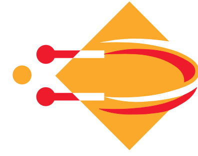


The IT University
of Copenhagen

Topics in Algorithms

Data Structures on Trees and Approximation Algorithms on Graphs

Inge Li Gørtz

Dissertation
Presented to the Faculty
of the IT University of Copenhagen
in Partial Fulfilment of
the Requirements for the Degree of
Doctor of Philosophy

Department of Theoretical Computer Science May 2005

Abstract

This dissertation is divided into two parts. Part I concerns algorithms and data structures on trees or involving trees. Here we study three different problems: efficient binary dispatching in object-oriented languages, tree inclusion, and union-find with deletions.

The results in Part II fall within the heading of approximation algorithms. Here we study variants of the k-center problem and hardness of approximation of the dial-a-ride problem.

Binary Dispatching The *dispatching problem* for object oriented languages is the problem of determining the most specialized method to invoke for calls at run-time. This can be a critical component of execution performance. The *unary dispatching problem* is equivalent to the *tree color problem*. The *binary dispatching problem* can be seen as a 2-dimensional generalization of the tree color problem which we call the *bridge color problem*.

We give a linear space data structure for *binary* dispatching that supports dispatching in logarithmic time. Our result is obtained by a employing a dynamic to static transformation technique. To solve the bridge color problem we turn it into a dynamic tree color problem, which is then solved persistently.

Tree Inclusion Given two rooted, ordered, and labeled trees P and T the tree inclusion problem is to determine if P can be obtained from T by deleting nodes in T. The tree inclusion problem has recently been recognized as an important query primitive in XML databases. We present a new approach to the tree inclusion problem which leads to a new algorithm that uses optimal linear space and has subquadratic running time or even faster when the number of leaves in one of the trees is small. More precisely, we give three algorithms that all uses $O(n_P+n_T)$ space and runs in $O(\frac{n_P\, n_T}{\log n_T})$, $O(l_P\, n_T)$, and $O(n_P\, l_T \log\log n_T)$, respectively. Here n_S and l_S are the number of nodes and leaves in tree S, respectively.

Union-Find with Deletions A classical union-find data structure maintains a collection of disjoint sets under *makeset*, *union* and *find* operations. In the union-find with deletions problem elements of the sets maintained may be deleted. We give a modification of the classical union-find data structure that supports *delete*, as well as *makeset* and *union*, in *constant* time, while still supporting *find* in $O(\log n)$ worst-case time and $O(\alpha(n))$ amortized time. Here n is the number of elements in the set returned by the *find* operation, and $\alpha(n)$ is a functional inverse of Ackermann's function.

Asymmetry in k-Center Variants Given a complete graph on n vertices with nonnegative (but possibly infinite) edge costs, and a positive integer k, the *k-center problem* is to find a set of k vertices, called *centers*, minimizing the maximum distance to any vertex and from its nearest center. We examine variants of the *asymmetric k*-center problem.

We provide an $O(\log^* n)$-approximation algorithm for the asymmetric *weighted* k-center problem. Here, the vertices have weights and we are given a total budget for opening centers. In the *p-neighbor* variant each vertex must have p (unweighted) centers nearby: we give an $O(\log^* k)$-bicriteria algorithm using $2k$ centers, for small p. In k-center with minimum coverage, each center is required to serve a minimum of clients. We give an $O(\log^* n)$-approximation algorithm for this problem. We also show that the following three versions of the asymmetric k-center problem are inapproximable: *priority k-center*, *k-supplier*, and *outliers with forbidden centers*.

Finite Capacity Dial-a-Ride Given a collection of objects in a metric space, a specified destination point for each object, and a vehicle with a capacity of at most k objects, the *finite capacity dial-a-ride problem* is to compute a shortest tour for the vehicle in which all objects can be delivered to their destinations while ensuring that the vehicle carries at most k objects at any point in time. In the *preemptive* version of the problem an object may be dropped at intermediate locations and then picked up later by the vehicle and delivered.

We study the hardness of approximation of the preemptive finite capacity dial-a-ride problem. Let N denote the number of nodes in the input graph, i.e., the number of points that are either sources or destinations. We show that the preemptive Finite Capacity Dial-a-Ride problem has no $\min\{O(\log^{1/4-\varepsilon} N), k^{1-\varepsilon}\}$-approximation algorithm for any constant $\varepsilon > 0$ unless all problems in NP can be solved by randomized algorithms with expected running time $O(n^{\mathrm{polylog} n})$.

Acknowledgments

Stephen Alstrup and Theis Rauhe were my advisors in the first 3 years of my PhD studies until they went on leave to start up their own company. I wish to thank them for their support, advice, and inspiration, and for recruiting me to the IT University. Anna Östlin Pagh and Lars Birkedal were my advisors in the last year of my PhD. I want to thank both of them for their support and advice.

I am especially grateful to Moses Charikar for all the time he has spent introducing me to the area of approximation algorithms. Part II of this dissertation is done under his supervision, and—although he had no official obligations—he acted as an advisor for me while I was visiting Princeton University. It has been a pleasure to work with him during my visits.

