log n}{2}, j]$. Because we have run the simple RMQ algorithm on $(r_1, r_2, \ldots, r_{\frac{2n}{\log n}})$, k can be found in constant time given

[1] For simplicity, we assume n is a power of two.

$[i'+1, j'-1]$. Because both $A[i, \frac{i' \log n}{2}]$ and $A[\frac{j' \log n}{2}, j]$ have lengths $\leq \frac{2n}{\log n}$, we can also find a_{i^*} and a_{j^*} in constant time. Note that the maximum of $\{r_k, a_{i*}, a_{j*}\}$ is also the maximum element in $A[i, j]$. Thus, if a_{i^*} is the maximum of $\{r_k, a_{i*}, a_{j*}\}$, then we can directly return i^*. Similarly, if a_{j^*} is the maximum of $\{r_k, a_{i*}, a_{j*}\}$, then return j^*. Otherwise, if r_k is the the maximum of $\{r_k, a_{i*}, a_{j*}\}$, then find and return the index of the maximum element in $A[\frac{(k-1)\log n}{2} + 1, \frac{k \log n}{2}]$, which can be done in constant time because $A[\frac{(k-1)\log n}{2} + 1, \frac{k \log n}{2}]$ has length equal to $\frac{2n}{\log n}$.

Theorem 7.2: [RMQ [10, 33, 43]] The RMQ problem can be solved in $O(n)$ preprocessing time and $O(1)$ time per query.

For our purposes, a D-heap is a rooted degree-D tree in which each node contains a field *value*, satisfying the restriction that the value of any node is larger than or equal to the values of its children. Note that we do not require the tree to be balanced. Frederickson [34] proposed an algorithm for finding the k largest elements in a D-heap in $O(k)$ time. When Frederickson's algorithm traverses the heap to find the k largest nodes, it does not access a node unless it has ever accessed the node's parent. This property makes it possible to run the Frederickson's algorithm without first explicitly building the entire heap in the memory as long as we have a way to obtain the information of a node given the information of its parent.

We sketch an $O(k \log\log k)$-time algorithm [34] for enumerating the k largest value nodes in a heap as follows. For simplicity, we assume all nodes in the heap have different values. A node is said to be of rank i if it is the i^{th} largest node. The algorithm runs by first finding a node u in the heap in $O(k \log\log k)$ time such that the rank of u is between k and ck for some constant c. Then the algorithm identifies all nodes in the heap not smaller than u in $O(ck) = O(k)$ time and returns the k largest nodes among them. To find u, we form at most $2\lceil k/\lfloor \log k \rfloor \rceil + 1$ groups of nodes, called clans. Each clan is of size at most $\lfloor \log k \rfloor$ and represented by the smallest node in it; representatives are managed in an auxiliary heap. We form the first clan C_1 in $O(\log k \log\log k)$ time by grouping the largest $\lfloor \log k \rfloor$ nodes in the original heap and initialize the auxiliary heap with the representative of C_1. Set the *offspring* $os(C_1)$ of C_1 to the set of nodes in the original heap which are children of C_1 but not in C_1, and set the poor relations $pr(C_1)$ of C_1 to the empty set. Then for i from 1 to $\lfloor \log k \rfloor$, do the following. Extract the largest element in the auxiliary heap and let C_j be the clan represented

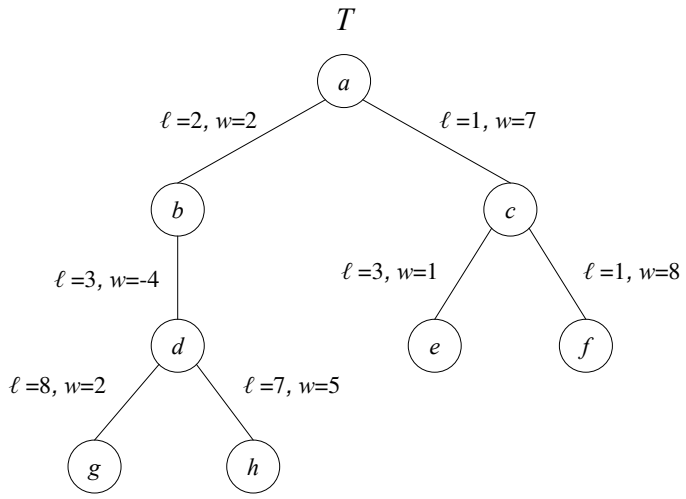


Figure 7.1: A tree T associated with a edge length function ℓ and a edge weight function w.

by the element extracted. If $os(C_j)$ is not empty, then form a new clan C_{i+1} in $O(\log k \log\log k)$ time by grouping the $\lfloor \log k \rfloor$ largest nodes from the subheaps rooted at $os(C_j)$ in the original heap. Insert the representative of C_{i+1} into the auxiliary heap. Set $os(C_{i+1})$ to the group of nodes in the original heap which are children of C_{i+1} but not in C_{i+1}, and set $pr(C_{i+1})$ to the group of nodes which are members of $os(C_j)$ but not included in C_{i+1}. If $pr(C_j)$ is not empty, then form a new clan C_{i+2} in $O(\log k \log\log k)$ time by grouping the $\lceil k/\lfloor \log k \rfloor \rceil$ largest nodes from the subheaps rooted at $pr(C_j)$ in the original heap. Insert the representative of C_{i+2} into the auxiliary heap. Set $os(C_{i+1})$ to the group of nodes in the original heap which are children of C_{i+2} but not in C_{i+2}, and set $pr(C_{i+2})$ to the group of nodes which are members of $pr(C_j)$ but not included in C_{i+2}. When the loop terminates, set u to the last element extracted from the auxiliary heap. Since at most $2\lceil k/\lfloor \log k \rfloor \rceil + 1$ clans are formed and each clan can be formed in $O(\log k \log\log k)$ time, the total time is $O(k \log\log k)$.

By applying the above approach recursively, plus some speed-up techniques, Frederickson [34] obtained an $O(k)$-time algorithm.

Theorem 7.3: [Frederickson [34]] For any constant D, we can find the k largest nodes in any D-heap, in $O(k)$ time.

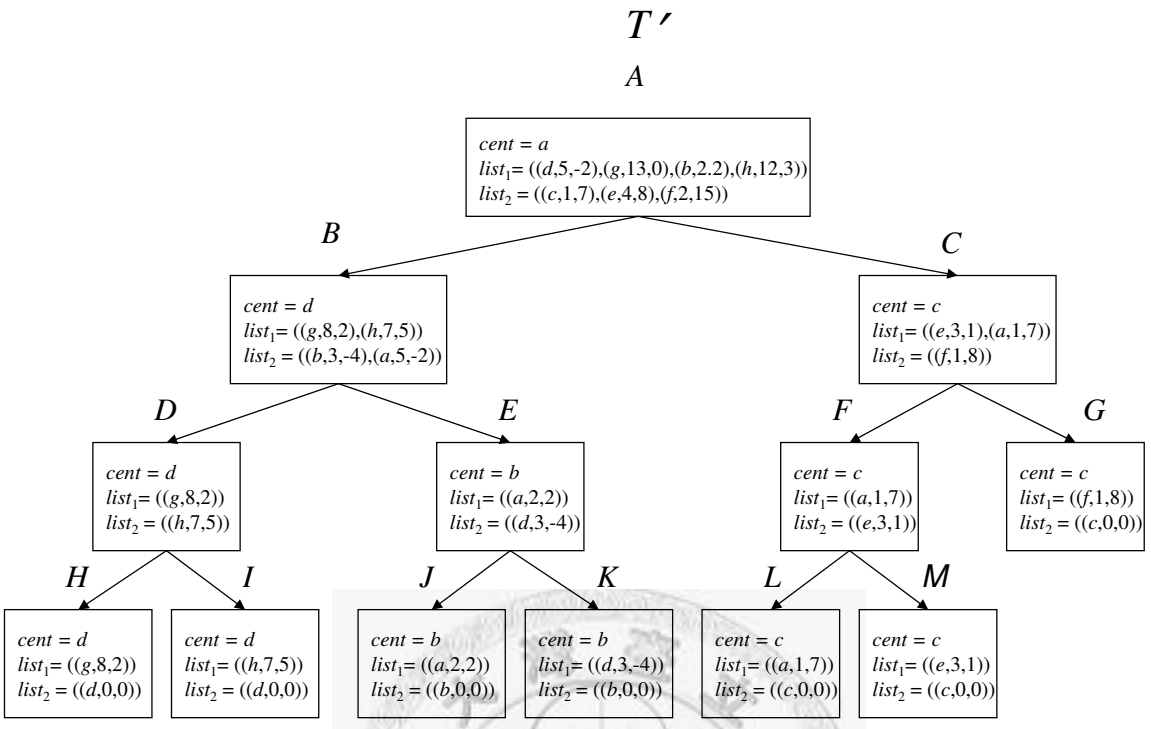


Figure 7.2: A centroid decomposition tree T' of the tree in Figure 7.1.

7.2 An $O(|V|\log|V|+k)$-Time Algorithm

For simplicity, we only consider paths with at least two vertices, and we don't distinguish between the path from u to v and the path from v to u, i.e., the path from u to v and the path from v to u are considered the same. Thus each path is uniquely determined by the unordered pair of its end vertices. We define the length and weight of an unordered pair $\{u, v\}$ to be the length and weight of the path from u to v respectively. We say an unordered pair $\{u, v\}$ of vertices is legal if and only if its weight is in the interval $[L, U]$. So our task becomes to find the k longest legal unordered pairs of vertices in T.

We sketch the main idea of our algorithm before describing the details. We find all legal unordered pairs of vertices in T by dividing T into two subtrees T_1 and T_2 of roughly the same size and uniting all the legal unordered pairs of vertices $\{u, v\}$ satisfying $u \in V(T_1)$ and $v \in V(T_2)$ with those recursively computed legal unordered pairs of vertices in T_1 and T_2. After finishing this recursive process, we have all legal unordered pairs of vertices in T. We then build a heap consisting of all legal unordered pairs of vertices in T and find the k longest unordered

pairs in this heap by applying the Frederickson's algorithm [34]. The major difficulty is that the number of legal unordered pairs of vertices in T may be much larger than $|V|\log|V|+k$. Thus we have to represent the set of all legal unordered pairs of vertices in T in a succinct way such that we are still able to build an implicit representation of the heap stated above and run the Frederickson's algorithm [34] on this implicitly-represented heap without loss of efficiency.

We now describe our algorithm in details. First, we construct a centroid decomposition tree $T'=(V',E',r)$ of T in $O(|V|\log|V|)$ time by Theorem 7.1. For each $v\in V'$ and $i\in\{1,2\}$, let $(v_{i,j},\ell(v.cent,v_{i,j}),w(v.cent,v_{i,j}))$ be the j^{th} element of $v.list_i$ if it exists. Note that since $\sum_{v\in V'}(|v.list_1|+|v.list_2|+1)=O(|V|\log|V|)$, we can find $\ell(v.cent,v_{i,j})$ and $w(v.cent,v_{i,j})$ for all $v\in V'$, $i\in\{1,2\}$ and $1\leq j\leq|v.list_i|$ in total $O(|V|\log|V|)$ time. By the next lemma, in total $O(|V|\log|V|)$ time, for each $v\in V'$ and $1\leq i\leq|v.list_1|$ we can find an interval $[p_i^v,q_i^v]$ such that $w(v.cent,v_{1,i})+w(v.cent,v_{2,j})=w(v_{1,i},v_{2,j})\in[L,U]$ for all $j\in[p_i^v,q_i^v]$ and $w(v_{1,i},v_{2,j})\notin[L,U]$ for all $j\notin[p_i^v,q_i^v]$. Note that the set of all legal unordered pairs of vertices in T is equal to the set $\bigcup_{v\in V'}\bigcup_{i=1}^{|v.list_1|}\{\{v_{1,i},v_{2,j}\}:j\in[p_i^v,q_i^v]\}$.

Lemma 7.1: Let $T'=(V',E',r)$ be a centroid decomposition tree of $T=(V,E)$. In total $O(|V|\log|V|)$ time, for all $v\in V'$ and $1\leq i\leq|v.list_1|$, we can find an interval $[p_i^v,q_i^v]$ such that $w(v_{1,i},v_{2,j})\in[L,U]$ for all $j\in[p_i^v,q_i^v]$ and $w(v_{1,i},v_{2,j})\notin[L,U]$ for all $j\notin[p_i^v,q_i^v]$.

Proof: Since $\sum_{v\in V'}(|v.list_1|+|v.list_2|+1)=O(|V|\log|V|)$, we only have to show that for each $v\in V'$, we can compute $[p_i^v,q_i^v]$ for all $1\leq i\leq|v.list_1|$ in total $O(|v.list_1|+|v.list_2|+1)$ time. Given $v\in V'$, we claim the following procedure computes $[p_i^v,q_i^v]$ for all $1\leq i\leq|v.list_1|$ in total $O(|v.list_1|+|v.list_2|+1)$ time.

1. Let $n=|v.list_1|$ and $m=|v.list_2|$.
2. If $n=0$ or $m=0$ then stop.
3. Set p and q to m.
4. For $i\leftarrow 1$ to n do
   (a) While$(w(v_{1,i},v_{2,p-1})\geq L$ and $p-1\geq 1)$ do $p\leftarrow p-1$.
   (b) While$(w(v_{1,i},v_{2,q})>U$ and $q\geq p)$ do $q\leftarrow q-1$.

(c) $p_i^v \leftarrow p$ and $q_i^v \leftarrow q$.

It is not hard to see the running time of this procedure is $O(|v.list_1| + |v.list_2| + 1)$ since both the values of p and q are nonincreasing. To verify the correctness, it suffices to note that since the list $v.list_i$, $i \in \{1, 2\}$, is sorted on $w(v.cent, v_{i,j})$, the sequence $(p_1^v, \ldots, p_{|v.list_1|}^v)$ and the sequence $(q_1^v, \ldots, q_{|v.list_1|}^v)$ must be nonincreasing. □

Then for each $v \in V'$, we preprocess $v.list_2$ so that given any interval $[i, j]$, we can find the index k, denoted RMQ$(v.list_2, i, j)$, in $[i, j]$ such that $\ell(v.cent, v_{2,k})$ achieves maximum in $O(1)$ time. By Theorem 7.2, this preprocessing can be done in $O(\sum_{v \in V'} |v.list_2|) = O(|V| \log |V|)$ time.

We next turn to define some data structures first. For each $v \in V'$ and $1 \leq i \leq |v.list_1|$, define $H(v_{1,i})$ to be a rooted ordered binary tree in which each node contains three fields *pair*, *value*, and *interval*, satisfying the following properties: (1) There are total $|v.list_1|$ nodes in $H(v_{1,i})$ and the interval of the root of $H(v_{1,i})$ is $[p_i^v, q_i^v]$, (2) for each node u of $H(v_{1,i})$, if $p < k$ then u's left child has interval $[p, k-1]$, and if $k < q$ then u's right child has interval $[k+1, q]$, where $[p, q] = u.interval$ and $k =$RMQ$(v.list_2, p, q)$, and (3) for each node u of $H(v_{1,i})$, if $u.interval = [p, q]$ then $u.pair = \{v_{1,i}, v_{2,k}\}$ and $u.value = \ell(v_{1,i}, v_{2,k})$, where $k =$RMQ$(v.list_2, p, q)$.

Let us now return to describe our algorithm. Let $V(H(v_{1,i}))$ denote the set of nodes in $H(v_{1,i})$. Since the set of all legal unordered pairs of vertices in T is equal to the set $\bigcup_{v \in V'} \bigcup_{i=1}^{|v.list_1|} \{\{v_{1,i}, v_{2,j}\} | j \in [p_i^v, q_i^v]\} = \bigcup_{v \in V'} \bigcup_{i=1}^{|v.list_1|} \{u.pair | u \in V(H(v_i))\}$, the remaining work becomes to find the k largest value nodes in $\bigcup_{v \in V'} \bigcup_{i=1}^{|v.list_1|} V(H(v_{1,i}))$. Clearly, we can not afford to construct $H(v_{1,i})$ explicitly for each $v_{1,i}$. But notice that given any node u of $H(v_{1,i})$, we can always construct u's children in $O(1)$ time since we have done the RMQ preprocessing on the list $v.list_2$. Thus we only construct the root of $H(v_{1,i})$ in the first instance and expand the tree as needed. Since we have known p_i^v and q_i^v for each $v \in V'$ and $1 \leq i \leq |v.list_1|$ and done the RMQ preprocessing on the list $v.list_2$ for each $v \in V'$, we can construct, in total $O(|V| \log |V|)$ time, the root of $H(v_{1,i})$ for each $v_{1,i}$. Then we place these roots into a balanced 2-heap by the *heapify* operation [26] in linear time, i.e., in $O(|V| \log |V|)$ time. Note that each $H(v_{1,i})$ is a 2-heap, so we have conceptually built a 4-heap for the set $\bigcup_{v \in V'} \bigcup_{i=1}^{|v.list_1|} V(H(v_{1,i}))$.

Now by Theorem 7.3, we can apply Frederickson's algorithm [34] to find the k largest value nodes in that 4-heap in $O(k)$ time. Of course, except the roots of all $H(v_{1,i})$, all the nodes in that 4-heap are not physically created until they are needed in running Frederickson's [34] algorithm. We summarize this section by the following theorem.

Theorem 7.4: Let $T = (V, E)$ be a tree with a length function $\ell : E \to \mathbb{R}$ and a weight function $w : E \to \mathbb{R}$. Given T, a positive integer k and two numbers L, U with $L \leq U$, we can find the k longest paths among all paths in T with weights in the interval $[L, U]$ in $O(|V| \log |V| + k)$ time.

7.3 An $\Omega(|V| \log |V| + k)$ Lower Bound

We prove that the Weight-Constrained Longest Path Problem has an $\Omega(|V| \log |V|)$ bound in the algebraic computation tree model. It follows that the Weight-Constrained k Longest Paths Problem has an $\Omega(|V| \log |V| + k)$ lower bound in the algebraic computation tree model since extra $\Omega(k)$ time is necessary for outputting the answer.

Definition 7.4: [Set Intersection Problem] Given two sets $\{x_1, x_2, \ldots, x_n\}$ and $\{y_1, y_2, \ldots, y_n\}$, the Set Intersection Problem asks whether there exist indices i and j such that $x_i = y_j$.

Lemma 7.2: [Ben-Or [13]] The Set Intersection Problem has an $\Omega(n \log n)$ lower bound in the algebraic computation tree model.

Theorem 7.5: The Weight-Constrained Longest Path Problem has an $\Omega(|V| \log |V|)$ lower bound in the algebraic computation tree model.

Proof: We reduce the Set Intersection Problem to the Weight-Constrained Longest Path Problem. Given two sets $\{x_1, x_2, \ldots, x_n\}$ and $\{y_1, y_2, \ldots, y_n\}$, we construct, in $O(n)$ time, a problem instance of the Weight-Constrained Longest Path Problem as follows. We first construct a tree $T = (V, E)$, where $V = \{x'_1, \ldots, x'_n\} \cup \{y'_1, \ldots, y'_n\} \cup \{c_1, c_2\}$ and $E = \{\overline{x'_1 c_1}, \ldots, \overline{x'_n c_1}\} \cup \{\overline{y'_1 c_2}, \ldots, \overline{y'_n c_2}\} \cup \{\overline{c_1 c_2}\}$. Define the length function $\ell : E \to \mathbb{R}$ by letting $\ell(e) = 1$ for all $e \in E$. Define the weight function $w : E \to \mathbb{R}$ by letting $w(\overline{x'_i c_1}) = x_i$ and $w(\overline{y'_i c_2}) = -y_i$ for all $i = 1, \ldots, n$, and $w(\overline{c_1 c_2}) = 0$. Set both the weight lower bound

L and the weight upper bound U of paths to 0. It can be verified that the longest path in T with weight $= 0$ has length 3 if and only if there exist indices i and j such that $x_i = y_j$. Since in this reduction we have $|V| = 2n + 2$ and the SET INTERSECTION PROBLEM has an $\Omega(n \log n)$ in the algebraic computation tree model by Lemma 7.2, we conclude that the WEIGHT-CONSTRAINED LONGEST PATH PROBLEM has an $\Omega(|V| \log |V|)$ lower bound in the algebraic computation tree model. □

Corollary 7.1: The WEIGHT-CONSTRAINED k LONGEST PATHS PROBLEM has an $\Omega(|V| \log |V| + k)$ lower bound in the algebraic computation tree model.

7.4 An Application of Finding the Sum-Constrained k Longest Segments

In biological sequence analysis, several researchers have devoted to the problem of finding the longest segment whose sum is not less than a specified lower bound L [3, 20, 73]. Allison [3] gave an algorithm which runs in linear time if the input sequence is a 0-1 sequence and L is a rational number. For real number sequences and real number lower bound, Wang and Xu [73] provided the first linear time algorithm, and Chen and Chao [20] gave an alternative linear time algorithm which runs in an online manner. We consider a more general problem in which both the lower bound L and the upper bound U of the sums of the segments are given and we want to find the k longest segments whose sums satisfy both the lower bound condition and the upper bound condition.

Theorem 7.6: Given a sequence $A = (a_1, a_2, \ldots, a_n)$ of real numbers and an interval $[L, U]$, we can find, in $O(n \log n + k)$ time, the k longest segments whose sums are in the interval $[L, U]$.

Proof: Directly from Theorem 7.4. □

Part III

Selection Problems

Chapter 8

Length-Constrained Sum Selection

Given a sequence $A = (a_1, a_2, \ldots, a_n)$ of $n > 1$ real numbers, and two positive integers L, U with $1 \leq L < U \leq n$, a segment $A[i, j]$ is said to be feasible if and only if its length is in $[L, U]$. The Length-Constrained Sum Selection Problem, for a given k, is to find the k^{th} largest sum among all sums of feasible segments of A.

When there are no length constraints, i.e., $L = 1$ and $U = n$, the Length-Constrained Sum Selection Problem becomes the Sum Selection Problem. Bengtsson and Chen [11] first studied the Sum Selection Problem and gave an $O(n \log^2 n)$-time algorithm for it. Recently, Lin and Lee provided an $O(n \log n)$-time algorithm [54] for the Sum Selection Problem and an expected $O(n \log(U-L+1))$-time randomized algorithm [55] for the Length-Constrained Sum Selection Problem.

In this chapter, we study the Length-Constrained Sum Selection Problem and provide an $O(n \log(U - L + 1))$-time algorithm for it. In addition, we prove that the Sum Selection Problem has an $\Omega(n \log n)$ lower bound in terms of n, which answers the question raised by Lin and Lee [54].

8.1 Preliminaries

A matrix $X \in \mathbb{R}^{n \times m}$ is said to be *sorted* if the values of each row and each column are in nondecreasing order. From the results of Frederickson and Johnson [37], we have the following theorem.

Theorem 8.1: The selection problem in a collection of sorted matrices is given a rank k and a collection of sorted matrices $\{X_1, \ldots, X_N\}$ in which X_j has dimensions $n_j \times m_j$, $n_j \geq m_j$, to find the k^{th} largest element among all elements of sorted matrices in $\{X_1, \ldots, X_N\}$. This problem can be solved in $O(\sum_{j=1}^{N} m_j \log(2n_j/m_j))$ time.

8.2 Subroutine

Let $P, Q \subseteq \mathbb{R}^2$ be two n-point multisets and $Ar \geq b$ be a set of λ inequalities on x and y, where $A \in \mathbb{R}^{\lambda \times 2}$, $r = \binom{x}{y}$, and $b \in \mathbb{R}^{\lambda}$. Define the *constrained Minkowski sum* $(P \oplus Q)_{Ar \geq b}$ as the multiset

$$\{(p+q) | p \in P, q \in Q, A(p+q) \geq b\}.$$

Given P, Q, and a linear inequality $L : x \geq c$, we next describe a divide-and-conquer approach to store the y-coordinates of $(P \oplus Q)_{x \geq c}$ as a collection of sorted matrices in $O(n \log n)$ time. For ease of exposition, we assume that no two points in P and Q have the same x-coordinate and n is a power of two. The algorithm proceeds as follows. Assume P and Q have been presorted into P_x and Q_x (P_y and Q_y, respectively) in nondecreasing order of x-coordinates (y-coordinates, respectively). Let $P_x = ((x_1, y_1), \ldots, (x_n, y_n))$, $Q_x = ((\bar{x}_1, \bar{y}_1), \ldots, (\bar{x}_n, \bar{y}_n))$, $P_y = ((x'_1, y'_1), \ldots, (x'_n, y'_n))$, and $Q_y = ((\bar{x}'_1, \bar{y}'_1), \ldots, (\bar{x}'_n, \bar{y}'_n))$. First, we divide P_x into two halves of equal size: $A = ((x_1, y_1), \ldots, (x_{n/2}, y_{n/2}))$ and $B = ((x_{n/2+1}, y_{n/2+1}), \ldots, (x_n, y_n))$. Next, we find a point $(\bar{x}_t, \bar{y}_t)$ of Q_x such that $x_{n/2} + \bar{x}_t < c$ and t is maximized and divide Q_x into two halves: $C = ((\bar{x}_1, \bar{y}_1), \ldots, (\bar{x}_t, \bar{y}_t))$ and $D = ((\bar{x}_{t+1}, \bar{y}_{t+1}), \ldots, (\bar{x}_n, \bar{y}_n))$. The point set $(P \oplus Q)_{x \geq c}$ is the union of $(A \oplus C)_{x \geq c}$, $(A \oplus D)_{x \geq c}$, $(B \oplus C)_{x \geq c}$, and $(B \oplus D)_{x \geq c}$. Because $\bar{x}_t$ is the largest x-coordinate among all x-coordinates of points in Q_x such that $x_{n/2} + \bar{x}_t < c$, we know that the points in $A \oplus C$ cannot satisfy the constraint $x \geq c$. Hence, we only need to consider points in $A \oplus D$, $B \oplus C$, and $B \oplus D$. Because P_x and Q_x are in nondecreasing order of x-coordinates, it is guaranteed that all points in $B \oplus D$ satisfy the constraint L, i.e., $B \oplus D = (B \oplus D)_{x \geq c}$. Construct in linear time $row\_vector = (r_1, r_2, \ldots, r_{n/2})$ which is the y-coordinates in the subsequence of P_y resulting from removing points with x-coordinates no greater than $x_{n/2}$ from P_y. Construct in linear time $column\_vector = (c_1, c_2, \ldots, c_{n-t})$ which is the y-coordinates in the subsequence of Q_y resulting from removing points with x-coordinates

no greater than $\bar{x}_t$ from Q_y. Note *row_vector* is the same as the result of sorting B into non-decreasing order of y-coordinates, and *column_vector* is the same as the result of sorting D into non-decreasing order of y-coordinates. Thus, we have $\{y : (x, y) \in (B \oplus D)_{x \geq c}\} = \{y : (x, y) \in B \oplus D\} = \{r_i + c_j : 1 \leq i \leq |row\_vector|, 1 \leq j \leq |column\_vector|\}$. Therefore, we can store the y-coordinates of $B \oplus D = (B \oplus D)_{x \geq c}$ as a sorted matrix X of dimensions $|row\_vector| \times |column\_vector|$ where the (i, j)-th element of X is $r_i + c_j$. Observe that it is not necessary to explicitly construct the sorted matrix X, which needs $\Omega(n^2)$ time. Because the (i, j)-th element of X can be obtained by summing up r_i and c_j, we only need to keep *row_vector* and *column_vector*. The rest is to construct the sorted matrices for the y-coordinates of points in $(A \oplus D)_{x \geq c}$ and $(B \oplus C)_{x \geq c}$. It is accomplished by applying the above approach recursively. The pseudocode is shown in Figure 8.1. We now analyze the time complexity.

Lemma 8.1: Given a matrix $X \in \mathbb{R}^{\mathfrak{N} \times \mathfrak{M}}$, we define the *side length* of X to be $\mathfrak{N} + \mathfrak{M}$. Letting $T(n', m')$ be the time complexity of running ConstructMatrices(P_x, Q_x, P_y, Q_y, L), where $n' = |P_x| = |P_y|$ and $m' = |Q_x| = |Q_y|$, we have $T(n', m') = O((n' + m') \log(n' + 1))$. Similarly, letting $M(n', m')$ be the sum of the side lengths of all sorted matrices created by running ConstructMatrices(P_x, Q_x, P_y, Q_y, L), we have $M(n', m') = O((n' + m') \log(n' + 1))$.

Proof: It suffices to prove that $T(n', m') = O((n' + m') \log(n' + 1))$. By Algorithm Construct-Matrices in Figure 8.1, we have $T(n', m') \leq \max_{0 \leq i \leq m'}\{c'(n' + m') + T(n'/2, i) + T(n'/2, m' - i)\}$ for some constant c'. Then by induction on n', it is easy to prove that $T(n', m')$ is $O((n' + m') \log(n' + 1))$. □

8.3 An $O(n \log(U - L + 1))$-Time Algorithm

Let $P = \{p_0, p_1, \ldots, p_n\}$ and $Q = \{q_0, q_{-1}, \ldots, q_{-n}\}$, where $p_i = (x_i, y_i) = (i, \sum_{t=1}^{i} a_t)$ and $q_{-i} = (\overline{x}_i, \overline{y}_i) = (-i, -\sum_{t=1}^{i} a_t)$ for all $i = 0, 1, \ldots, n$. A point (x, y) in $P \oplus Q$ is said to be a feasible point if and only if $L \leq x \leq U$. Note that each feasible segment $A[i, j]$ corresponds to a feasible point $(x, y) = p_j + q_{1-i}$ in $P \oplus Q$. Thus, the Length-Constrained Sum Selection Problem is equivalent to finding the k^{th} largest y-coordinate among all y-coordinates of feasible

points in $P \oplus Q$. We next show how to do this in $O(n \log(U - L + 1))$ time. For simplicity, we assume n is a multiple of $U - L$.

1. Let $i_t = t(U - L)$ and $j_t = L - t(U - L)$ for $t = 0, 1, \ldots, \frac{n}{U-L}$.
2. **for** $t \leftarrow 0$ to $\frac{n}{U-L}$ **do**
   (a) Let P_t=$\{p_h \in P : i_t - (U - L) < h \leq i_t\}$ and $Q_t = Q_{t,1} \cup Q_{t,2}$, where $Q_{t,1} = \{q_h \in Q : j_t \leq h < j_t + (U - L)\}$ and $Q_{t,2} = \{q_h \in Q : j_t + (U - L) \leq h < j_t + 2(U - L)\}$.
   (b) Store the y-coordinates of points in $(P_t \oplus Q_{t,1})_{x \geq L}$ as a set $N_{t,1}$ of sorted matrices such that the sum of side lengths of the sorted matrices in $N_{t,1}$ is no greater than $c \cdot ((|P_t| + |Q_{t,1}|) \log(|P_t| + |Q_{t,1}| + 1))$ for some constant c.
   (c) Store the y-coordinates of points in $(P_t \oplus Q_{t,2})_{x \leq U}$ as a set $N_{t,2}$ of sorted matrices such that the sum of side lengths of the sorted matrices in $N_{t,2}$ is no greater than $c \cdot ((|P_t| + |Q_{t,2}|) \log(|P_t| + |Q_{t,2}| + 1))$ for some constant c.
3. **return** the k^{th} largest of the elements of sorted matrices in $\bigcup_{t=0}^{\frac{n}{(U-L)}} (N_{t,1} \cup N_{t,2})$.

The following lemma ensures the correctness.

Lemma 8.2: $(P \oplus Q)_{L \leq x \leq U} = \bigcup_{t=0}^{\frac{n}{(U-L)}} ((P_t \oplus Q_{t,1})_{L \leq x} \cup (P_t \oplus Q_{t,1})_{x \leq U})$.

Proof: We prove that $(P \oplus Q)_{L \leq x \leq U} \overset{(1)}{=} \bigcup_{t=0}^{\frac{n}{(U-L)}} (P_t \oplus Q)_{L \leq x \leq U} \overset{(2)}{=} \bigcup_{t=0}^{\frac{n}{(U-L)}} (P_t \oplus Q_t)_{L \leq x \leq U} \overset{(3)}{=} \bigcup_{t=0}^{\frac{n}{(U-L)}} ((P_t \oplus Q_{t,1})_{L \leq x \leq U} \cup (P_t \oplus Q_{t,2})_{L \leq x \leq U}) \overset{(4)}{=} \bigcup_{t=0}^{\frac{n}{(U-L)}} ((P_t \oplus Q_{t,1})_{L \leq x} \cup (P_t \oplus Q_{t,2})_{x \leq U})$. It is clear that equations (1) and (3) are true, so only equations (2) and (4) remain to be proved.

We first prove equation (2) by showing that $(P_t \oplus Q)_{L \leq x \leq U} = (P_t \oplus Q_t)_{L \leq x \leq U}$. Suppose for contradiction that there exist $p_i \in P_t$ and $q_j \notin Q_t$ such that $L \leq x_i + \overline{x}_j = i + j \leq U$. By $p_i \in P_t$, we have $(t - 1)(U - L) < i \leq t(U - L)$; by $q_j \notin Q_t$, we have either $j < L - t(U - L)$ or $j \geq L - (t - 2)(U - L)$. It follows that $i + j$ is either less than L or larger than U, a contradiction. To prove equation (4), it suffices to prove that all points in $(P_t \oplus Q_{t,1})$ must have x-coordinates less than U and all points in $(P_t \oplus Q_{t,2})$ must have x-coordinates larger than L. Let $p_i \in P_t$, $q_j \in Q_{t,1}$ and $q_{j'} \in Q_{t,2}$. It follows that $(t - 1)(U - L) < x_i = i \leq t(U - L)$, $L - t(U - L) \leq \overline{x}_j = j < L - (t - 1)(U - L)$, and $L - (t - 1)(U - L) \leq \overline{x}_{j'} = j' < L - (t - 2)(U - L)$. Thus, we have $x_i + \overline{x}_j = i + j < U$ and $L < x_i + \overline{x}_{j'} = i + j'$. □

Since $|P_t|, |Q_{t,1}|$, and $|Q_{t,2}|$ are no greater than $(U-L)$ for all t, each execution of Step 2.b and Step 2.c can be done in $O((U-L)\log(U-L+1))$ time by Lemma 8.1. There are total $\frac{n}{U-L}+1$ iterations of the **for**-loop in Step 2, so the total time spent on Step 2.b and Step 2.c is $O(n\log(U-L+1))$. The sum of side lengths of sorted matrices in $\bigcup_{t=0}^{\frac{n}{(U-L)}}(N_{t,1} \cup N_{t,2})$ is $O(\sum_{t=0}^{\frac{n}{U-L}}\sum_{i=1}^{2}(|P_t|+|Q_{t,i}|)\log(|P_t|+|Q_{t,i}|+1)) = O(\sum_{t=0}^{\frac{n}{U-L}}(U-L)\log(U-L+1)) = O(n\log(U-L+1))$. Therefore, by Theorem 8.1, Step 3 can be done in $O(n\log(U-L+2))$ time. Putting everything together, we have that the total running time is $O(n\log(U-L+1))$. The next theorem summarizes the results of this subsection.