A special thanks goes to Mikkel Thorup, who has provided helpful guidance over the years, and for being the one evoking my interest in algorithms.

Thank you to Morten Heine B. Sørensen for introducing me to the world of research.

I also want to thank my excellent co-authors: Anthony Wirth, Uri Zwick, Philip Bille, Mikkel Thorup, and Gerth Stølting Brodal.

I am most grateful to Bernard Chazelle for being my host when I was visiting Princeton University the first time. I also want to thank all the people I met at Princeton who made my stay a very pleasant experience.

Thank you to the people at the IT University, especially the people in the Department of Theoretical Computer Science, for creating a pleasant working environment.

I want to thank Matthew Andrews, Christian Worm Mortensen, and Martin Zachariazen for useful discussions.

I thank the anonymous reviewers of my papers for their helpful suggestions.

Finally, I want to thank all the people who have proof-read parts of this dissertation: Martin Zachariasen, Anthony Wirth, Philip Bille, Jesper Gørtz, Søren Debois, and Rasmus Pagh.

Contents

Abstract iii

Acknowledgments v

I Data Structures and Algorithms on Trees 1

1 Introduction to Part I 3

1.1 Overview . . . 3
1.2 Models of Computation . . . 4
1.3 Prior Publication . . . 5
1.4 On Chapter 2: Binary Dispatching . . . 5
1.5 On Chapter 3: Tree Inclusion . . . 12
1.6 On Chapter 4: Union-find with Deletions . . . 15

2 Binary Dispatching 23

2.1 Introduction . . . 23
2.2 Preliminaries . . . 27
2.3 The Bridge Color Problem . . . 29
2.4 A Data Structure for the Bridge Color Problem . . . 31

3 Tree Inclusion 39

3.1 Introduction . . . 39
3.2 Notation and Definitions . . . 43
3.3 Computing Deep Embeddings . . . 46
3.4 A Simple Tree Inclusion Algorithm . . . 49
3.5 A Faster Tree Inclusion Algorithm . . . 56

4 Union-Find with Deletions 69

4.1 Introduction . . . 69
4.2 Preliminaries . . . 73

4.3 Augmenting Worst-Case Union-Find with Deletions 75
4.4 Faster Amortized Bounds 81

II Approximation Algorithms 87

5 Introduction to Part II 89
5.1 Overview 89
5.2 Approximation Algorithms 90
5.3 Prior Publication 90
5.4 On Chapter 6: Asymmetry in k-Center Variants 91
5.5 On Chapter 7: Dial-a-Ride 95

6 Asymmetry in k-Center Variants 103
6.1 Introduction 103
6.2 Definitions 107
6.3 Asymmetric k-Center Review 108
6.4 Asymmetric Weighted k-Center 109
6.5 Asymmetric p-Neighbor k-Center 114
6.6 Inapproximability Results 117
6.7 Asymmetric k-Center with Minimum Coverage 120

7 Finite Capacity Dial-a-Ride 125
7.1 Finite Capacity Dial-a-Ride 125
7.2 Relation between Buy-at-Bulk and Dial-a-Ride 129
7.3 The Network 130
7.4 Hardness of Buy-at-Bulk with Cost Function $\lceil \frac{x}{k} \rceil$ 134
7.5 Routing in the Network 142
7.6 Hardness of Preemptive Dial-a-Ride 148

8 Future Work 151
8.1 Multiple Dispatching 151
8.2 Tree Inclusion 152
8.3 Union-find with Deletions 152
8.4 Asymmetric k-Center 152
8.5 Dial-a-Ride 153

Part I

Data Structures and Algorithms on Trees

Chapter 1

Introduction to Part I

The papers in this part of the dissertation all concerns trees and data structures. Three problems are studied in this part: The *binary dispatching problem* (Chapter 2), the *tree inclusion problem* (Chapter 3), and the *union-find with deletions problem* (Chapter 4).

1.1 Overview

In this section we will give a short overview of the problems studied in this part of the dissertation.

Binary Dispatching The *dispatching problem* for object oriented languages is the problem of determining the most specialized method to invoke for calls at run-time. This can be a critical component of execution performance. The *unary dispatching problem* is equivalent to the *tree color problem*: Given a rooted tree T, where each node has zero or more colors, construct a data structure that supports the query *firstcolor*(v, c), that is, to return the nearest ancestor of v with color c (this might be v itself).

The *binary dispatching problem* can be seen as a 2-dimensional generalization of the tree color problem which we call the *bridge color problem*. In Chapter 2 we give a linear space data structure for *binary* dispatching that supports dispatching in logarithmic time. To solve the bridge color problem we turn it into a dynamic tree color problem, which is then solved persistently.

Tree Inclusion Given two rooted, ordered, and labeled trees P and T the *tree inclusion problem* is to determine if P can be obtained from T by deleting nodes in T.

In Chapter 3 we present a new approach to the tree inclusion problem. This leads to a new algorithm that use optimal linear space and has subquadratic running time or even faster when the number of leaves in one of the trees is small.

The running time of our tree inclusion algorithm depends on the *tree color problem*, which we also used in our data structure for the binary dispatching problem. We show a general connection between a data structure for the tree color problem and the tree inclusion problem. To achieve subquadratic running time we divide T into small trees or forests, called *micro trees* or *clusters*, of logarithmic size which overlap with other micro trees in at most two nodes. Each micro tree is represented by a constant number of nodes in a *macro tree*. The nodes in the macro tree are then connected according to the overlap of the micro tree they represent. We show how to efficiently preprocess the micro trees and the macro tree such that queries in the micro trees can be performed in constant time. This cuts a logarithmic factor off the quadratic running time.

Union-Find with Deletions A classical union-find data structure maintains a collection of disjoint sets under the operations *makeset*, *union* and *find*. In the *union-find with deletions problem* elements of the sets maintained may be deleted. In Chapter 4 we give a data structure for the union-find with deletions problem that supports *delete*, as well as *makeset* and *union*, in *constant* time, while still supporting *find* in $O(\log n)$ worst-case time and $O(\bar{\alpha}(\lfloor \frac{M+N}{N} \rfloor, n))$ amortized time. Here n is the number of elements in the set returned by the *find* operation, and $\bar{\alpha}(\cdot,\cdot)$ is a functional inverse of Ackermann's function. Our data structure, like most other union-find data structures, maintains the elements of each set in a rooted tree.

1.2 Models of Computation

The models of computation considered are the RAM *model* and the *pointer machine model*. These are briefly described below.

RAM Model A random access machine (RAM) has a memory which comprises an unbounded sequence of registers, each of which is capable of holding an integer. Arithmetic operations are allowed to compute the address of a memory register. On a unit cost RAM with logarithmic word size the size of a register is bounded by $O(\log n)$, where n is the input problem size. It can perform arithmetic operations such as addition, comparison, and multiplication in constant time. A more formal definition can be found in the book by Aho *et al.* [1].

Pointer Machine A pointer machine has a memory consisting of an unbounded collection of registers. Each register is a record with a finite number of named fields. The memory can be modelled as a directed graph with bounded degree. No arithmetic is allowed to compute the address of a node. The only possibility to access a node is to follow pointers.

The results in Chapter 2 and 3 rely on a unit-cost RAM with logarithmic word-size. The results in Chapter 4 also hold on a pointer machine.

1.3 Prior Publication

The results in this part of the dissertation have all been published or accepted for publication:

1. "Time and Space Efficient Multi-Method Dispatching".
   Stephen Alstrup, Gerth Stølting Brodal, Inge Li Gørtz, and Theis Rauhe.
   Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT) 2002.

2. "The Tree Inclusion Problem: In Optimal Space and Faster".
   Philip Bille and Inge Li Gørtz.
   Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP) 2005.

3. "Union-Find with Constant Time Deletions".
   Stephen Alstrup, Inge Li Gørtz, Theis Rauhe, Mikkel Thorup, and Uri Zwick.
   Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP) 2005.

In the following we will refer to these papers as paper 1, 2, and 3.

1.4 On Chapter 2: Binary Dispatching

In Chapter 2 we consider the *binary dispatching* problem. The chapter is an extended version of paper 1. In this section we formally define the problem, discuss its applications, and relate our results to other work. The result is achieved using a novel application of fully persistence, we believe is of independent interest.

1.4.1 Multiple Dispatching

In object oriented languages the modular units are abstract data types called *classes* and *selectors*. Each selector has possibly multiple implementations—denoted *methods*—each in a different class. The classes are arranged in a class hierarchy, and a class can inherit methods from its superclasses (classes above it in the class hierarchy). Therefore, when a selector s is invoked in a class c, the relevant method for s inherited by class c has to be determined. The *dispatching problem* is to determine the most specialized method to invoke for a method call. This specialization depends on the actual arguments of the method call at run-time and can be a critical component of execution performance in object oriented languages. Most object oriented languages rely on dispatching of methods with a single argument, but multi-method dispatching—where the methods take more than one argument—is used in object oriented languages such as Cecil [25], CLOS [24], Dylan [38], and MultiJava [36, 46].

Formally, let T be a rooted tree denoting the class hierarchy. Each node in T corresponds to a class, and T defines a partial order $\preceq$ on the set of classes:

$$A \preceq B \iff A \text{ is an ancestor of B (not necessarily a proper ancestor).}$$

If A is a proper ancestor of B we write $A \prec B$. Similarly, $B \succeq A$ ($B \succ A$) if B is a (proper) descendant of A. Let $\mathcal{M}$ be the set of methods. Each method takes a number of classes as arguments. A method invocation is a query of the form $s(A_1, \dots, A_d)$ where s is the name of a method in $\mathcal{M}$ and $A_1, \dots, A_d$ are class instances. Let $s(A_1, \dots, A_d)$ be such a query. We say that

$$s(B_1, \dots, B_d) \text{ is \textit{applicable} for } s(A_1, \dots, A_d) \iff B_i \preceq A_i \text{ for all } i \in \{1, \dots, d\} .$$

The *most specialized method* for a query $s(A_1, \dots, A_d)$ is the method $s(B_1, \dots, B_d)$ such that

1. $s(B_1, \dots, B_d)$ is applicable for $s(A_1, \dots, A_d)$,
2. for every other method $s(C_1, \dots, C_d)$ applicable for $s(A_1, \dots, A_d)$ we have $C_i \preceq B_i$ for all i.