Theorem 8.2: The Length-Constrained Sum Selection Problem can be solved in $O(n\log(U-L+1))$ time.

8.4 An Lower Bound for Sum Selection

Let X and Y be two sets of real numbers. Define the Cartesian sum of X and Y, denoted by $X+Y$, to be the multiset $\{x+y : x \in X \text{ and } y \in Y\}$.

Lemma 8.3: [Johnson and Kashdan [45]] Let $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ be two sets of real numbers. It has an $\Omega(n\log n)$ lower bound to find the median of $X+Y$.

Theorem 8.3: The Sum Selection Problem has an $\Omega(n\log n)$ lower bound in terms of n, where n is length of the input sequence.

Proof: Let $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ be two sets of real numbers. We reduce the problem of finding the median of $X+Y$ to the Sum Selection Problem. Without loss of generality we can assume x_i and y_i are positive for all $i = 1, \ldots, n$. Let $m = -(\max_{i=1}^{n} x_i + \max_{i=1}^{n} y_i)$. Construct a sequence $A = (-x_1, x_1, \ldots, -x_n, x_n, m, y_1, -y_1, \ldots, y_n, -y_n)$. It is easy to verify that $x_i + y_j$ is the median of $X+Y$ if and only if $(-x_i, x_i, \ldots, -x_n, x_n, m, y_1, -y_1, \ldots, y_j, -y_j)$ is the $(\frac{n^2}{2})^{th}$ smallest-sum segment of A. Since $|A| = 4n+1$, by Lemma 8.3, the Sum Selection Problem must have an $\Omega(n\log n)$ lower bound. □

Subroutine ConstructMatrices($P_x, Q_x, P_y, Q_y, L : x \geq c$)

Input: P_x and P_y are the results of sorting the multiset $P \subseteq \mathbb{R}^2$ in nondecreasing order of x-coordinates and y-coordinates, respectively. Q_x and Q_y are the results of sorting the multiset $Q \subseteq \mathbb{R}^2$ in nondecreasing order of x-coordinates and y-coordinates, respectively. A linear constraint L: $x \geq c$.

Goal: Store the y-coordinates of points in $(P \oplus Q)_{x \geq c}$ as a collection of sorted matrices.

1 $n' \leftarrow |P_x|$; $m' \leftarrow |Q_x|$.
2 Let $P_x = ((x_1, y_1), \ldots, (x_{n'}, y_{n'}))$, $Q_x = ((\bar{x}_1, \bar{y}_1), \ldots, (\bar{x}_{m'}, \bar{y}_{m'}))$,
  $P_y = ((x'_1, y'_1), \ldots, (x'_{n'}, y'_{n'}))$, and $Q_y = ((\bar{x}'_1, \bar{y}'_1), \ldots, (\bar{x}'_{m'}, \bar{y}'_{m'}))$.
3 $\bar{x}_0 \leftarrow -\infty$.
4 **if** $n' \leq 0$ or $m' \leq 0$ **then**
5 **return**
6 **if** $n' = 1$ or $m' = 1$ **then**
7 Scan points in $P_x \oplus Q_x$ to find all points satisfying L and construct the sorted matrix for y-coordinates of these points.
8 **return**
9 **for** $t \leftarrow m'$ downto 0 **do**
10 **if** $x_{n'/2} + \bar{x}_t < c$ **then**
11 *row_vector* $\leftarrow$ subsequence of P_y resulting from removing points with x-coordinates $\leq x_{n'/2}$.
12 *column_vector* $\leftarrow$ subsequence of Q_y resulting from removing points with x-coordinates $\leq \bar{x}_t$.
13 $A_x \leftarrow P_x[1, n'/2]$. $B_x \leftarrow P_x[n'/2 + 1, n']$. $C_x \leftarrow Q_x[1, t]$. $D_x \leftarrow Q_x[t + 1, m']$.
14 $A_y \leftarrow$ subsequence of P_y resulting from removing points with x-coordinates $> x_{n'/2}$.
15 $B_y \leftarrow$ subsequence of P_y resulting from removing points with x-coordinates $\leq x_{n'/2}$.
16 $C_y \leftarrow$ subsequence of Q_y resulting from removing points with x-coordinates $> \bar{x}_t$.
17 $D_y \leftarrow$ subsequence of Q_y resulting from removing points with x-coordinates $\leq \bar{x}_t$.
18 ConstructMatrices(A_x, D_x, A_y, D_y, L: $x \geq c$).
19 ConstructMatrices(B_x, C_x, B_y, C_y, L: $x \geq c$).
20 **return**
21 **end for**

Figure 8.1: Storing y-coordinates of $(P \oplus Q)_{x \geq c}$ as a collection of sorted matrices.

Part IV

Ordering Problems

Chapter 9

A Tight Analysis of the Katriel-Bodlaender Algorithm

A topological order *ord* of a directed acyclic graph (DAG) $G = (V, E)$ is a linear order of all its vertices such that if G contains an edge (u, v), then $ord(u) < ord(v)$. In this chapter we study an online variant of the topological ordering problem in which the edges of the DAG are given one at a time and we have to update the order *ord* each time an edge is added.

Alpern *et al.* [4] give an algorithm which takes $O(||\delta|| \log ||\delta||)$ time for each edge insertion, where $||\delta||$ measures the number of edges and nodes of a minimal subgraph that needs to be updated. (For a formal definition of $||\delta||$, please see [4, 65, 68].) Pearce and Kelly [65] propose a different algorithm which needs slightly more time to process an edge insertion in the worst case than the algorithm given by Alpern *et al.* [4], but show experimentally their algorithm perform well on sparse graphs.

Marchetti-Spaccamela *et al.* [60] give an algorithm which takes $O(mn)$ time for inserting m edges. Katriel [46] shows that the analysis is tight. Recently, Katriel and Bodlaender [47] modify the algorithm proposed by Alpern *et al.* [4], which is referred to as the Katriel-Bodlaender algorithm in this chapter. They prove that their algorithm has both an $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$ upper bound and an $\Omega(m^{3/2})$ lower bound on the running time for m edge insertions. This is the best amortized result for sparse graphs so far. They also analyze the complexity of their algorithm on structured graphs. They show that it runs in time $O(mk \log^2 n)$ where k is the treewidth of the underlying undirected graph and can be implemented to run in $O(n \log n)$

time on trees. On the other hand, Ajwani et al. [2] proposed an $O(n^{2.75})$-time algorithm, independent of the number of edges inserted. This is the best amortized result for dense graphs so far.

The only non-trivial lower bound is due to Ramalingam and Reps [68], who show that any algorithm needs $\Omega(n \log n)$ time while inserting $n-1$ edges in the worst case if all labels are maintained explicitly.

In this chapter, we prove that the Katriel-Bodlaender algorithm takes $\Theta(m^{3/2}+mn^{1/2}\log n)$ time for inserting m edges. By combining this with Ajwani *et al.*'s result [2], we get an upper bound of $O(\min\{m^{3/2}+mn^{1/2}\log n, n^{2.75}\})$ for online topological ordering. It is an improvement over the previous best upper bound of $O(\min\{m^{3/2}\log n, m^{3/2}+n^2\log n, n^{2.75}\})$.

The rest of this chapter is organized as follows. In Section 2, we describe how the Katriel-Bodlaender algorithm works, define notation and introduce some theorems proved in [47]. Section 3 proves that the Katriel-Bodlaender algorithm runs in $O(m^{3/2}+mn^{1/2}\log n)$ time, and Section 4 shows it needs $\Omega(m^{3/2}+mn^{1/2}\log n)$ time. Since the upper bound matches the lower bound, our analysis is tight. Section 5 summarizes our results.

9.1 The Katriel-Bodlaender Algorithm

The pseudo code of the Katriel-Bodlaender algorithm is given in Figure 9.1. The algorithm works as follows. The topological order of nodes is maintained by an *order data structure* ORD, which can maintain a total order and support the following operations in constant time [29, 9]:

- $InsertAfter(x,y)$ ($InsertBefore(x,y)$): Inserts x immediately after (before) y in the total order.
- $Delete(x)$: Removes x from the total order.
- $>_{ord}(x,y)$: Returns true if and only if x follows y in the total order.
- $Next(x)$ ($Prev(x)$): Returns the element that appears immediately after (before) x in the total order.

Initially the nodes are inserted into ORD in an arbitrary order. Each time a new edge $(Source, Target)$ arrives, $AddEdge(Source, Target)$ is called to insert the edge $(Source, Target)$

```
Function AddEdge(Source, Target)
 1 ToS ← []. FromT ← [].
 2 ToSNeighbors ← []. FromTNeighbors ← [].
 3 ToSIndegree ← 0. FromTOutdegree ← 0.
 4 s ← Source. t ← Target.
 5 while s >_ord t and s ≠ nil and t ≠ nil do
 6   m_s ← ToSIndegree. ℓ_s ← Indegree[s].
 7   m_t ← FromTOutdegree. ℓ_t ← Outdegree[t].
 8   if m_s + ℓ_s ≤ m_t + ℓ_t then
 9     ToS.Push(s).
10     foreach (w, s) ∈ E do ToSNeighbors.Insert(w).
11     ToSIndegree ← ToSIndegree + Indegree[s].
12     s ← ToSNeighbors.ExtractMax.
13   end if
14   if m_s + ℓ_s ≥ m_t + ℓ_t then
15     FromT.Push(t).
16     foreach (t, w) ∈ E do FromTNeighbors.Insert(w).
17     FromTOutdegree ← FromTOutdegree + Outdegree[t].
18     t ← FromTNeighbors.ExtractMin.
19   end if
20 end while
21 if s = nil then s ← ORD.Prev(Target).
22 if t = nil then t ← ORD.Next(Source).
23 while ToS.NotEmpty do
24   s' ← ToS.Pop.
25   ORD.Delete(s'). ORD.InsertAfter(s', s). s ← s'.
26 end while
27 while FromT.NotEmpty do
28   t' ← FromT.Pop.
29   ORD.Delete(t'). ORD.InsertBefore(t', t). t ← t'.
30 end while
31 E ← E ∪ (Source, Target). Outdegeree[Source]++. Indegree[Target]++.
```

Figure 9.1: The Katriel-Bodlaender algorithm for online topological ordering.

into the graph and update the total order maintained by ORD to a valid topological order for the modified graph.

It remains to describe how $AddEdge(Source, Target)$ operates. In each iteration of the first while loop, there is one node s which is a candidate for insertion into stack ToS (the

node with maximal rank in the current topological order which reaches a node in ToS but is not in ToS) and one node t which is a candidate for insertion into stack $FromT$ (the node with minimal rank in the current topological order which can be reached from a node in $FromT$ but is not in $FromT$). The algorithm always adds at least one of them into the relevant set. The way in which it decides which candidate(s) to add aims to balanced the number of edges outgoing from nodes in $FromT$ and the number of edges entering into nodes in ToS. That is, the algorithm always chooses a candidate so that the increase of $\max\{\sum_{v\in ToS} Indegree[v], \sum_{v\in FromeT} Outdegree[v]\}$ is fewer after adding the candidate into its relevant set. If a tie occurs, then both s and t are added into their relevant sets. If s is added to its relevant set ToS, all nodes which can reach s by one edge are inserted into $ToSNeighbors$ and then s is reset to the max element in $ToSNeighbors$. $ToSNeighbors$ is a priority queue maintaining all nodes which can reach nodes in ToS by one edges but is not in ToS. $ToSNeighbors$ is implemented by Fibonacci heaps [38] which can support insertions and extractions in $O(1)$ and $O(\log n)$ amortized time respectively. $ToSNeighbors$ determine the ranks of its elements according to the total order maintained by ORD. Similarly, if t is added to $FromT$, then all nodes which are reachable from t by one edge are inserted into $FromTNeighbors$ and then t is reset to the min element in $FromTNeighbors$. $FromTNeighbors$ is a priority queue maintaining all nodes which can be reached from nodes in $FromT$ by one edge but is not in $FromT$. $FromTNeighbors$ is also implemented by Fibonacci heaps and determine the ranks of its elements according to the total order maintained by ORD. The first while loop stops when $t >_{ord} s$ or any one of $ToSNeighbors$ and $FromTNeighbors$ is empty. If $ToSNeighbors$ ($FromTNeighbors$) is empty when the first while loop stops then s (t) is reset to $ORD.Prev(Target)$ ($ORD.Next(Source)$) before we update ORD. The update of ORD is carried out by fulfilling the following two tasks. First, delete all nodes in ToS from ORD and then insert them, in the same relative order among themselves, immediately after s. Secondly, delete all nodes in $FromT$ from ORD and then insert them, in the same relative order among themselves, immediately before t. After the update of ORD, the edge $(Source, Target)$ is inserted into the graph.

In the following, we define some notation. Let n and m be the number of nodes and edges in the DAG $G = (V, E)$ respectively. Let $G_i = (V, E_i)$ be the graph after the i^{th} edge insertion.

Let $Indegree_i[v]$ ($Outdegree_i[v]$) be the indegree (outdegree) of v in G_i. Let $FromT_i$ (ToS_i) denote the set of nodes in the stack $FromT$ (ToS) at the end of the first while loop upon the insertion of the i^{th} edge. Let s_i (t_i) denote the value of the variable s (t) at the end of the first while loop upon the insertion of the i^{th} edge. Let $T_i = \sum_{v \in FromT_i} Outdegree_{i-1}[v]$ and $S_i = \sum_{v \in ToS_i} Indegree_{i-1}[v]$. Let x_i denote $\max\{T_i, S_i\}$ and y_i denote $\max\{|ToS_i|, |FromT_i|\}$. Let $>_{ord_i}$ be the total order maintained by ORD after the i^{th} edge insertion. The following three theorems are proved in [47].

Theorem 9.1: The Katriel-Bodlaender algorithm needs $O(m^{3/2} + \sum_{1 \le i \le m} y_i \log n)$ time to insert m edges into an initially empty n-node graph.

Theorem 9.2: The Katriel-Bodlaender algorithm needs $\Omega(m^{3/2})$ time to insert m edges into an initially empty n-node graph.

Theorem 9.3: $Indegree_{i-1}[s_i] + S_i \ge x_i$ and $Outegree_{i-1}[t_i] + T_i \ge x_i$, for all i in $[1, m]$.

9.2 An $O(m^{3/2} + mn^{1/2} \log n)$ Upper Bound

In this section, we prove that the algorithm runs in time $O(m^{3/2} + mn^{1/2} \log n)$. By Theorem 9.1, we know the algorithm runs in time $O(m^{3/2} + \sum_{1 \le i \le m} y_i \log n)$, so we only have to show $\sum_{1 \le i \le m} y_i = O(mn^{1/2})$. For simplicity, we assume that $x_i \ge y_i$ for all i in $[1, m]$, although it should be $x_i \ge y_i - 1$ for all i in $[1, m]$.

An edge $e = (u, v)$ is called to be in front of (behind) a node w in G_i if and only if there is a path from v (w) to w (u) in G_i. A pair $(e, w) \in E \times V$ is called to be ordered in G_i if and only if e is either in front of or behind w in G_i. In the following proofs, we adopt one of the potential functions defined in [47]: The number of ordered pairs in $E \times V$. Let Φ_i denote the set $\{(e, w) \in E \times V : (e, w)$ is ordered in $G_i\}$, ϕ_i denote $|\Phi_i|$ and $\triangle\phi_i$ denote $\phi_i - \phi_{i-1}$.

Lemma 9.1: For all edges e incoming into ToS_i in G_{i-1} and for all nodes w in $FromT_i$, e is not in front of w in G_{i-1}.