There might not be a most specialized method, i.e., we might have two applicative methods $s(B_1, \dots, B_d)$ and $s(C_1, \dots, C_d)$ where $B_i \prec C_i$ and $C_j \prec B_j$ for some indices $1 \leq i, j \leq d$. That is, neither method is more specialized than the other. Multi-method dispatching is to find the most specialized applicable method in $\mathcal{M}$ if it exists. If it does not exist or in case of ambiguity, “no applicable method” resp. “ambiguity” is reported instead.

The *d-ary dispatching problem* is to construct a data structure that supports multi-method dispatching with methods having up to d arguments, where $\mathcal{M}$ is static but queries are online. The cases $d = 1$ and $d = 2$ are the *unary* and *binary dispatching* problems respectively.

Let N be the number of nodes in T, i.e., the number of classes. Let m denote the number of methods and M the number of distinct method names in $\mathcal{M}$.

Unary Dispatching and the Tree Color Problem

In the tree color problem we are given a tree T. Each node in T can have zero or more colors from a set of colors C. The problem is to support the query *firstcolor*(v, c), that is, to return the nearest ancestor of v with color c (this might be v itself). The tree color problem is the same as the unary dispatching problem ($d = 1$) if we let colors represent the method names.

The unary dispatching problem/tree color problem has been studied by a number of people.

The best known result using linear space for the unary dispatching problem (or static tree color problem) due to Muthukrishnan and Müller [93] is $O(\text{loglog} N)$ query time with expected linear preprocessing time. The expectation in the preprocessing time is due to perfect hashing in a van Emde Boas predecessor data structure [120, 121]. Since the tree color data structure is static it is possible to get rid of the expectation using the deterministic dictionary by Hagerup *et al.* [63] together with a simple two-level approach (see e.g. [119]).

The tree color problem has also been studied in the dynamic setting. Here we have the update operations *color*(v, c) and *uncolor*(v, c), which add and removes the color c from v's set of colors, respectively. In the unary dispatching problem this corresponds to adding and removing methods. Alstrup *et al.* [7] showed how to solve the dynamic tree color problem with expected update time $O(\text{loglog } N)$ for both *color*(v, c) and *uncolor*(v, c), and query time $= (\log N/\text{loglog } N)$, using linear space and preprocessing time. Alstrup *et al.* also showed how to add the update operations *AddLeaf* in amortized constant time and *RemoveLeaf* in worst case constant time while maintaining the time bounds on the other operations. In the unary dispatching problem this corresponds to adding or removing classes in the bottom of the class hierarchy.

Dietz [41] showed how to solve the incremental tree color problem in expected amortized time $O(\text{loglog} N)$ for *color* and expected amortized time $O(\text{loglog} N)$ per query using linear space, when the nodes are inserted top-down and each node has exactly one color.

In all the above algorithms the expected preprocessing and/or update times are due to hashing.

1.4.2 Persistent Data Structures and the Plane Sweep Technique

Before we state our results for the binary dispatching problem we will introduce the concept of persistence and describe the plane sweep technique, where partial persistence is used to turn a static d dimensional problem into a dynamic $d-1$ dimensional problem. Several of the earlier and related results on binary dispatching uses the plane sweep technique. To construct our data structure for the binary dispatching problem we use a technique similar to the plane sweep technique, but we use full persistence instead of partial persistence.

Persistence

An update operation on a data structure can be seen as generating a new version of the data structure. Data structures that one encounters in traditional algorithmic settings are *ephemeral* in the sense that an update operation destroys the old version of the data structure, leaving only the new one. In a *persistent* data structure all previous versions of the data structure can be queried. The concept of persistent data structures was introduced by Driscoll *et al.* [44].

We distinguish between two different types of persistence: partial persistence and full persistence. A data structure is *partially persistent* if all versions can be queried but only the newest one can be updated. A data structure is *fully persistent* if every version can be both queried and updated.

In addition to its ephemeral arguments a persistent update or query takes as an argument the version of the data structure to which the query or update refers. The *version graph* is a directed graph where each node corresponds to a version of the data structure and there is an edge from node v_1 to a node v_2 if and only if v_2 was created by an update operation to v_1. The version graph for a partially persistent data structure is a path, for a fully persistent data structure it is a tree, and for a confluently persistent data structure it is a directed acyclic graph (DAG).

Making Data Structures Persistent In the following let m denote the number of versions. Driscoll *et al.* [44] showed how to make any ephemeral data structure on a RAM partially persistent with slowdown $O(\log m)$ for both updates and queries. The extra space cost is $O(1)$ per ephemeral memory modification. Using this method together with the van Emde Boas predecessor data structure [120] and dynamic perfect hashing [43] gives slowdown $O(\log\log m)$ per query and expected

slowdown $O(\text{loglog}m)$ per update with extra space cost $O(1)$ per ephemeral memory modification [71].

Dietz [41] showed how to make any ephemeral data structure on a RAM fully persistent with $O(\text{loglog}\ m)$ slowdown for queries and expected amortized $O(\text{loglog}\ m)$ slowdown for updates. The extra space cost is $O(1)$ per ephemeral memory modification.

For bounded degree linked data structures it is possible to get better bounds. Driscoll *et al.* [44] showed how to make any such data structure partially or fully persistent with worst-case slowdown $O(1)$ for queries, amortized slowdown $O(1)$ for updates, and amortized $O(1)$ extra space cost per memory modification.

For more about how to implement a persistent data structure, and applications of persistent data structures, see the surveys by Kaplan [72] and Italiano and Raman [71].

Plane Sweep Technique

In the plane sweep technique partial persistence is used to turn a dynamic d-dimensional data structure into a static $d+1$ dimensional data structure.

This technique was first used by Sarnak and Tarjan [104] to give an algorithm for the planar point location problem. In the planar point location problem we are given a subdivision of the Euclidian plane into polygons by a collection of n line segments which intersect only at their endpoints. The goal is to construct a data structure such that, given a query point we can efficiently determine the polygon containing it.

Sarnak and Tarjan construct the data structure as follows. Imagine moving an infinite line—denoted the sweep line—across from left to right, beginning at the leftmost endpoint of any line segment. As the sweep line moves, the line segments currently intersecting it are maintained in a partially persistent balanced binary search tree in order of their intersection with the sweep line. The plane is divided into vertical slabs, within which the search tree does not change.

Given a query point q, first locate the slab in which the x-coordinate lies, and then query this version of the partially persistent search tree to find the two line segments immediately above and below q in this slab. This uniquely determines the polygon in which q lies.

Sarnak and Tarjan showed how to implement a partially persistent search tree with worst-case $O(\log n)$ query and update time, and an amortized $O(1)$ space cost per update. This gives a data structure for the planar point location problem using $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time.

1.4.3 Our Results and Techniques

Our main result is a data structure for the binary dispatching problem which is of "particular interest" quoting Ferragina *et al.* [50]. Our data structure uses $O(m)$ space and query time $O(\log m)$ on a unit-cost RAM with word size logarithmic in N with $O(N + m \operatorname{loglog} m)$ time for preprocessing.

We reduce the binary dispatching problem to a problem we call the *bridge color problem*. This is a generalization of the tree color problem. In the bridge color problem we are given two rooted trees T_1 and T_2, and a set of edges—called *bridges*—connecting nodes in T_1 to nodes in T_2. Each bridge has a color from a set of colors C. A bridge is a triple $(c, v_1, v_2) \in C \times V(T_1) \times V(T_2)$ and is denoted by $c(v_1, v_2)$. The *bridge color problem* is to construct a data structure which supports the query *firstcolorbridge*(c,v_1,v_2).

firstcolorbridge(c, v_1, v_2) Find a bridge $c(w_1, w_2)$ such that:

1. $w_1 \preceq v_1$ and $w_2 \preceq v_2$.
2. There is no other bridge $c(w_1', w_2')$ such that $w_1 \prec w_1' \preceq v_1$ or $w_2 \prec w_2' \preceq v_2$.

If there is no bridge satisfying the first condition return NIL. If there is a bridge satisfying the first condition but not the second then return "ambiguity".

The binary dispatching problem can be reduced to the bridge color problem the following way. Let T_1 and T_2 be copies of the tree T in the binary dispatching problem. For every method $s(v_1, v_2) \in \mathcal{M}$ make a bridge of color s between $v_1 \in V(T_1)$ and $v_2 \in V(T_2)$.

We solve the bridge color problem by constructing two fully persistent data structures for the dynamic tree color problem. For each node v in T_1 we have a version of the tree color data structure for T_2. That is, T_1 can be seen as the version tree. For version v ($v \in V(T_1)$) a node $u \in V(T_2)$ has color c if and only if there is a bridge of color c from an ancestor of v to u. We similarly construct a fully persistent tree color data structure for T_1 with T_2 corresponding to the version tree. We can then answer *firstcolorbridge* queries by performing a constant number of persistent *firstcolor* queries.

Our technique can be seen as the fully persistent analogue to the partial persistent plane sweep technique. It has been referred to by Kaplan [72] as one of the few interesting applications of fully persistent data structures.

In the data structure described above we need a data structure that supports insert and predecessor queries on a set of integers from $\{1, \ldots, n\}$. This can be

solved in worst case $O(\text{loglog}\, n)$ time per operation on a RAM using a data structure of van Emde Boas [120]. We show how to do modify this data structure such that it only uses worst case $O(1)$ memory modifications per update.