Proof: Let $e = (u, v)$. Suppose for the contradiction that there is a path from v to w in G_{i-1}. It implies that $w >_{ord_{i-1}} v$. There are three cases to consider. Case 1: The iteration

in which variable s was assigned v is before the iteration in which variable t was assigned w in the i^{th} call of *AddEdge*. Since the nodes were assigned to variable s in decreasing order, we had $t >_{ord_{i-1}} s$ after variable t was assigned w and then left the loop. It contradicts to the assumption that w is in $FromT_i$. Case 2: The iteration in which variable t was assigned w is before the iteration in which variable s was assigned v in the i^{th} call of *AddEdge*. Since the nodes were assigned to variable t in increasing order, we had $t >_{ord_{i-1}} s$ after the variable s was assigned v and then left the loop. It contradicts to the assumption that v is in ToS_i. Case 3: Variable s and variable t was assigned v and w respectively at the same iteration in the i^{th} call of *AddEdge*. Since $w >_{ord_{i-1}} v$, we had $t >_{ord_{i-1}} s$ after variable t was assigned w and then left the loop. It contradicts to the assumption that w is in $FromT_i$ and v is in ToS_i. □

Lemma 9.1 states that all the S_i edges incoming into ToS_i are not in front of $FromT_i$ in G_{i-1}. Because all these S_i edges became in front of $FromT_i$ after the i^{th} edge insertion, we know $\triangle\phi_i \geq S_i \times |FromT_i|$. To pave the way for proving Lemma 9.5, we have to show $y_i^2 \leq \triangle\phi_i$ when $y_i = |FromT_i|$. If S_i was always larger than or equal to y_i when $y_i = |FromT_i|$, then we could jump to prove Lemma 9.5 directly. Since it is not the case, we need more lemmas. There are two cases to consider: First, $w <_{ord_{i-1}} s_i$ for all w in $FromT_i$, i.e., s_i is after $FromT_i$ in the total order $<_{ord_{i-1}}$. Second, some nodes in $FromT_i$ are after s_i in the total order $<_{ord_{i-1}}$. The following lemma deals with the first case.

Lemma 9.2: If $w <_{ord_{i-1}} s_i$ for all w in $FromT_i$ and $y_i = |FromT_i|$, then $y_i^2 \leq \triangle\phi_i$.

Proof: Since $w <_{ord_{i-1}} s_i$ for all w in $FromT_i$, we can deduce that all edges incident to s_i in G_{i-1} are not in front of w in G_{i-1} for all w in $FromT_i$. By combining this result with Lemma 9.1, we know there are at least $Indegree_{i-1}[s_i] + S_i$ edges not in front of w in G_{i-1} for all w in $FromT_i$. Because all these $Indegree_{i-1}[s_i] + S_i$ edges are in front of w in G_i for all w in $FromT_i$ and $y_i = |FromT_i|$, we can deduce that $(Indegree_{i-1}[s_i] + S_i) \times y_i \leq \triangle\phi_i$. By Theorem 9.3 and the assumption $y_i \leq x_i$, we have $y_i^2 \leq x_i \times y_i \leq (Indegree_{i-1}[s_i] + S_i) \times y_i \leq \triangle\phi_i$. □

The following lemma is used in the proof of Lemma 9.4 which deals with the second case.

Lemma 9.3: If there exists w in $FromT_i$ such that $w >_{ord_{i-1}} s_i$, then, in the i^{th} call of *AddEdge*, the iteration in which variable t was assigned t_i is not after the iteration in which variable s was assigned s_i.

Proof: Suppose for the contradiction that the iteration in which variable s was assigned s_i is before the iteration in which variable t was assigned t_i. Let $\hat{t}$ be the last element pushed into $FromT_i$. Consider the iteration in which variable t was assigned t_i. At the beginning, the value of variable s was s_i and the value of variable t was $\hat{t}$. Since $\hat{t} >_{ord_{i-1}} s_i$, we failed in the test condition and left the loop. Thus, $\hat{t}$ was not pushed into $FromT_i$, a contradiction. □

Lemma 9.4: If there exists w in $FromT_i$ such that $w >_{ord_{i-1}} s_i$ and $y_i = |FromT_i|$, then $y_i^2 \leq \triangle\phi_i$.

Proof: Consider the iteration in which variable t was assigned value t_i in the i^{th} call of *AddEdge*. The value of $m_t + \ell_t$ was equal to T_i. By Lemma 9.3, we know the value of variable s was not s_i when line 6 was executed, so $m_s + \ell_s \leq S_i$. Since variable t was selected to be assigned a new value, we know $m_t + \ell_t \leq m_s + \ell_s$. By combining the results above, we get $T_i \leq S_i$. It implies that $S_i = x_i$. By Lemma 9.1, we know there are at least $S_i = x_i$ edges not in front of w in G_{i-1} for all w in $FromT_i$. Because all these x_i edges are in front of w in G_i for all w in $FromT_i$ and $y_i = |FromT_i|$, we can deduce that $x_i y_i \leq \triangle\phi_i$. By the assumption $x_i \geq y_i$, we have $y_i^2 \leq \triangle\phi_i$. □

Lemma 9.5: $\sum_{y_i=|FromT_i|} y_i^2 \leq mn$.

Proof: By Lemma 9.2 and Lemma 9.4, we know $y_i^2 \leq \triangle\phi_i$ if $y_i = |FromT_i|$. Since $\phi_0 = 0$, $\phi_m \leq mn$, $\triangle\phi_i \geq 0$, and $y_i^2 \leq \triangle\phi_i$ if $y_i = |FromT_i|$, we can deduce that $\sum_{y_i=|FromT_i|} y_i^2 \leq \sum_{1\leq i\leq m} \triangle\phi_i \leq mn$. □

The following lemma can be proved by an argument similar to the one for proving Lemma 9.5.

Lemma 9.6: $\sum_{y_i=|ToS_i|} y_i^2 \leq mn$.

Theorem 9.4: $\sum_{1 \le i \le m} y_i$ is $O(mn^{1/2})$.

Proof: By Lemma 9.5 and Lemma 9.6, we know $\sum_{1 \le i \le m} y_i^2 \le 2mn$. Since $\sum_{y_i < n^{1/2}} y_i \le mn^{1/2}$, we only have to show $\sum_{y_i \ge n^{1/2}} y_i$ is $O(mn^{1/2})$. Since $n^{1/2} \sum_{y_i \ge n^{1/2}} y_i \le \sum_{y_i \ge n^{1/2}} y_i^2 \le \sum_{1 \le i \le m} y_i^2 \le 2mn$, we have $\sum_{y_i \ge n^{1/2}} y_i \le 2mn^{1/2} = O(mn^{1/2})$. □

Theorem 9.5: The Katriel-Bodlaender algorithm needs $O(m^{3/2} + mn^{1/2} \log n)$ time to insert m edges into an initially empty n-node graph.

Proof: Theorem 9.1 states that the Katriel-Bodlaender algorithm needs $O(m^{3/2} + \sum_{1 \le i \le m} y_i \log n)$ time to insert m edges into an initially empty n-node graph. Theorem 9.4 states that $\sum_{1 \le i \le m} y_i$ is $O(mn^{1/2})$. By combining these two results, we know that the Katriel-Bodlaender algorithm needs $O(m^{3/2} + mn^{1/2} \log n)$ time to insert m edges into an initially empty n-node graph. □

9.3 An $\Omega(m^{3/2} + mn^{1/2} \log n)$ Lower Bound

In this section, we prove that the algorithm runs in time $\Omega(m^{3/2} + mn^{1/2} \log n)$.

Theorem 9.6: The Katriel-Bodlaender algorithm needs $\Omega(m^{3/2} + mn^{1/2} \log n)$ time to insert m edges into an initially empty n-node graph.

Proof: It is equivalent to showing that the algorithm needs $\Omega(\max\{m^{3/2} + mn^{1/2} \log n\})$ time to insert m edges into an initially empty n-node graph. Theorem 9.2 states that the algorithm needs $\Omega(m^{3/2})$ time to insert m edges into an initially empty n-node graph. Since $m^{3/2} \ge mn^{1/2} \log n$ if and only if $m \ge n \log^2 n$, we only have to show that the algorithm needs $\Omega(mn^{1/2} \log n)$ time if $m \le n \log^2 n$. Without loss of generality we assume that $m \ge n$. In the following, we describe an input which takes the algorithm $\Omega(mn^{1/2} \log n)$ time to process if $n \le m \le n \log^2 n$. For simplicity, we assume that both n and m are exact powers of 16.

Let $\{v_0, v_2, \ldots, v_{n-1}\}$ be the nodes of the DAG sorted by the order maintained by ORD before edge insertions. Let $u_i = v_{\frac{n}{4}+i}$ for $i = 0, \ldots, \frac{3n}{4} - 1$. Define P_i to be $(v_{\frac{(i-1)n^{1/2}}{4}}, v_{\frac{(i-1)n^{1/2}}{4}+1}, \ldots, v_{\frac{in^{1/2}}{4}-1})$,

for $i = 1, \ldots, n^{1/2}$. Define Q_i to be $(u_{(i-1)n^{1/4}}, u_{(i-1)n^{1/4}+1}, \ldots, u_{in^{1/4}-1})$, for $i = 1, \ldots, \frac{m}{2n^{1/2}}$. Let $Source_i = u_{in^{1/4}-1}$, i.e., the last node of Q_i, for $i = 1, \ldots, \frac{m}{2n^{1/2}}$. Let $Target_i = v_{\frac{(i-1)n^{1/2}}{4}}$, i.e., the first node of P_i, for $i = 1, \ldots, n^{1/2}$.

The input is composed of four parts.

Part 1. Construct $n^{1/2}$ identical subgraphs as in Figure 2(a) by inserting the edge $(v_{\frac{(i-1)n^{1/2}}{4}}, v_{\frac{(i-1)n^{1/2}}{4}+j})$ for all $i = 1, \ldots, n^{1/2}$ and $j = 1, \ldots, \frac{n^{1/2}}{4} - 1$. There are $n/4 - n^{1/2} < n/4 \leq m/4$ edge insertions in this part.

Part 2. Construct $\frac{m}{2n^{1/2}}$ identical subgraphs as in Figure 2(b) by inserting the edge $(u_{(i-1)n^{1/4}+j}, u_k)$ for all $i \in [1, \frac{m}{2n^{1/2}}]$, $j \in [0, n^{1/4} - 2]$ and $k \in ((i-1)n^{1/4} + j, in^{1/4})$. There are $\frac{m}{2n^{1/2}} \times \frac{n^{1/2}-n^{1/4}}{2} < m/4$ edge insertions in this part.

Part 3. This part is composed of $n^{1/2}$ rounds and there are $\frac{m}{2n^{1/2}}$ edge insertions in each round. In the i^{th} round, the following edges are inserted in their listed order: $\{(Source_1, Target_i)$, $(Source_2, Target_i)$,...,$(Source_{\frac{m}{2n^{1/2}}}, Target_i)\}$. There are $m/2$ edge insertions in this part. Figure 2(c) illustrates how the total order maintained by ORD changes when Part 3 arrives.

Part 4. Insert edges without causing cycles until there are m edges in the DAG.

Upon the insertion of edge $(Source_i, Target_j)$ in Part 3 for all i and j, all nodes in P_j are inserted into $FromTNeighbors$ at the same iteration and then extracted. Since there are $n^{1/2}/4$ nodes in P_j for all j, each edge insertion in Part 3 takes the algorithm $\Omega(n^{1/2} \log n)$ time to process. Because there are $m/2$ edges in Part 3, the total complexity is $\Omega(mn^{1/2} \log n)$. □

Theorem 9.7: The Katriel-Bodlaender algorithm needs $\Theta(m^{3/2} + mn^{1/2} \log n)$ time to insert m edges into an initially empty n-node graph..

Proof: It follows directly from Theorem 9.5 and Theorem 9.6. □

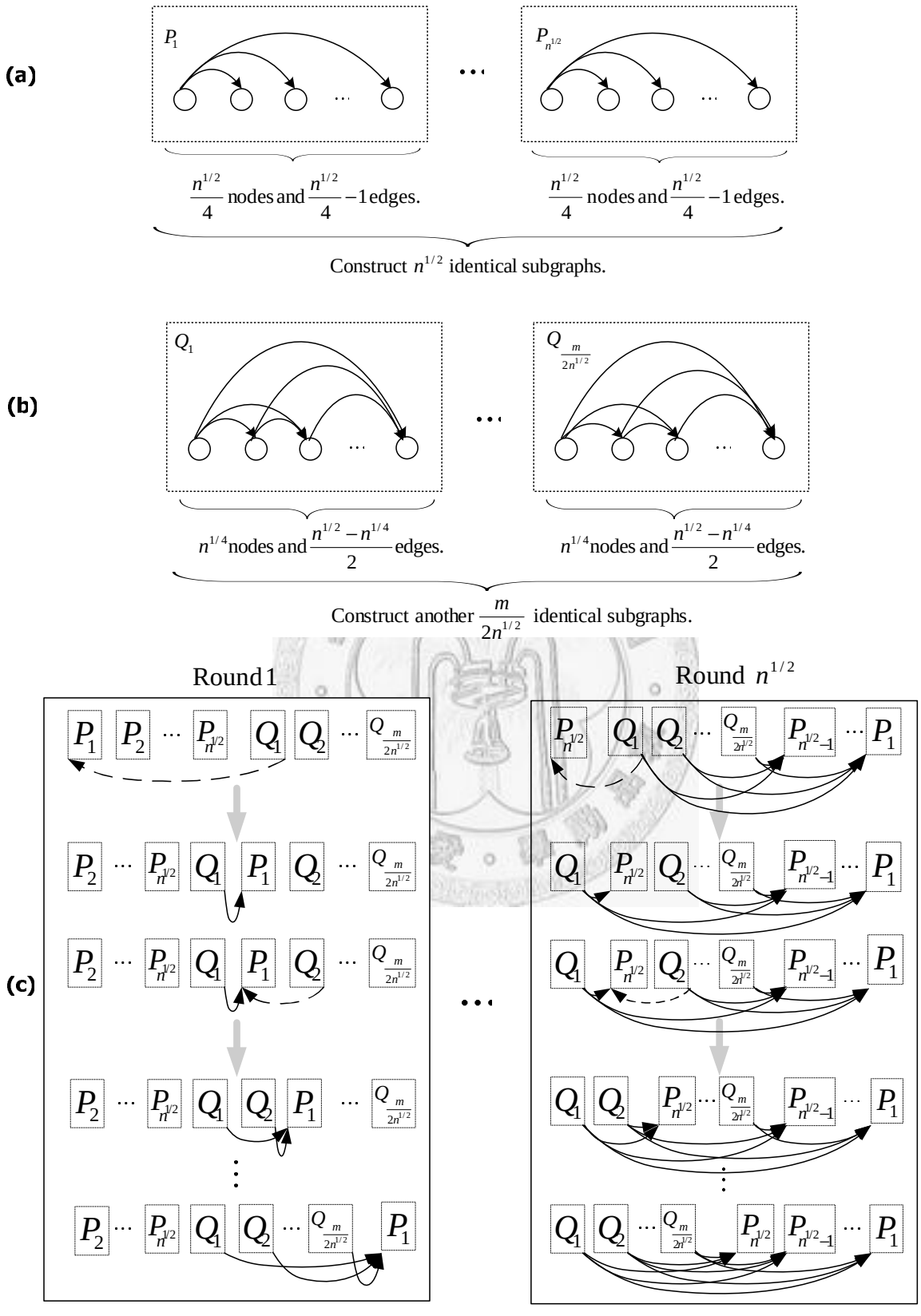


Figure 9.2: An input which requires $\Omega(mn^{1/2}\log n)$ time if $n \leq m \leq n\log^2 n$.

Chapter 10

An $\tilde{O}(n^{2.5})$-Time Algorithm for Online Topological Ordering

In this chapter we present an $\tilde{O}(n^{2.5})$-time algorithm for maintaining the topological order of a directed acyclic graph with n vertices while inserting m edges. This is an improvement over the previous result of $O(n^{2.75})$ by Ajwani, Friedrich, and Meyer.

10.1 Algorithm

We keep the current topological order as a bijective function $T : V \to [1..n]$. Let $d(u, v)$ denote $|T(v) - T(u)|$, $u \to v$ express that there is an edge from u to v, $u \rightsquigarrow v$ express that there is a path from u to v and $u < v$ be a short form of $T(u) < T(v)$. Let $n^{0.5} < t_0 < t_1 < t_2 < \cdots < t_{p-1} < t_p < t_{p+1} = n$, where $p = O(\log n)$ is a nonnegative integer. In Section 10.5, we show how to determine the values of these parameters.

Figur 10.1 gives the pseudo code of our algorithm. T is initialized with the topological order of the starting graph. Whenever an edge (u, v) is inserted into the graph, Insert(u, v) is called. If $u < v$, then Insert(u, v) does not change T and simply insert the edge into the graph. If $u > v$, then Insert(u, v) calls Reorder$(v, u, 0, 0)$ to update T such that T is still a valid topological order and $T(u) < T(v)$. After the call to Reorder$(v, u, 0, 0)$, Insert(u, v) can safely insert the edge into the graph.

It remains to explain how the procedure Reorder(u, v, f_1, f_2) works. The duty of the

procedure REORDER(u, v, f_1, f_2) is to update T such that T is still a valid topological order and $T(u) > T(v)$. The flag $f_1 = 1$ indicates that the set $A' = \{w : u \rightarrow w \text{ and } w \leq v\}$ has been known to be empty. The flag $f_2 = 1$ indicates that the set $B' = \{w : w \rightarrow v \text{ and } w \geq u\}$ has been known to be empty. If $T(u) > T(v)$, then we directly exit. Otherwise, there are two cases to consider:

1: $t_i < d(u, v) \leq t_{i+1}$ for some $i = 0, \ldots, p$. In this case, we first have to compute $\hat{A}_i = \{w : u \rightarrow w,\ d(u, w) \leq t_i, \text{ and } w < v\}$ and $\hat{B}_i = \{w : w \rightarrow v,\ d(w, v) \leq t_i, \text{ and } w > u\}$. If $\hat{A}_i = \emptyset$ and $f_1 = 0$, then we still have to compute $\hat{A}_{i+1} = \{w : u \rightarrow w,\ d(u, w) \leq t_{i+1}, \text{ and } w < v\}$ and set $A = \hat{A}_{i+1}$; otherwise, we directly set $A = \hat{A}_i$. Similarly, if $\hat{B}_i = \emptyset$ and $f_2 = 0$, then we still have to compute $\hat{B}_{i+1} = \{w : u \rightarrow w,\ d(u, w) \leq t_{i+1}, \text{ and } w < v\}$ and set $B = \hat{B}_{i+1}$; otherwise, we directly set $B = \hat{B}_i$.

2: $d(u, v) \leq t_0$. In this case we directly set $A = \hat{A}_0 = \{w : u \rightarrow w,\ d(u, w) \leq t_0, \text{ and } w < v\}$ and $B = \hat{B}_0 = \{w : w \rightarrow v,\ d(w, v) \leq t_0, \text{ and } w > u\}$.

If both A and B are empty, then we directly swap u and v and exit the procedure. Otherwise, let $T_{origianl}$ be the topological order at the start of the execution of the procedure. For each $u' \in \{u\} \cup A$, considered in order of decreasing $T_{original}(u')$, we do the following. For each $v' \in \{v' : v' \in B \cup \{v\} \text{ and } T_{original}(v') > T_{orginal}(u')\}$, considered in order of increasing $T_{original}(v')$, recursively call REORDER(u', v', f_1', f_2'). The first flag f_1' is set to 1 if and only if $u' = u$ and $A = \emptyset$, and the second flag f_2' is set to 1 if and only if $v' = v$ and $B = \emptyset$.

The idea behind the algorithm. Our algorithm broadly follows the algorithm by Ajwani *et al.* [2]. The main difference is that Ajwani *et al.* always set A to $\hat{A}_{i+1}$ and B to $\hat{B}_{i+1}$ during the execution of REORDER but we set A to $\hat{A}_{i+1}$ only if $\hat{A}_i = \emptyset$ and B to $\hat{B}_{i+1}$ only if $\hat{B}_i = \emptyset$. We prove that the total number of calls to REORDER doesn't increase (bounded above by $O(n^2)$) by introducing this modification. Thus intuitively, our algorithm should run faster because in each call to REORDER we might only need to compute $\hat{A}_i$ and $\hat{B}_i$ instead of $\hat{A}_{i+1}$ and $\hat{B}_{i+1}$.

10.2 Data Structures

10.2.1 Main Data Structures

In the following, we describe the main data structures used in our algorithm.

The current topological order T and its inverse T^{-1} are stored as arrays. Thus finding $T(i)$ and $T^{-1}(u)$ can be done in constant time.

The DAG $G = (V, E)$ is stored as an array of vertices. For each vertex u we maintain two adjacency lists $InList(u)$ and $OutList(u)$. The backward adjacency list $InList[u]$ contains all vertices v with $(v, u) \in E$. The forward adjacency list $OutList(u)$ contains all vertices v with $(u, v) \in E$. Adjacency lists are implemented by using n-bit arrays and support the following operations.

1. List-Insert: Given a vertex and a list, add the vertex to the list.
2. List-Search: Given a vertex and a list, determine if the vertex is in the list. If yes, return 1. Else, return 0.

Since the adjacency lists are implemented by using n-bit arrays, it takes $O(1)$ time per List-Insert or List-Search operation.

10.2.2 Auxiliary Data Structures

In the following we describe some auxiliary data structures which are used in our algorithm to improve the time complexity. For each vertex u, we maintain two arrays of pails: $InPails(u)[0 \ldots d+1]$ and $OutPails(u)[0 \ldots d+1]$. $InPails(u)[i]$ contains all vertices v with $0 < d(v, u) \leq t_i$ and $(v, u) \in E$. $OutPails(u)[i]$ contains all vertices v with $0 < d(u, v) \leq t_i$ and $(u, v) \in E$. A vertex v in a pail is stored with its vertex index (and not $T(v)$) as its key. Pails are implemented by using balanced binary search trees and support the following operations data structure should support the following three operations.

1. Pail-Insert: Given a vertex and a pail, add the vertex to the pail.
2. Pail-Delete: Given a vertex and a pail, delete the vertex from the pail.
3. Pail-Collect-All: Given a pail, report all vertices in the pail.

It takes $O(\log n)$ time per PAIL-INSERT or PAIL-DELETE and $O(1+\gamma)$ time per PAIL-COLLECT-ALL, where γ is the number of vertices in the pail.

10.2.3 Instructions for Data Structures

Given a DAG G with a valid topological order and two vertices u and v with $u \not\rightsquigarrow v$, define sorted vertex sets $\hat{A}_i$ and $\hat{B}_i$, $i = 0, \ldots, p+1$, as follows:

$$\hat{A}_i = \{w : u \to w \text{ and } d(u,w) \le t_i \text{ and } w < v\} \text{ sorted by the topological order.}$$

$$\hat{B}_i = \{w : w \to v \text{ and } d(u,w) \le t_i \text{ and } w > u\} \text{ sorted by the topological order.}$$

In the following we discuss how to insert an edge, compute vertex sets $\hat{A}_i$, and $\hat{B}_i$, and swap two vertices in terms of the above five basic operations.

a. Inserting an edge (u,v): This means inserting vertex v to the forward adjacency list of u and u to the backward adjacency list of v. This requires two LIST-INSERT operations and at most $2(p+2)$ PAIL-INSERT operations. Thus inserting an edge (u,v) can be done in $O(p \log n) = \tilde{O}(1)$ time.

b. Computing $\hat{A}_i$ and $\hat{B}_i$: $\hat{A}_i$ can be computed by sorting the vertices in $OutPail(u)[i]$ and choosing all w with $w < v$. This can be done by first calling PAIL-COLLECT-ALL to collect all the vertices in $OutPail(u)[i]$ in $O(t_i)$ time. Note that for all these vertices w, we have $0 < T(w) - T(u) \le t_i$. Thus we can sort these vertices in $O(t_i)$ time by counting sort and then choose all w with $w < v$ in $O(|\hat{A}_i| + 1)$ time. The total time required to compute $\hat{A}_i$ is $O(t_i + |\hat{A}_i|) = O(t_i)$. Similarly, the time required to compute $\hat{B}_i$ is $O(t_i)$.

c. Computing $\hat{A}_i$ and $\hat{B}_i$ when $t_i < d(u,v) \le t_{i+1}$: $\hat{A}_i$ can be computed by sorting the vertices in $OutPail(u)[i]$. This can be done by first calling PAIL-COLLECT-ALL to collect all the vertices in $OutPail(u)[i]$ in $O(|\hat{A}_i| + 1)$ time, and then sorting these vertices in $O((|\hat{A}_i| + 1) \log n)$ time. Thus the total time required to compute $\hat{A}_i$ is $\tilde{O}(|\hat{A}_i| + 1)$. Similarly, the time required to compute $\hat{B}_i$ is $\tilde{O}(|\hat{B}_i| + 1)$.

d. Swapping u and v: Without loss of generality assume $u < v$. When swapping u and v, we need to update the pails, T, and T^{-1}. We now show how to update the pails. For all vertices w with $T(u) - t_i \le T(w) < \min\{T(u), T(u) - t_i + d(u,v)\}$, we delete w

from $InPail(u)[i]$ and delete u from $OutPail(w)[i]$. For all vertices w with $\max\{T(v) + t_i - d(u,v), T(v)\} < T(w) \leq T(v) + t_i\}$, we delete w from $OutPail(v)[i]$ and delete v from $InPail(w)[i]$. It requires total $O(d(u,v))$ PAIL-DELETE operations for each i. For all w with $\max\{T(v), T(u) + t_i\} < T(w) \leq T(u) + t_i + d(u,v)$, if w is in the forward adjacency list of u, then insert w into $OutPail(u)[i]$ and insert u into $InPail(w)[i]$. For all w with $T(v) - t_i - d(u,v) \leq T(w) < \min\{T(u), T(v) - t_i + d(u,v)\}$, if w is in the backward adjacency list of v, then insert w into $InPail(v)[i]$ and insert v into $OutPail(w)[i]$. It requires total $O(d(u,v))$ LIST-SEARCH and PAIL-INSERT operations for each i. In total, we need $O(p \cdot d(u,v))$ LIST-SEARCH, PAIL-INSERT, and PAIL-DELETE operations. Updating T and T^{-1} is trivial and can be done in constant time. Thus the total time is $O(p \cdot d(u,v) \log n) = \tilde{O}(d(u,v))$.

10.3 Correctness

In this section, we argue that our algorithm is correct. We say a call of a recursive procedure leads to an operation "by itself" if and only if this operation is executed during the execution of this call and not during the execution of subsequent recursive calls. Given a DAG G with a valid topological order T and two vertices u, v of G with $u \prec v$, let $A' = \{w : u \rightarrow w \text{ and } w \leq v\}$ and $B' = \{w : w \rightarrow v \text{ and } w \geq u\}$. We say the flag f_1 of the call to REORDER(u, v, f_1, f_2) is correctly set only if $(f_1 \Rightarrow (A' = \emptyset)) = 1$. That is, if $f = 1$, then A' is empty. We say the flag f_2 of the call to REORDER(u, v, f_1, f_2) is correctly set only if $f_2 \Rightarrow (B' = \emptyset) = 1$. That is, if $f_2 = 1$, then B' is empty.

Lemma 10.1: Given a DAG G with a valid topological order and two vertices u, v of G with $u < v$, let $A' = \{w : u \rightarrow w \text{ and } w < v\}$ and $B' = \{w : w \rightarrow v \text{ and } w > u\}$. If the flags are correctly set, then in the call of REORDER(u, v, f_1, f_2), $A := \emptyset$ if and only if $A' = \emptyset$. Similarly, if the flags are correctly set, then in the call of REORDER(u, v, f_1, f_2), $B := \emptyset$ if and only if $B' = \emptyset$.

Proof: We only prove that $A := \emptyset$ if and only if $A' = \emptyset$. It can be proved in a similar way that $B := \emptyset$ if and only if $B' = \emptyset$. There are two cases to consider.

Case 1: $t_i < d(u,v) \leq t_{i+1}$ for some i with $0 < i \leq p$. By the algorithm, $A := \emptyset$ if and only if

$$\hat{A}_i = \{w : u \rightarrow w \text{ and } d(u,w) \leq t_i \text{ and } w < v\} = \emptyset \text{ and}$$

$$(\hat{A}_{i+1} = \{w : u \rightarrow w \text{ and } d(u,w) \leq t_{i+1} \text{ and } w < v\} = \emptyset \vee f_1 = 1).$$

By $t_i < d(u,v) \leq t_{i+1}$, we have $\hat{A}_i \subseteq \hat{A}_{i+1} = A'$. From $\hat{A}_i \subseteq \hat{A}_{i+1} = A'$ and $(f_1 \Rightarrow (A' = \emptyset)) = 1$, we conclude that $A := \emptyset$ if and only if $A' = \emptyset$.

Case 2: $d(u,v) \leq t_0$. By the algorithm, $A := \emptyset$ if and only if

$$\hat{A}_0 = \{w : u \rightarrow w \text{ and } d(u,w) \leq t_0 \text{ and } w < v\} = \emptyset.$$

By $d(u,v) \leq t_0$, we have $\hat{A}_0 = A'$. From $\hat{A}_0 = A'$, we conclude that $A := \emptyset$ if and only if $A' = \emptyset$. □

Lemma 10.2: Given a DAG G with a valid topological order and two vertices u, v of G with $u < v$, let $A' = \{w : u \rightarrow w \text{ and } w < v\}$ and $B' = \{w : w \rightarrow v \text{ and } w > u\}$. If the flags are correctly set, then REORDER(u, v, f_1, f_2) leads to a swap by itself if and only if $A' = \emptyset$ and $B' = \emptyset$.

Proof: By the algorithm, the call to REORDER(u, v, f_1, f_2) leads to a swap by itself if and only if in this call $A := \emptyset$ and $B := \emptyset$. By Lemma 10.1, $A := \emptyset$ and $B := \emptyset$ if and only if $A' = \emptyset$ and $B' = \emptyset$. □

Given a DAG G with a valid topological order, REORDER(u, v, f_1, f_2) is said to be *local* if and only if the execution of REORDER(u, v) does not affect $T(w)$ for all w with $w > v$ or $w < u$.

Lemma 10.3: Given a DAG G with a valid topological order and two vertices u, v with $u \not\rightsquigarrow v$, if the flags are correctly set, then REORDER(u, v, f_1, f_2) maintains a valid topological order and stop with $v < u$ and is local.

Proof: We prove the lemma by induction on $T(v) - T(u)$. When $T(u) - T(v) \leq 0$, the lemma is trivially correct.

Assume the lemma to be true when $T(v) - T(u) < k$, where $k > 0$. We prove that the lemma is true when $T(v) - T(u) = k$. If $A' = \{w : u \rightarrow w \text{ and } w < v\} = \emptyset$ and $B' = \{w : w \rightarrow v \text{ and } w > u\} = \emptyset$, then by Lemma 10.2, line 13 is executed. Thus, REORDER(u, v, f_1, f_2) maintains a valid topological order, stops with $v < u$, and only $T(u)$ and $T(v)$ are updated, so the lemma follows. If $A' \neq \emptyset$ or $B' \neq \emptyset$, by Lemma 10.2, the **for**-loops are executed. Let T' be the initial topological order. By our induction hypothesis and Lemma 10.1, the following loop invariants hold:

1. T is a valid topological order.
2. At the start of the execution of line 19, $T(v') - T(u') < k$ and $T'(u) \leq T(u') < T(v') \leq T'(v)$.
3. At the start of the execution of line 19, $u' \not\rightsquigarrow v'$.
4. The flags are correctly set for the recursive call.

By the loop invariants and our induction hypothesis, each recursive call REORDER(u', v', f_1', f_2') in the **for**-loops stops with $v' < u'$ and is local. Since the last recursive call is REORDER(u, v), the entire procedure stops with $v < u$. Since each recursive call REORDER(u', v') is local and starts with $T'(u) \leq T(u') < T(v') \leq T'(v)$, the topological order of vertices w with $T'(w) > T'(v)$ or $T'(w) < T'(u)$ is not affected. Thus the entire procedure maintains a valid topological order, stops with $v < u$, and is local. □

Theorem 10.1: Given a DAG G with a valid topological order and two vertices u, v of G with $u \not\rightsquigarrow v$, the call to INSERT(u, v) adds an edge (u, v) to G and maintains a valid topological order.

Proof: Because $u \not\rightsquigarrow v$, we know that u and v are two different vertices and either $u < v$ or $u > v$. If $u < v$ then the theorem is trivially correct. Assume that $v > u$. By Lemma 10.3, REORDER$(v, u, 0, 0)$ stops with $u < v$ and maintain a valid topological order. Thus when line 2 of INSERT is ready to be executed, we have a valid topological order and $u < v$, and add an edge (u, v) to G doesn't affect the validness of the topological order. □

In addition to the correctness of the algorithm, we also want to prove that the flags are always correctly set.

Lemma 10.4: Given a DAG G with a valid topological order and two vertices u, v of G with $u < v$, consider a call to REORDER(u, v, f_1, f_2). If the flags f_1 and f_2 are correctly set, then while executing this call, all subsequent calls to REORDER also own correct flags.

Proof: We prove the lemma by induction on the depth of the recursion tree. By Lemma 10.3, the call to REORDER(u, v, f_1, f_2) would stop, so the depth of the recursion tree is finite. If the depth is zero, then no recursive calls are made and the lemma follows.

Assume the lemma to be true when the depth of the recursion tree is less than k, where $k > 0$. We prove that the lemma is true when the depth of the recursion tree is k. Since $k > 0$, there is at least one recursive call. Thus the **for**-loops are executed. By Lemma 10.1 and Lemma 10.2, the following loop invariants hold:

1. T is a valid topological order.
2. At the start of the execution of line 19, $T(u') < T(v')$.
3. At the start of the execution of line 19, $u' \not\rightsquigarrow v'$.
4. The flags are correctly set for the recursive call.

By the loop invariants and our induction hypothesis, each recursive call to REORDER(u', v', f_1', f_2') in the **for**-loops, together with all subsequent calls to REORDER in it, own correct flags, and the lemma follows. □

Theorem 10.2: Given a DAG G with a valid topological order and two vertices u, v of G with $u \not\rightsquigarrow v$, while executing INSERT(u, v), the flags are correctly set for all calls to REORDER.