1.4.4 Previous Results and Related Work

For the d-ary dispatching, $d \geq 2$, the result of Ferragina *et al.* [50] is a data structure using space $O(m\ (t\ \text{log}m/\text{log}t)^{d-1})$ and query time $O((\text{log}m/\text{log}t)^{d-1}\text{loglog}N)$, where t is a parameter $2 \leq t \leq m$. For the case $t = 2$ they are able to improve the query time to $O(\text{log}^{d-1}m)$ using fractional cascading [30]. They obtain their results by reducing the dispatching problem to a point-enclosure problem in d dimensions: Given a point q, check whether there is a smallest rectangle containing q. In the context of the geometric problem, Ferragina *et al.* also present applications to approximate dictionary matching.

Packet Classification Problem Eppstein and Muthukrishnan [47] looked at a similar problem called *packet classification*. Here there is a database of m filters available for preprocessing. A packet filter i in an IP network is a collection of d-dimensional ranges $[l_i^1, r_i^1] \times \cdots \times [l_i^d, r_i^d]$, an action A_i, and a priority p_i. An IP packet P is a d-dimensional vector of values $[P_1, \ldots, P_d]$. A filter i applies to packet P if $P_j \in [l_i^j, r_i^j]$ for $j = 1, \ldots, d$.

The packet classification problem is given a packet P to determine the filter of highest priority that applies to P.

The ranges of the different filters are typically nested [47]. That is, if two ranges intersect, one is completely contained in the other. In this case the packet classification problem is essentially the same as the multiple dispatching problem.

For the case $d = 2$ Eppstein and Muthukrishnan gave an algorithm using space $O(m^{1+o(1)})$ and query time $O(\text{loglog}\, m)$, or $O(m^{1+\varepsilon})$ and query time $O(1)$. They reduced the problem to a geometric problem, very similar to the one in [50]. To solve the problem they used the plane-sweep approach to turn the static two-dimensional rectangle query problem into a partially persistent dynamic one-dimensional problem.

Later Results In 2004 Kwok and Poon [99] gave an algorithm for the binary dispatching problem with the same time and space bounds as ours. They reduce the problem to a point enclosure problem on a 2-dimensional grid the same way as Ferragina *et al.* [50], and then apply the plane sweep technique. Kwok and Poon

claim their algorithm is simpler than ours because they use partial persistence instead of full persistence.

Related Work Kwok and Poon [99] also study two related problems: *2-d point enclosure for nested rectangles* and *2-dimensional packet classification problem for conflict resolved filters*. In the point enclosure for nested rectangles problem the set of rectangles is nested and the problem is to find the most specific rectangle enclosing the query point. A set of rectangles is nested if any two rectangles from the set either have no intersection or one is completely contained in the other. The 2-dimensional packet classification problem for conflict resolved filters is the same as the problem studied by Eppstein and Muthukrishnan [47] except all conflicts are resolved, that is, all packets has a unique most specific matching filter and thus there is no need to check for ambiguity. For both problems Kwok and Poon give a linear space algorithm with $O(\log\log^2 m)$ query time.

Thorup [119] studies the dynamic stabbing problem in one or more dimensions. His goal is to get very fast query time trading it for slow update time, but still keeping linear space.

1.5 On Chapter 3: Tree Inclusion

In Chapter 3 we consider the *tree inclusion* problem. The chapter is the full version of paper 2 and is a minor revision of the technical report [22]. In this section we formally define the problem, discuss its applications, and relate our results to other work.

1.5.1 Tree Comparison

Let T be a rooted tree. We say that T is *labeled* if each node is assigned a symbol from an alphabet Σ and we say that T is *ordered* if a left-to-right order among siblings in T is given. All trees in this chapter are rooted and labeled.

Comparison of trees occurs in several diverse areas such as computational biology, structured text databases, image analysis, automatic theorem proving, and compiler optimization [111, 128, 80, 82, 69, 101, 129]. Many different ways to compare trees have been devised.

Tree Inclusion A tree P is *included* in T, denoted $P \sqsubseteq T$, if P can be obtained from T by deleting nodes of T. Deleting a node v in T means making the children of v children of the parent of v and then removing v. The children are inserted in

the place of v in the left-to-right order among the siblings of v. The *tree inclusion problem* is to determine if P can be included in T and if so report all subtrees of T that include P. The tree P and T is often called the *pattern* and *target*, respectively.

In this chapter we consider the ordered tree inclusion problem. For some applications considering *unordered* trees is more natural. However, this problem has been proved to be NP-complete [91, 80], whereas the ordered version can be solved in polynomial time. From now on when we say tree inclusion we refer to the ordered version of the problem.