Proof: If $u < v$ then there are no calls to REORDER made while executing INSERT(u, v), and the lemma follows. If $u > v$ then INSERT(u, v) calls REORDER$(v, u, 0, 0)$. By Lemma 10.4, the call to REORDER$(v, u, 0, 0)$, together with all subsequent calls to REORDER in it, own correct flags, and the lemma follows. □

10.4 Running Time

In this section, we analyze the time required to insert a sequence of edges. By Theorem 10.2, the flags are always correctly set. To avoid unnecessary discussion, each lemma, theorem, corollary, and proof in this section is state under the assumption that the flags are correctly set. To avoid notational overload, sometimes we just write REORDER(u, v) and ignore the flags.

10.4.1 Properties

Lemma 10.5: Given a DAG G with a valid topological order and two vertices u and v with $u < v$, then during the execution of REORDER(u, v), we have that (1) $T(x)$ is nondecreasing if $u \rightsquigarrow x$; (2) $T(y)$ is nonincreasing if $y \rightsquigarrow v$; and (3) $T(z)$ doesn't change if $u \not\rightsquigarrow z$ and $z \not\rightsquigarrow v$.

Proof: We prove the lemma by induction on the depth of the recursion tree. By Lemma 10.3, the call to REORDER(u, v) would stop, so the depth of the recursion tree is finite. If the depth is zero, then no recursive calls are made. It follows that line 13 is executed, so the lemma follows.

Assume the lemma to be true when the depth of the recursion tree is less than k, where $k > 0$. We prove that the lemma is true when the depth of the recursion tree is k. Since $k > 0$, there is at least one recursive call. Thus the **for**-loops are executed. By Lemma 10.1 and Lemma 10.2, the following loop invariants hold:

1. T is a valid topological order.
2. At the start of the execution of line 19, $T(u') < T(v')$.
3. At the start of the execution of line 19, $u' \not\rightsquigarrow v'$.
4. The flags are correctly set for the recursive call.

Note that for each recursive call REORDER(u', v') in the **for**-loops, we have $u \rightsquigarrow u'$ and $v' \rightsquigarrow v$. Let $u \rightsquigarrow x$. Then we have either $u' \rightsquigarrow x$ or ($u' \not\rightsquigarrow x$ and $x \not\rightsquigarrow v'$). By the induction hypothesis, $T(x)$ is nondecreasing or doesn't change during the execution of the recursive call. Thus $T(x)$ is nondecreasing if $u \rightsquigarrow x$. Let $y \rightsquigarrow v$. Then we have either $y \rightsquigarrow v'$ or ($u' \not\rightsquigarrow y$ and $y \not\rightsquigarrow v'$). By the induction hypothesis, $T(y)$ is nonincreasing or doesn't change during the execution of the

recursive call. Thus $T(y)$ is nonincreasing if $y \rightsquigarrow v$. Let $u \not\rightsquigarrow z$ and $z \not\rightsquigarrow v$. Then $u' \not\rightsquigarrow z$ and $z \not\rightsquigarrow v'$. By the induction hypothesis, $T(z)$ doesn't change during the execution of the recursive call. Thus $T(z)$ doesn't change if $u \not\rightsquigarrow z$ and $z \not\rightsquigarrow v$. □

Lemma 10.6: Given a DAG G with a valid topological order and two vertices u and v with $u < v$, for all x and y, REORDER(u, v) leads to at most one swap of x and y.

Proof: Suppose that REORDER(u, v) leads to at least one swap of x and y. Without loss of generality we assume that $x < y$ before the first swap occurs. Then the first swap of x and y leads to increase of $T(x)$ and decrease of $T(y)$. Thus by Lemma 10.5, $T(x)$ is nondecreasing and $T(y)$ is nonincreasing during the execution of REORDER(u, v). After the first swap, we have $x > y$. Since $T(x)$ is nondecreasing and $T(y)$ is nonincreasing, we know there are no more swaps. □

Theorem 10.3: While we insert a sequence of edges, for all vertices x and y, after the first swap of x and y, the relative order of x and y doesn't change.

Proof: Suppose that the vertex pair (x, y) is swapped at least once while inserting a sequence of edges. By Lemma 10.6 and the algorithm INSERT, each edge insertion leads to at most one swap of x and y. Let (u, v) be the first edge whose insertion leads to a swap of x and y. Without loss of generality we assume that $x < y$ before the first swap occurs. We prove that $x > y$ holds after the first swap of x and y. Consider the execution process of INSERT(u, v). The first swap occurs during the execution of REORDER(v, u). By Lemma 10.5, we have $v \rightsquigarrow x$ and $y \rightsquigarrow u$. Since $T(x)$ is nondecreasing and $T(y)$ is nonincreasing, after the first swap of x and y, $x > y$ holds until REORDER(v, u) returns. After REORDER(v, u) returns, the edge (u, v) is added to the graph. Thus we have $y \rightsquigarrow x$ and $x > y$ just after INSERT(u, v) returns. By Lemma 10.3, calls to REORDER maintain a valid topological order, so $x > y$ holds hereafter. □

Corollary 10.1: While we insert a sequence of edges, for all vertices x and y, there is at most one swap of x and y.

Lemma 10.7: Given a DAG G with a valid topological order and two vertices u and v with $u < v$, REORDER(u, v) leads to a swap of u and v.

Proof: We prove the lemma by induction on the depth of the recursion tree. By Lemma 10.3, the call to REORDER(u, v) would stop, so the depth of the recursion tree is finite. If the depth is zero, then no recursive calls are made. It follows that line 13 is executed, so the lemma follows.

Assume the lemma to be true when the depth of the recursion tree is less than k, where $k > 0$. We prove that the lemma is true when the depth of the recursion tree is k. Since $k > 0$, there is at least one recursive call. Thus the **for**-loops are executed. By Lemma 10.1 and Lemma 10.2, the following loop invariants hold:

1. T is a valid topological order.
2. At the start of the execution of line 19, $T(u') < T(v')$.
3. At the start of the execution of line 19, $u' \not\rightsquigarrow v'$.
4. The flags are correctly set for the recursive call.

Note that when executing the last recursive call REORDER(u', v') in the **for**-loops, we have $u' = u$ and $v' = v$. By our induction hypothesis and the loop invariants, the last recursive call leads to a swap of u and v, and the lemma follows. □

Theorem 10.4: While we insert a sequence of edges, the summation of $|A| + |B|$ over all calls of REORDER is $O(n^2)$.

Proof: Consider arbitrary vertices u and $\hat{v}$. We prove that $\hat{v} \in B$ occurs at most once over all calls of REORDER$(u, \cdot)$. This proves that the summation of $|B|$ over all calls of REORDER$(u, \cdot)$ is less than or equal to n. Therefore the summation of $|B|$ over all calls of REORDER$(\cdot, \cdot)$ is less than or equal to n^2.

Consider the execution process of the first call of REORDER$(u, \cdot)$ for which $\hat{v} \in B$. By the algorithm, a recursive call to REORDER$(u, \hat{v})$ is made in the **for**-loops. Before the recursive call to REORDER$(u, \hat{v})$ in the **for**-loops, at the start of the execution of each recursive call to

REORDER(u', v') in the **for**-loops, we have $u < \hat{v}$ and ($u < u' < v'$ or $u = u' < v' < \hat{v}$). This follows from the order in which we make the recursive calls and the local property (Lemma 10.3). Since $u < \hat{v}$ and ($u < u' < v'$ or $u = u' < v' < \hat{v}$), by the local property, $u < \hat{v}$ holds during the execution of the call to REORDER(u', v'). Thus before the recursive call to REORDER$(u, \hat{v})$ in the **for**-loops, all recursive calls to REORDER(u', v') in the **for**-loops don't lead to a call to REORDER$(u, \cdot)$ for which $\hat{v} \in B$; otherwise, the call to REORDER$(u, \cdot)$ for which $\hat{v} \in B$ leads to a call to REORDER$(u, \hat{v})$ which further leads to $\hat{v} > u$ by Lemma 10.3, leading to a contradiction. Suppose for the contradiction that the recursive call to REORDER$(u, \hat{v})$ in the **for**-loops leads to a call to REORDER(u, v'') for which $\hat{v} \in B$. By the order in which we make the recursive calls and the local property, at the start of the execution of the recursive call to REORDER$(u, \hat{v})$ in the **for**-loops, we have $u < \hat{v}$. Since $\hat{v} \rightsquigarrow v''$, at the start of the execution of the recursive call to REORDER$(u, \hat{v})$ in the **for**-loops, we have $u < \hat{v} < v''$. Thus by the local property, $v'' > u$ holds during the execution of the recursive call to REORDER$(u, \hat{v})$ in the **for**-loops. However, by Lemma 10.3, REORDER(u, v'') stops with $v'' < u$, which is a contradiction. After the recurve call to REORDER$(u, \hat{v})$ in the **for**-loops, we have $\hat{v} < u$ by Lemma 10.3. Since $\hat{v} > u$ before the recurve call to REORDER$(u, \hat{v})$ in the **for**-loops, by Lemma 10.7, this recursive call leads to a swap of u and $\hat{v}$. Thus after the recurve call to REORDER$(u, \hat{v})$ in the **for**-loops, we have $\hat{v} < u$ and, by Lemma 10.3, the relative order of u and $\hat{v}$ doesn't change hereafter. Since $\hat{v} < u$ holds hereafter, there are no more calls of REORDER$(u, \cdot)$ for which $\hat{v} \in B$. Putting all things together, it follows that $\hat{v} \in B$ occurs at most once over all calls of REORDER$(u, \cdot)$.

Similarly, we can prove that for arbitrary vertices $\hat{u}$ and v, $\hat{u} \in A$ occurs at most once over all calls of REORDER$(\cdot, v)$. It follows that the summation of $|A|$ over all calls of REORDER$(\cdot, \cdot)$ is less than or equal to n^2. □

Corollary 10.2: While we insert a sequence of edges, REORDER is called $O(n^2)$ times.

Proof: By the algorithm, a call to REORDER for which $|A| + |B| = 0$ leads to a swap by itself. By Corollary 10.1, there are at most n^2 swaps. Thus there are at most n^2 calls to REORDER for which $|A| + |B| = 0$. By Theorem 10.4, there are $O(n^2)$ calls to REORDER for which $|A| + |B| > 0$. Therefore, there are $O(n^2)$ calls to REORDER in total. □

Let $S = \{(u, v) :$ there is a swap of u and v such that $u < v$ while we insert the edges$\}$. Define

$$D(u,v) = \begin{cases} \text{the value of } d(u,v) \text{ while swapping } u \text{ and } v & \forall (u,v) \in S; \\ 0 & \forall (u,v) \notin S. \end{cases}$$

Since by Corollary 10.1 each vertex pair is swapped at most once, $D(u, v)$ is well defined. Let k be a number with $1 \leq k \leq n$. Define

$$D_k(u,v) = \begin{cases} D(u,v) & \text{if } D(u,v) \leq k; \\ k & \text{otherwise.} \end{cases}$$

The following theorem is the key to our running time analysis.

Theorem 10.5: For all $k \in [1, n]$ with $k \geq n^{0.5}$, we have $\sum D_k(u, v) = O(n^2 \cdot \sqrt{k})$.

Proof: Let $k = n^r$. Let T^* denote the final topological order. Define $x(T^*(u), T^*(v)) = D(u, v)$ and $z(T^*(u), T^*(v)) = D_k(u, v)$ for all vertices u and v. The following linear inequalities are proved to be true by Ajwani *et al.* [2].

(1) $x(i, j) \leq 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq i$.

(2) $x(i, j) \leq n$ for all $1 \leq i \leq n$ and $i < j \leq n$.

(3) $\sum_{j>i} x(i, j) - \sum_{j<i} x(j, i) \leq n$ for all $1 \leq i \leq n$.

It is easy to derive the following linear inequalities from the definitions of $x(i, j)$ and $z(i, j)$.

(4) $z(i, j) \leq n^r$ for all $1 \leq i, j \leq n$.

(5) $z(i, j) \leq x(i, j)$ for all $1 \leq i, j \leq n$.

(6) $0 \leq z(i, j)$ for all $1 \leq i \leq n$ and $1 \leq j \leq n$.

(7) $0 \leq x(i, j)$ for all $1 \leq i \leq n$ and $1 \leq j \leq n$.

We aim to estimate an upper bound on the objective values of the following linear program.

$$\max \sum_{1 \leq i,j \leq n} z(i, j) \text{ such that}$$

(1) $x(i,j) \leq 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq i$

(2) $x(i,j) \leq n$ for all $1 \leq i \leq n$ and $i < j \leq n$

(3) $\sum_{j>i} x(i,j) - \sum_{j<i} x(j,i) \leq n$ for all $1 \leq i \leq n$

(4) $z(i,j) \leq n^r$ for all $1 \leq i,j \leq n$

(5) $z(i,j) \leq x(i,j)$ for all $1 \leq i,j \leq n$

(6) $0 \leq z(i,j)$ for all $1 \leq i \leq n$ and $1 \leq j \leq n$

(7) $0 \leq x(i,j)$ for all $1 \leq i \leq n$ and $1 \leq j \leq n$.

In order to prove the upper bound on the objective values of the above linear program, we consider its dual problem.

$$\min \left[\sum_{0 \leq i < j < n} n \cdot Y_{i \cdot n+j} + \sum_{0 \leq i < n} n \cdot Y_{n^2+i} + \sum_{0 \leq i,j < n} n^r \cdot Z_{i \cdot n+j} \right] \text{ such that}$$

(1) $Y_{i \cdot n+j} - W_{i \cdot n+j} \geq 0$ for all $0 \leq i \leq n$ and $0 \leq j \leq i$

(2) $Y_{i \cdot n+j} - W_{i \cdot n+j} + Y_{n^2+i} - Y_{n^2+j} \geq 0$ for all $0 \leq i < n$ and $j > i$

(3) $Z_{i \cdot n+j} + W_{i \cdot n+j} \geq 1$ for all $0 \leq i < n$ and $0 \leq j < n$

(4) $Y_i \geq 0$ for all $0 \leq i < n^2 + n$

(5) $Z_i \geq 0$ for all $0 \leq i < n^2$

(6) $W_i \geq 0$ for all $0 \leq i < n^2$.

Let c be a large enough constant, e.g. 120, such that $(i + c \cdot n^{r/2})^{r/2} \geq (i^{r/2} + 1)$ for any $1 \leq i \leq n$. The following is a feasible solution to the dual problem.

$$\begin{cases} Y_{i\cdot n+j} = 1 & \text{for all } 0 \le i < n \text{ and } 0 \le j \le i \\ Z_{i\cdot n+j} = 0 & \text{for all } 0 \le i < n \text{ and } 0 \le j \le i \\ W_{i\cdot n+j} = 1 & \text{for all } 0 \le i < n \text{ and } 0 \le j \le i \\ Y_{i\cdot n+j} = 0 & \text{for all } 0 \le i < n \text{ and } i < j \le i + c \cdot n^{r/2} \\ Z_{i\cdot n+j} = 1 & \text{for all } 0 \le i < n \text{ and } i < j \le i + c \cdot n^{r/2} \\ W_{i\cdot n+j} = 0 & \text{for all } 0 \le i < n \text{ and } i < j \le i + c \cdot n^{r/2} \\ Y_{i\cdot n+j} = 0 & \text{for all } 0 \le i < n \text{ and } i + c \cdot n^{r/2} < j < n \\ Z_{i\cdot n+j} = 0 & \text{for all } 0 \le i < n \text{ and } i + c \cdot n^{r/2} < j < n \\ W_{i\cdot n+j} = 1 & \text{for all } 0 \le i < n \text{ and } i + c \cdot n^{r/2} < j < n \\ Y_{n^2+i} = (n-i)^{r/2} & \text{for all } 0 \le i < n \end{cases}$$

This feasible solution to the dual problem has an objective value of $O(n \sum_{i=1}^{n} i^{r/2} + n \cdot n^r \cdot c \cdot n^{r/2}) = O(n^{2+r/2} + n^{1+r+r/2}) = O(n^{2+r/2}) = O(n^2 \cdot \sqrt{k})$, which by the primal-dual theorem is an upper bound on the objective values of the original linear program. □

Lemma 10.8: Given a DAG G with a valid topological order and two vertices u and v with $u < v$, consider a call to REORDER(u, v). If $A' = \{w : u \to w \text{ and } w < v\} = \emptyset$, then when executing this call, we have the first flag $f_1 = 1$ for all subsequent calls to REORDER$(u, \cdot)$.

Proof: We prove the lemma by induction on the depth of the recursion tree. By Lemma 10.3, the call to REORDER(u, v) would stop, so the depth of the recursion tree is finite. If the depth is zero, then no subsequent recursive calls are made, so the lemma follows.

Assume the lemma to be true when the depth of the recursion tree is less than k, where $k > 0$. We prove that the lemma is true when the depth of the recursion tree is k. Since $k > 0$, there is at least one recursive call. Thus the **for**-loops are executed. By Lemma 10.1, $A = \emptyset$. By the local property (Lemma 10.2) and the order in which we make the recursive calls, any subsequent call to REORDER$(u, \cdot)$ must occur during the execution of last iteration of the outer **for**-loop. Consider any first level recursive call REORDER(u', v', f_1', f_2') in the last iteration of

the outer **for**-loop. Note that we have $u' = u < v'$ and $\{w : u \to w \text{ and } w < v' \leq v\} \subseteq A' = \emptyset$ when this call is ready to be executed. Since $u' = u$ and $A = \emptyset$, we also have $f'_1 = 1$. By the induction hypothesis, we also have the first flag $f_1 = 1$ for all subsequent calls to REORDER$(u, \cdot)$ while executing this call. It completes the proof. □

Lemma 10.9: Given a DAG G with a valid topological order and two vertices u and v with $u < v$, let $A' = \{w : u \to w \text{ and } w < v\} = \emptyset$. Then a call to REORDER$(u, v)$ stops with u at the initial position of v. That is, letting T_{before} be the topological order just before the call to REORDER(u, v), then the call to REORDER(u, v) returns a topological order T_{after} such that $T_{after}(u) = T_{before}(v)$.

Proof: We prove the lemma by induction on the depth of the recursion tree. By Lemma 10.3, the call to REORDER(u, v) would stop, so the depth of the recursion tree is finite. If the depth is zero, then no recursive calls are made. It follows that line 13 is executed, so the lemma follows.

Assume the lemma to be true when the depth of the recursion tree is less than k, where $k > 0$. We prove that the lemma is true when the depth of the recursion tree is k. Since $k > 0$, there is at least one recursive call. Thus the **for**-loops are executed. Let T_{before} be the initial topological order. By Lemma 10.1, Lemma 10.2, and the induction hypothesis, the following loop invariants hold:

1. T is a valid topological order.
2. At the start of the execution of line 19, $T(u) = T(u') < T(v') = T_{before}(v')$.
3. At the start of the execution of line 19, $u' \not\rightsquigarrow v'$.
4. After the execution of line 19, $T(u) = T_{before}(v')$.
5. The flags are correctly set for the recursive call.

Note that for the last recursive call REORDER(u', v') in the **for**-loops, we have $v' = v$. Thus by the loop invariants, we have $T(u) = T_{before}(v)$ after the last recursive call in the **for**-loops. □

Lemma 10.10: Given a DAG G with a valid topological order and two vertices u and v with $u < v$, when executing a call of REORDER(u, v) in which both $\hat{A}_i$ and $\hat{A}_{i+1}$ are computed, u is moved right with distance at least t_i.

Proof: Since both $\hat{A}_i$ and $\hat{A}_{i+1}$ are computed, by the algorithm, we have $t_i < d(u, v) \leq t_{i+1}$ and $\hat{A}_i = \emptyset$. There are two cases to consider.

Case 1: $\hat{A}_{i+1} = \emptyset$. It follows that $A' = \{w : u \rightarrow w \text{ and } w < v\} = \emptyset$. By Lemma 10.9, u is moved right to the initial position of v. Since initially $d(u, v) > t_i$, u is moved right with distance at least t_i.

Case 2: $\hat{A}_{i+1} \neq \emptyset$. By the algorithm, the **for**-loops are executed. Let $\hat{u}$ be the vertex with lowest topological order in $\hat{A}_{i+1}$. Let $T_{initial}$ be the initial topological order. Since $\hat{A}_i = \emptyset$, we have initially $d(u, \hat{u}) > t_i$, i.e., $T_{initial}(\hat{u}) - T_{initial}(v) > t_i$. By Lemma 10.3 and the order in which we make the recursive calls, before the last iteration of the outer **for**-loop, $T(v) \geq T_{initial}(\hat{u})$ holds. Consider the execution of the last iteration of the outer **for**-loop. Let T_{start} be the topological order at the start of this iteration. Then we have $T_{start}(v) - T_{initial}(u) \geq T_{initial}(\hat{u}) - T_{initial}(u) > t_i$. By Lemma 10.3 and Lemma 10.9, the following loop invariants hold.

1. T is a valid topological order.
2. At the start of the execution of line 19, $T(u) = T(u') < T(v') = T_{start}(v')$.
3. At the start of the execution of line 19, $u' \not\rightsquigarrow v'$.
4. At the start of the execution of line 19, $\{w : u \rightarrow w \text{ and } w < v' \leq v\} = \emptyset$.
5. After the execution of line 19, $T(u) = T_{start}(v')$.
6. The flags are correctly set for the recursive call.

Thus after this iteration, we have $T(u) = T_{start}(v) > T_{initial}(u) + t_i$, and the lemma follows. □

Theorem 10.6: While we insert a sequence of edges, there are at most $O(\frac{n^2\sqrt{t_{i+1}}}{t_i})$ calls of REORDER for which both $\hat{A}_i$ and $\hat{A}_{i+1}$ are computed. Similarly, there are at most $O(\frac{n^2\sqrt{t_{i+1}}}{t_i})$ calls of REORDER for which both $\hat{B}_i$ and $\hat{B}_{i+1}$ are computed.

Proof: We only prove that there are at most $O(\frac{n^2\sqrt{t_{i+1}}}{t_i})$ calls of REORDER for which both $\hat{A}_i$ and $\hat{A}_{i+1}$ are computed. It can be proved in a similar way that there are at most $O(\frac{n^2\sqrt{t_{i+1}}}{t_i})$ calls of REORDER for which both $\hat{B}_i$ and $\hat{B}_{i+1}$ are computed.

Let $C_1(u,v), C_2(u,v), \ldots, C_{m(u,v)}(u,v)$ be the calls to REORDER(u,v) for which both $\hat{A}_i$ and $\hat{A}_{i+1}$ are computed for all vertices u and v. Let $S_i(u,v) = \{w : C_i(u,v)$ leads to a swap of u and $w\}$ for $i = 1, \ldots, m(u,v)$. We prove that $\sum_{u,v}\sum_{i=1}^{m(u,v)}\sum_{w\in S_i(u,v)} D(u,w) = O(n^2\sqrt{t_{i+1}})$. By Lemma 10.10, $\sum_{w\in S_i(u,v)} D(u,w) \geq t_i$ for all vertices u and v and $i = 1, \ldots, m(u,v)$. It follows that $\sum_{u,v}\sum_{i=1}^{m(u,v)} 1 = O(\frac{n^2\sqrt{t_{i+1}}}{t_i})$.

In a call to REORDER(u,v), $\hat{A}_i$ and $\hat{A}_{i+1}$ are both computed only if $t_i < d(u,v) \leq t_{i+1}$. Thus by the local property (Lemma 10.3), we have $D(u,w) \leq t_{i+1}$ for all $w \in S_i(u,v)$, $i = 1, \ldots, m(u,v)$. By Theorem 10.5, we have $\sum_{u,v} D_t(u,v) = O(n^2\sqrt{t_{i+1}})$. Thus it suffices to show that in the summation $\sum_{u,v}\sum_{i=1}^{m(u,v)}\sum_{w\in S_i(u,v)} D(u,w)$, $D(u,w)$ is counted at most twice for each vertex pair (u,w).

To show that in the summation $\sum_{u,v}\sum_{i=1}^{m(u,v)}\sum_{w\in S_i(u,v)} D(u,w)$, $D(u,w)$ is counted at most twice for each vertex pair (u,v), we only have to prove that $S_i(u,v) \cap S_j(u,v') \cap S_k(u,v'') = \emptyset$ if $C_i(u,v)$, $C_j(u,v')$, and $C_k(u,v'')$ are three different calls.

Suppose for the contradiction that $w \in S_i(u,v) \cap S_j(u,v') \cap S_k(u,v'')$ and $C_i(u,v)$, $C_j(u,v')$, and $C_k(u,v'')$ are three different calls. Without loss of generality, we assume $C_i(u,v)$ occurs before $C_j(u,v')$ and $C_j(u,v')$ occurs before $C_k(u,v'')$. By Corollary 10.1, there is only one swap of u and w, so $C_j(u,v')$ must be a subsequent recursive call which occurs during the execution of $C_i(u,v)$ and $C_k(u,v'')$ must be a subsequent recursive call which occurs during the execution of $C_j(u,v')$. Consider the execution of $C_i(u,v)$. By Lemma 10.3 and the order in which we make the recursive calls in the **for**-loops, $C_j(u,v')$ must occur during the last iteration of the outer **for**-loop. Note that by Lemma 10.3, before the last iteration of the outer **for**-loop begins, all vertices in $\hat{A}_{i+1} = \{w : u \to w, d(u,w) \leq t_{i+1}$ and $w < v\} = \{w : u \to w$ and $w < v\}$ are moved to the right of v. Thus when the last iteration of the outer **for**-loop begins, there are not any vertices w between u and v with $u \to w$. Therefore, by the local property of REORDER, during the last iteration of the outer **for**-loop, for each call to REORDER(u,v'), we have $\{w : u \to w$ and $w < v' \leq v\} = \emptyset$. By Lemma 10.8, we have the first flag $f_1 = 1$ for each subsequent call to REORDER$(u,\cdot)$ during the execution of $C_j(u,v')$. It follows that the first

flag f_1 of the call $C_k(u, v'')$ is 1. Thus by the algorithm, we don't compute $\hat{A}_{i+1}$ in the call $C_k(u, v'')$, which is a contradiction. □

10.4.2 Running Time Analysis

Lemma 10.11: While we insert a sequence of edges, the total time spent on executing line 2 of INSERT is $\tilde{O}(n^2)$.

Proof: As discussed in Section 10.2.3, each execution of line 2 of INSERT can be done in $\tilde{O}(1)$ time. Since there are at most $n(n-1)/2$ edge insertions, the lemma follows. □

Lemma 10.12: While we insert a sequence of edges, the total time spent on computing $\hat{A}_i$ and $\hat{B}_i$, $i = 1, \ldots, p$, over all calls of REORDER(u, v) with $t_i < d(u, v) \leq t_{i+1}$ is $\tilde{O}(n^2)$.

Proof: As discussed in Section 10.2.3, it needs $\tilde{O}(|\hat{A}_i| + |\hat{B}_i| + 1)$ time to compute $\hat{A}_i$ and $\hat{B}_i$ in a call of REORDER(u, v) if $d(u, v) < t_{i+1}$. By Theorem 10.4, the summation of $|\hat{A}_i| + |\hat{B}_i|$, $i = 1, \ldots, p$, over all calls of REORDER is $O(n^2)$. By Corollary 10.2, REORDER is called $O(n^2)$ times. Thus the summation of $(|\hat{A}_i| + |\hat{B}_i| + 1)$ over all calls of REORDER is $O(n^2)$, and the lemma follows. □

Lemma 10.13: For each i with $1 \leq i \leq p + 1$, while we insert a sequence of edges, the total time spent on computing $\hat{A}_i$ and $\hat{B}_i$, over all calls of REORDER(u, v) with $d(u, v) \leq t_i$ is $O(\frac{n^2 t_i^{3/2}}{t_{i-1}})$.

Proof: If $d(u, v) \leq t_i$, then by the algorithm, $\hat{A}_i$ is computed only if $\hat{A}_{i-1}$ is also computed and is empty. Thus by Theorem 10.6, there are at most $O(\frac{n^2 t_i^{1/2}}{t_{i-1}})$ such calls. As discussed in Section 10.2.3, it needs $O(t_i)$ time to compute $\hat{A}_i$ in each of such calls. Thus the total time spent on computing $\hat{A}_i$ over all calls of REORDER(u, v) with $d(u, v) \leq t_i$ is $O(\frac{n^2 t_i^{3/2}}{t_{i-1}})$. Similarly, the total time spent on computing $\hat{B}_i$ over all calls of REORDER(u, v) with $d(u, v) \leq t_i$ is $O(\frac{n^2 t_i^{3/2}}{t_{i-1}})$. □

Lemma 10.14: While we insert a sequence of edges, the total time spent on computing $\hat{A}_0$ and $\hat{B}_0$ over all calls of REORDER is $O(n^2 \cdot t_0)$.

Proof: As discussed in Section 10.2.3, it needs $O(t_0)$ time to compute $\hat{A}_0$ and $\hat{B}_0$ in a call of REORDER. By Corollary 10.2, REORDER is called $O(n^2)$ times, and the lemma follows. □

Lemma 10.15: While we insert a sequence of edges, the total time spent on swapping vertices is $\tilde{O}(n^{2.5})$.

Proof: As discussed in Section 10.2.3, each swap of vertices u and v with $d(u,v) < t$ can be done in $\tilde{O}(d(u,v))$ time. By Theorem 10.5, $\sum D_n(u,v) = O(n^{2.5})$. Thus the total time is $\tilde{O}(n^{2.5})$. □

Theorem 10.7: While we insert a sequence of edges, the total time required is $\tilde{O}(\sum_{i=1}^{p+1} \frac{n^2 t_i^{3/2}}{t_{i-1}} + n^2 \cdot t_0)$.

Proof: It follows directly from the above lemmas. □

10.5 Further Discussion

Let $n^{0.5} < t_0 < t_1 < \cdots < t_p < t_{p+1} = n$ and $p = O(\log n)$. We have known that the running time is $\tilde{O}(\sum_{i=1}^{p+1} \frac{n^2 t_i^{3/2}}{t_{i-1}} + n^2 \cdot t_0)$. In this section, we show how to determine the values of these parameters. By letting

$$\frac{n^3}{\sqrt{t_{p+1}}} = \frac{n^2 t_{p+1}^{3/2}}{t_p} = \frac{n^2 t_p^{3/2}}{t_{p-1}} = \frac{n^2 t_{p-1}^{3/2}}{t_{p-2}} = \cdots = \frac{n^2 t_1^{3/2}}{t_0} = n^2 \cdot t_0,$$

we have

$$t_1 = t_0^{4/3}, t_i = \frac{t_{i-1}^{5/3}}{t_{i-2}^{2/3}} \text{ for all } i = 2, \ldots, p+1.$$

Let $t_i = t_0^{x_i}$ for $i = 0, \ldots, p+1$. we have

$$x_0 = 1, x_1 = 4/3, \text{ and } x_i = \frac{5x_{i-1}}{3} - \frac{2x_{i-2}}{3} \text{ for all } i = 2, \ldots, p+1.$$

By solving this linear second order recurrence relation, we get $x_i = 2 - (2/3)^i$ for all $i = 0, 1, 2, \ldots, p+1$. It follows that $t_{p+1} = t_0^{2-(2/3)^{(p+1)}}$. Since $t_{p+1} = n$, we have $t_0 = n^{f(p)}$, where $f(p) = \frac{3^{p+1}}{(2 \cdot 3^{p+1} - 2^{p+1})}$.

Corollary 10.3: While we insert a sequence of edges, the total time required is $\tilde{O}(n^{2+f(p)})$ if $t_0 = n^{f(p)}$ and $t_i = t_0^{2-(2/3)^i}$ for all $i = 1, \ldots, p+1$, where $f(p) = \frac{3^{p+1}}{(2 \cdot 3^{p+1} - 2^{p+1})}$.

Note that $f(p) = \frac{3^{p+1}}{(2 \cdot 3^{p+1} - 2^{p+1})} = 0.5 + \frac{2^p}{(2 \cdot 3^{p+1} - 2^{p+1})}$. By letting $\epsilon(p) = \frac{2^p}{(2 \cdot 3^{p+1} - 2^{p+1})}$, we have $\epsilon(p) < \frac{2^p}{3^{p+1}} < (3/2)^{-p}$. By letting $p = \log_{3/2} n$, we have $1 < n^{\epsilon(p)} < n^{1/n} < 2$ when $n > 2$. Thus $\tilde{O}(n^{2+f(p)}) = \tilde{O}(n^{2.5+\epsilon(p)}) = \tilde{O}(n^{2.5})$ if we choose $p = \lceil \log_{3/2} n \rceil$. The following theorem summarizes our discussion.

Theorem 10.8: There exists an $\tilde{O}(n^{2.5})$-time algorithm for online topological ordering.

10.6 Notes

In this chapter, we propose an $\tilde{O}(n^{2.5})$-time algorithm for maintaining the topological order of a DAG with n vertices while we insert m edges. Precisely, our algorithm runs in $O(n^{2.5} \log^2 n)$ time. Choosing a better implementation for the pails, like data structures discussed in Section 5 of [2], can further improve the time bound to $O(n^{2.5} \log n)$.

INSERT(u, v)

(* Insert edge (u, v) and calculate new topological ordering *)

1 **if** $v \leq u$ **then** REORDER$(v, u, 0, 0)$.

2 Insert edge (u, v) into graph.

REORDER(u, v, f_1, f_2)

(* Reorder vertices between u and v such that $v \leq u$ *)

1 **if** $v \leq u$ **then** exit.

2 **if** $t_i < d(u, v) \leq t_{i+1}$

3 **then**

4 $\hat{A}_i := \{w : u \to w,\ d(u, w) \leq t_i, \text{ and } w < v\}$.

5 $\hat{B}_i := \{w : w \to v,\ d(w, v) \leq t_i, \text{ and } w > u\}$.

6 $A := \hat{A}_i$ if $\hat{A}_i \neq \emptyset$ or $f_1 = 1$; otherwise, $A := \hat{A}_{i+1} := \{w : u \to w,\ d(u, w) \leq t_{i+1}, \text{ and } w < v\}$.