Related Tree Comparison Problems In the *tree pattern matching problem* [69, 84, 45, 37] the goal is to find an injective mapping f from the nodes of P to the nodes of T such that for every node v in P the ith child of v is mapped to the ith child of $f(v)$. The tree pattern matching problem can be solved in $O(n \log^{O(1)} n)$ time, where $n = n_P + n_T$. Another similar problem is the *subtree isomorphism* problem [33, 108], which is to determine if T has a subgraph which is isomorphic to P. Unlike tree inclusion the subtree isomorphism problem can be solved efficiently for unordered trees. The best algorithms for the subtree isomorphism problem use $O(n_P^{1.5}\, n_T/ \log n_P)$ for unordered trees and $O(n_P\, n_T/ \log n_P)$ time ordered trees [33, 108]. Both use $O(n_P\, n_T)$ space.

The tree inclusion problem can be considered a special case of the *tree edit distance problem* [111, 128, 81]. Here one wants to find the minimum sequence of insert, delete, and relabel operations needed to transform P into T. Inserting a node v as a child of v' in T means making v the parent of a consecutive subsequence of the children of v'. A relabeling of a node changes the label of a node. The currently best worst-case algorithm for this problem uses $O(n_P^2\, n_T \log n_T)$ time. The unordered tree edit distance is MAX SNP-hard [14]. For more details and references see the survey [21].

1.5.2 Applications

Recently, the tree inclusion problem has been recognized as an important query primitive for XML data and has received considerable attention, see *e.g.*, [105, 125, 124, 127, 106, 118]. The key idea is that an XML document can be viewed as an ordered, labeled tree and queries on this tree correspond to a tree inclusion problem. The ordered tree edit distance problem has been used to compare XML documents [94].

1.5.3 Results

The tree inclusion problem was initially introduced by Knuth [83, exercise 2.3.2-22] who gave a sufficient condition for testing inclusion. Motivated by applications in structured databases [79, 90] Kilpeläinen and Mannila [80] presented the first polynomial time algorithm using $O(n_P\ n_T)$ time and space, where n_P and n_T is the number of nodes in a tree P and T, respectively. The main idea behind this algorithm is following: Let $v \in V(P)$ and $w \in V(T)$ be nodes with children $v_1, \ldots, v_i$ and $w_1, \ldots, w_j$, respectively. To decide if $P(v)$ can be included $T(w)$ we try to find a sequence of numbers $1 \leq x_1 < x_2 < \cdots < x_i \leq j$ such that $P(v_k)$ can be included in $T(w_{x_k})$ for all k, $1 \leq k \leq i$. If we have already determined whether or not $P(v_s) \sqsubseteq T(w_t)$, for all s and t, $1 \leq s \leq i$, $1 \leq t \leq j$, we can efficiently find such a sequence by scanning the children of v from left to right. Hence, applying this approach in a bottom-up fashion we can determine, if $P(v) \sqsubseteq T(w)$, for all pairs $(v, w) \in V(P) \times V(T)$.

During the last decade several improvements of the original algorithm of [80] have been suggested [78, 2, 103, 31]. The previously best known bound is due to Chen [31] who presented an algorithm using $O(l_P\, n_T)$ time and $O(l_P \min\{d_T, l_T\})$ space. Here, l_S and d_S denotes the number of leaves of and the maximum depth of a tree S, respectively. This algorithm is based on an algorithm of Kilpeläinen [78]. Note that the time and space is still $\Theta(n_P\, n_T)$ for worst-case input trees.

Our Results We improve all of the previously known time and space bounds. We give three algorithms that all uses linear space and runs in $O(\frac{n_P\, n_T}{\log n_T})$, $O(l_P\, n_T)$, and $O(n_P\, l_T \log\log n_T)$, respectively.

Hence, for worst-case input this improves the previous time and space bounds by a logarithmic and linear factor, respectively. When P has a small number of leaves the running time of our algorithm matches the previously best known time bound of [31] while maintaining linear space. In the context of XML databases the most important feature of our algorithms is the space usage. This will make it possible to query larger trees and speed up the query time since more of the computation can be kept in main memory.

Our Techniques In this paper we take a different approach than the previous algorithms. The main idea is to construct a data structure on T supporting a small number of procedures, called the *set procedures*, on subsets of nodes of T. We show that any such data structure implies an algorithm for the tree inclusion problem. We consider various implementations of this data structure which all

use linear space. The first one gives an algorithm with $O(l_P\ n_T)$ running time. As it turns out, the running time depends on the *tree color problem*. We show a general connection between a data structure for the tree color problem and the tree inclusion problem. Plugging in a data structure of Muthukrishnan and Müller [93] we obtain an algorithm with $O(n_P\ l_T \log\log n_T)$ running time.

Based on the simple algorithms above we show how to improve worst-case running the time of the set procedures by a logarithmic factor. The general idea used to achieve this is to divide T into small trees or forests, called *micro trees* or *clusters* of logarithmic size which overlap with other micro trees in at most 2 nodes. This can be done in linear time using a technique by Alstrup *et al.* [5]. Each micro tree is represented by a constant number of nodes in a *macro tree*. The nodes in the macro tree are then connected according to the overlap of the micro tree they represent. We show how to efficiently preprocess the micro trees and the macro tree such that the set procedures use constant time for each micro tree. Hence, the worst-case running time is improved by a logarithmic factor.