7 $B := \hat{B}_i$ if $\hat{B}_i \neq \emptyset$ or $f_2 = 1$; otherwise, $B := \hat{B}_{i+1} := \{w : w \to v,\ d(w, v) \leq t_{i+1}, \text{ and } w > u\}$.

8 **else**

9 $A := \hat{A}_0 := \{w : u \to w,\ d(u, w) \leq t_0, \text{ and } w < v\}$.

10 $B := \hat{B}_0 := \{w : w \to v,\ d(w, v) \leq t_0, \text{ and } w > u\}$.

11 **if** $A = \emptyset$ and $B = \emptyset$

12 **then**

13 swap u and v.

14 **else**

15 **for** $u' \in \{u\} \cup A$ in decreasing topological order

16 **for** $v' \in B \cup \{v\} \wedge v' > u'$ in increasing topological order

17 $f_1' := 1$ if $(u = u' \text{ and } A = \emptyset)$; otherwise, $f_1' := 0$.

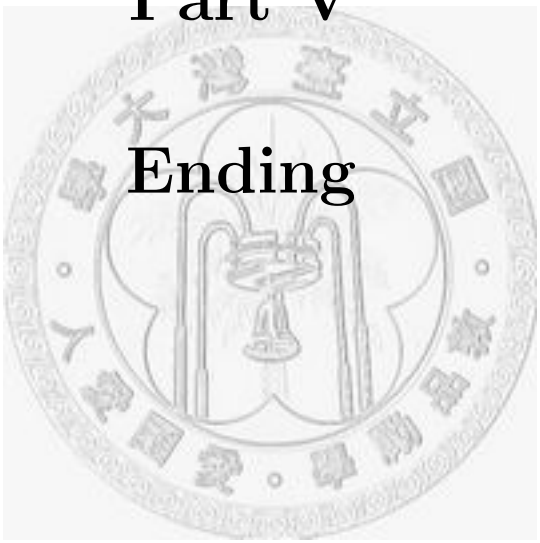
18 $f_2' := 1$ if $(v = v' \text{ and } B = \emptyset)$; otherwise, $f_2' := 0$.

19 REORDER(u', v', f_1', f_2').

Figure 10.1: The improved online topological ordering algorithm for dense graphs.

Part V

Ending



Chapter 11

Conclusions

In this chapter, we summarize the results and discuss our future work regarding the problems presented in this dissertation.

11.1 Summary

In this dissertation, we study a series of problems arising from sequence analysis. In Chapter 2, we study the Disjoint Segments with Maximum Sum of Densities problem and give an improved algorithm. In Chapter 3, we study the Length-Constrained Max-Eccentricity Segments Problem and obtain the first subquadratic time algorithm. In Chapter 4, we give a linear time algorithm for the Sum-Constrained Max-Density Intervals problem. In Chapter 5, we propose an optimal time algorithm for the Density Finding problem. In Chapters 6 and 7, we propose optimal algorithms for the Length-Constrained k Max-Sum Segments problem and the Weight-Constrained k Longest Paths problem, respectively. In Chapter 8, we study the Length-Constrained Sum Selection problem and propose an improved algorithm. In Chaper 9, we give a tight analysis of the Katriel-Bodlaender algorithm for online topological ordering. Finally, in Chapter 10, we propose an improved algorithm for online topological ordering on dense graphs.

11.2 Future Work

Min-Plus Convolution

The min-plus convolution of two vectors $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$ is a vector $\mathbf{z} = (z_0, z_1, \ldots, z_{n-1})$ such that $z_k = \min_{i=0}^{k}\{x_i + y_{k-i}\}$ for $k = 0, 1, \ldots, n-1$. Given two vectors $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$, the MIN-PLUS CONVOLUTION PROBLEM is to compute the min-plus convolution $\mathbf{z}$ of $\mathbf{x}$ and $\mathbf{y}$. This problem has appeared in the literature with various names such as "minimum convolution," "epigraphical sum," "inf-convolution," and "lowest midpoint" [8, 14, 32, 58, 62, 69, 71]. Although it is easy to obtain an $O(n^2)$-time algorithm, no subquadratic algorithm was known until recently Bremner *et al.* [17] proposed an $O(n^2/\log n)$-time algorithm. In Section 3.2, we give an slightly improved $O(n^2\frac{(\log\log n)^3}{(\log n)^2})$-time algorithm. To the best of our knowledge, there is not any non-trivial lower bound proved so far. Thus, there is still a large gap between the trivial lower bound of $O(n)$ and the upper bound of $O(n^2\frac{(\log\log n)^3}{(\log n)^2})$. Bridging this gap remains an open problem.

Minkowski Sum Selection

In the MINKOWSKI SUM SELECTION PROBLEM, we are given two n-point multisets $P, Q \subseteq \mathbb{R}^2$, a set of λ inequalities $Ar \geq b$, a linear objective function $f : \mathbb{R}^2 \to \mathbb{R}$, and a positive integer k. The goal is to find the k^{th} largest objective value among all objective values of points in the multiset $\{(p+q) : p \in P, q \in Q, A(p+q) \geq b\}$. Many known selection problems, like the SUM SELECTION PROBLEM [11, 54], the LENGTH-CONSTRAINED SUM SELECTION PROBLEM [55], and the CARTESIAN SUM SELECTION PROBLEM [45], are linear time reducible to the MINKOWSKI SUM SELECTION PROBLEM. It was proved in [45] that the MINKOWSKI SUM SELECTION PROBLEM has an $\Omega(n\log n)$ lower bound even if $\lambda = 0$. Recently, Luo *et al.* [57] proposed an optimal $O(n\log n)$-time algorithm for the case $\lambda \leq 1$, and it is of greatest interest whether one can prove an upper bound of $O(n\log n)$ for any fixed $\lambda > 1$.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ Algorithm for Online Topological Ordering. *SWAT*, 53–64, 2006.

[3] L. Allison. Longest Biased Interval and Longest Non-Negative Sum Interval. *Bioinformatics*, 19:1294–1295, 2003.

[4] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental Evaluation of Computational Circuits. *SODA*, 32–42, 1990.

[5] S. E. Bae and T. Takaoka. Algorithms for the Problem of k Maximum Sums and a VLSI Algorithm for the k Maximum Subarrays Problem. *ISPAN*, 247–253, 2004.

[6] S. E. Bae and T. Takaoka. Improved Algorithms for the k-Maximum Subarray Problem for Small k. *COCOON*, 621–631, 2005.

[7] S. E. Bae and T. Takaoka. A Sub-Cubic Time Algorithm for the k-Maximum Subarray Problem. *ISAAC*, 751–762, 2007.

[8] R. Bellman and W. Karush. Mathematical Programming and the Maximum Transform. *Journal of the Society for Industrial and Applied Mathematics*, 10(3):550–567, 1962.

[9] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two Simplified Algorithms for Maintaining Order in a List. *ESA*, 152–164, 2002.

[10] M. A. Bender and M. Farach-Colton. The LCA Problem Revisited. *LATIN*, 88–94, 2000.

[11] F. Bengtsson and J. Chen. Efficient Algorithms for k Maximum Sums. *Algorithmica* 46(1):27–41, 2006.

[12] J. Bentley. Programming Pearls: Algorithm Design Techniques. *Communications of the ACM*, 865–871, 1984.

[13] M. Ben-Or. Lower Bounds for Algebraic Computation Trees. *STOC*, 80–86, 1983.

[14] A. Bergkvist and P. Damaschke. Fast Algorithms for Finding Disjoint Subsequences with Extremal Densities. *Pattern Recognition*, 39(12):2281–2292, 2006.

[15] T. Bernholt, F. Eisenbrand, and T. Hofmeister. A Geometric Framework for Solving Subsequence Problems in Computational Biology Efficiently. *SoCG*, 310–318, 2007.

[16] T. Bernholt and T. Hofmeister. An Algorithm for a Generalized Maximum Subsequence Problem. *LATIN*, 178–189, 2006.

[17] D. Bremner, T. Chan, E. Demaine, J. Erickson, F. Hurtado, J. Iacono, S. Langerman, I. Streinu, and P. Taslakian. Necklaces, Convolutions, and X+Y. *ESA*, 160–171, 2006.

[18] G. S. Brodal and A. G. Jørgensen. A Linear Time Algorithm for the k Maximal Sums Problem. *MFCS*, 442–453, 2007.

[19] T. M. Chan. More Algorithms for All-Pairs Shortest Paths in Weighted Graphs. *STOC*, 2007.

[20] K.-Y. Chen and K.-M. Chao. Optimal Algorithms for Locating the Longest and Shortest Segments Satisfying a Sum or an Average Constraint. *Information Processing Letters*, 96(6):197–201, 2005.

[21] K.-Y. Chen and K.-M. Chao. On the Range Maximum-Sum Segment Query Problem. *Discrete Applied Mathematics*, 155(16):2043–2052, 2007.

[22] Y. H. Chen, H.-I Lu, and C. Y. Tang. Disjoint Segments with Maximum Density. *ICCS*, 845–850, 2005.

[23] C.-H. Cheng, K.-Y. Chen, W.-C. Tien, and K.-M. Chao. Improved Algorithms for the k Maximum-Sums Problems. *Theoretical Computer Science*, 362(1-3):162–170, 2006.

[24] C.-H. Cheng, H.-F. Liu, and K.-M. Chao. On Locating the Average-Constrained Maximum-Sum Segment. Research Report, 2008.

[25] K.-M. Chung and H.-I Lu. An Optimal Algorithm for the Maximum-Density Segment Problem. *SIAM Journal on Computing*, 34(2):373–387, 2004.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[27] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*, Second Edition, Springer, 2000.

[28] L. Davies and A. Kovac. Local Extremes, Runs, Strings and Multiresolution (with discussion). *Annals of Statistics*, 29(1):1–65, 2001.

[29] P. F. Dietz and D. D. Sleator. Two Algorithms for Maintaining Order in a List. *STOC*, 365–372, 1987.

[30] D. Eppstein. Finding the k Shortest Paths. *SIAM Journal on Computing*, 28:652–673,1998.

[31] T.-H. Fan, S. Lee, H.-I Lu, T.-S. Tsou, T.-C. Wang, and A. Yao. An Optimal Algorithm for Maximum-Sum Segment and Its Application in Bioinformatics Extended Abstract. *CIAA*: 251-257, 2003.

[32] P. Felzenszwalb and D. Huttenlocher. Distance Transforms of Sampled Functions. Technical Report TR2004-1963, Cornell Computing and Information Science, 2004.

[33] J. Fischer and V. Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. *CPM*, 36–48, 2006.

[34] G. N. Frederickson. An Optimal Algorithm for Selection in a Min-Heap. *Information and Computation* 104:197–214, 1993.

[35] G. N. Frederickson and D. B. Johnson. Optimal Algorithms for Generating Quantile Information in $X + Y$ and Matrices with Sorted Columns. in *Proceedings of the 13th Annual Conference on Information Science and Systems*, 47–52, 1979.

[36] G. N. Frederickson and D. B. Johnson. Finding kth paths and p-Centers by Generating and Searching Good Data Structures. *Journal of Algorithms*, 4:61–80, 1983.

[37] G. N. Frederickson and D. B. Johnson. Generalized Selection and Ranking: Sorted Matrices. *SIAM Journal on Computing*, 13(1):14–30, 1984.

[38] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[39] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining Optimized Association Rules for Numeric Attributes. *Journal of Computer and System Sciences*, 58(1):1–12, 1999.

[40] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data Mining with Optimized Two-Dimensional Association Rules. *ACM Transactions on Database Systems*, 26(2):179-213, 2001.

[41] M. Goldwasser, M.-Y. Kao and H.-I Lu. Linear-Time Algorithms for Computing Maximum-Density Sequence Segments with Bioinformatics Applications. *Journal of Computer and System Sciences*, 70(2):128–144, 2005.

[42] Y. Han. An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Path. *ESA*, 411–417, 2006.

[43] D. Harel and R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.

[44] X. Huang. An Algorithm for Identifying Regions of a DNA Sequence that Satisfy a Content Requirement. *Computer Applications in the Biosciences*, 10(3):219–225, 1994.

[45] D. B. Johnson and S. D. Kashdan. Lower Bounds for Selection in $X + Y$ and Other Multisets. *Journal of the ACM*, 25:556–570, 1978.

[46] I. Katriel. On Algorithms for Online Topological Ordering and Sorting. Research Report MPI-I-2004-1-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[47] I. Katriel and H. L. Bodlaender. Online Topological Ordering. *ACM Transactions on Algorithms*, 2(3):364–379, 2006.

[48] S. K. Kim. A Linear-Time Algorithm for Finding a Maximum-Density Segment of a Sequence. *Information Processing Letters*, 86(6):339–342, 2003.

[49] S. K. Kim. Finding a Longest Nonnegative Path in a Constant Degree Tree. *Information Processing Letters*, 93:275–279, 2003.

[50] D. T. Lee. Computational Geometry. *ACM Computing Surveys*, 28(1):21–31, 1996.

[51] D. T. Lee, T.-C. Lin, and H.-I Lu. Fast Algorithms for the Density Finding Problem *Algorithmica*, DOI:10.1007/s00453-007-9023-8, 2007.

[52] Y.-L. Lin, X. Huang, T. Jiang, and K.-M. Chao. MAVG: Locating Non-Overlapping Maximum Average Segments in a Given Sequence. *Bioinformatics*, 19(1):151–152, 2003.

[53] Y.-L. Lin, T. Jiang, and K.-M. Chao. Efficient Algorithms for Locating the Length-Constrained Heaviest Segments with Applications to Biomolecular Sequence Analysis. *Journal of Computer and System Sciences*, 65(3):570–586, 2002.

[54] T.-C. Lin and D. T. Lee. Efficient Algorithm for the Sum Selection Problem and k Maximum Sums Problem. *ISAAC*, 460–473, 2006.

[55] T.-C. Lin and D. T. Lee. Randomized Algorithm for the Sum Selection Problem. *Theoretical Computer Science*, 377(1-3):151-156, 2007.

[56] D. Lipson, Y. Aumann, A. Ben-Dor, N. Linial, and Z. Yakhini. Efficient Calculation of Interval Scores for DNA Copy Number Data Analysis. *Journal of Computational Biology*, 13(2):215-228, 2006.

[57] C.-W. Luo, P.-A. Chen, Hsiao-Fei Liu, and Kun-Mao Chao. Constrained Minkowski Sum Selection and Finding. Research Report.

[58] P. Maragos. Differential Morphology. *Nonlinear Image Processing*, 289–329, 2000.

[59] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-Line Graph Algorithms for Incremental Compilation. *WG*, 70–86, 1993.

[60] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a Topological Order Under Edge Insertions. *Information Processing Letters* 59(1):53–58, 1996.

[61] N. Megiddo, A. Tamir, E. Zemel, and R. Chandrasekaran. An $O(n \log^2 n)$ Algorithm for the kth Longest Path in a Tree with Applications to Location Problems. *SIAM Journal on Computing*, 10:328–337, 1981.

[62] J.-J. Moreau. Inf-Convolution, Sous-Additivité, Convexité Des Fonctions Numériques. *Journal de Mathématiques Pures et Appliquées*, 49:109–154, 1970.

[63] S. M. Omohundro, C.-C. Lim, and J. Bilmes. The Sather Language Compiler/Debugger Implementation. *Technical Report TR-92-017, International Computer Science Institute, Berkley*, 1992.

[64] D. J. Peace. Some Directed Graph Algorithms and Their Application to Pointer Analysis. Ph.D. Thesis, Imperial College of Science, Technology and Medicine, University of London, 2005.

[65] D. J. Pearce and P. H. J. Kelly. A Dynamic Algorithm for Topologically Sorting Directed Acyclic Graphs. *ACM Journal of Experimental Algorithms*, 11(1.7):1–24, 2007.

[66] D. J. Pearce, P. H. J. Kelly, and Chris Hankin. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Journal*, 12(4):309–335, 2004.

[67] F. Preparata and Michael Shamos. *Computational Geometry: An introduction*. Springer-Verlag, New York, 1985.

[68] G. Ramalingam and T. W. Reps. On Competitive On-Line Algorithms for the Dynamic Priority-Ordering Problem. *Information Processing Letters*, 51(3):155–161, 1994.

[69] R. T. Rockafellar. *Convex Analysis*, 1970.

[70] M. I. Shamos. Geometry and Statistics: Problems at the Interface, in Algorithms and Complexity: New Directions and Recent Results, Joseph F. Traub, ed., Academic Press, New York, 251–280, 1976.

[71] T. Strömberg. The Operation of Infimal Convolution. *Dissertationes Mathematicae*, 352:58, 1996.

[72] T. Takaoka. Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication. *Electronic Notes in Theoretical Computer Science*, 61:191-200, 2002.

[73] L. Wang and Y. Xu. SEGID: Identifying Interesting Segments in (Multiple) Sequence Alignments. *Bioinformatics*, 19(2):297–298, 2003.

[74] B. Y. Wu, K.-M. Chao, and C. Y. Tang. An Efficient Algorithm for the Length-Constrained Heaviest Path Problem on a Tree. *Information Processing Letters*, 69(2):63-67, 1999.