1.6 On Chapter 4: Union-find with Deletions

In Chapter 4 we consider the *union-find with deletions* problem. The chapter is a revision of paper 3. In this section we formally define the problem, discuss its applications, and relate our results to other work.

1.6.1 Union-Find

A union-find data structure maintains a collection of disjoint sets under the operations *makeset*, which creates a new set, *union*, which combines two sets into one, and *find*, which locates the set containing an element. More formally, a classical union-find data structure allows the following operations on a collection of disjoint sets:

- *makeset*(x): Create a set containing the single element x, and return the name of the set.
- *union*(A, B): Combine the sets A and B into a new set, destroying sets A and B.
- *find*(x): Find and return (the name of) the set that contains x.

The union-find data structure can also be seen as a data structure maintaining an equivalence relation, i.e., elements are in the same set if they are equivalent

according to the equivalence relation. To find out if two elements a and b are equivalent, compare *find*(a) and *find*(b). The two elements are equivalent if the names of the sets returned by the *find* operations are the same.

The union-find data structure has many applications in a wide range of areas. For example, in finding minimum spanning trees [1], finding dominators in graphs [113], and in checking flow reducibility [112]. For an extensive list of such applications, and more information on the problem and many of its variants, see the survey of Galil and Italiano [58].

Kaplan *et al.* [75] studied the union-find with deletions problem, in which elements may be deleted. In the *union-find with deletions problem*—or *union-find-delete* for short—we, in addition to the three operations above allow a delete operation:

- *delete*(x): Deletes x from the set containing it.

Note that a delete operation does not get the set containing x as a parameter.

A union-find-delete data structure can be used in any applications where an equivalence relation is needed on a collection over a dynamic set of items. One such application is in implementation of meldable heaps [74].

1.6.2 Classical Union-Find Data Structures

Most union-find data structure represents the sets as rooted trees, with the nodes representing elements.

A simple union-find data structure (attributed by Aho *et al.* [1] to McIlroy and Morris), which employs two simple heuristics, *union by rank* and *path compression*, was shown by Tarjan [114] (see also Tarjan and van Leeuwen [117]) to be very efficient.

Union by Rank The *union by rank* heuristic keeps the trees shallow as follows. Each node is given a rank, which is an upper bound on its height. The rank of a set is the rank of the root of the tree representing the set. When performing *makeset* the rank is defined to be zero. When performing a *union* of two sets, the root with the lowest rank is made a child of the root with the highest rank. If both sets have the same rank, an arbitrary one is made the root, and the rank of the new root is increased by one.

The union by rank heuristics on its own implies that *find* operations take $O(\log n)$ worst-case time. Here n is the number of elements in the set returned by the *find* operation. All other operations take constant worst-case time. It is

possible to trade a slower *union* for a faster *find*. Smid [109], building on a result of Blum [23], gave for any k a data structure that supports *union* in $O(k)$ time and *find* in $O(\log_k n)$ time. When $k = \log n/\log\log n$, both *union* and *find* take $O(\log n/\log\log n)$ time. Fredman and Saks [57] (see also Ben-Amram and Galil [19]) showed that this tradeoff is optimal, i.e., that any algorithm for the union-find problem requires $\Omega(\log n/\log\log n)$ single-operation worst-case time in the cell probe model[1]. More generally, Alstrup *et al.* [3] showed that $t_q = \Omega(\log n/\log t_u)$, where t_q is the worst-case query time and t_u is the worst-case update time. This matches the upper bounds given by Smid.

Path Compression The *path compression* heuristic changes the structure of the tree during a *find* by moving nodes closer to the root. When carrying out a *find*(x) operation all nodes on the path from x to the root are made children of the root.

Tarjan and van Leeuwen [117] showed the path compression heuristic alone runs in $O(N + M \log_{2+M/N} N))$ total time, where M is the number of *find* operations and N the number of *makeset* operations (hence there is at most $N-1$ *union* operations).

Tarjan [114] (see also Tarjan and van Leeuwen [117]) showed that the data structure using both union by rank and path compression performs a sequence of M *find* operations and N *makeset* and *union* operations in $O(N+M\,\alpha(M+N,N))$ total time. Here $\alpha(\cdot,\cdot)$ is an slowly growing function, which is the functional inverse of Ackermann's function (for a formal definition see Section 4.1.1). In other words, the *amortized* cost of each *makeset* and *union* operation is $O(1)$, while the amortized cost of each *find* operation is $O(\alpha(M+N,N))$, only marginally more than a constant. Fredman and Saks [57] obtained a matching lower bound in the cell probe model of computation, showing that this data structure is essentially optimal in the amortized setting. More precisely, they showed that in the cell probe model any union-find data structure requires $\Omega(M\,\alpha(M+N,N))$ time to execute M *find* operations and $N-1$ *union* operations, beginning with N singleton sets. Ben-Amram and Galil [19] gave a tradeoff between the amortized *find* time and the amortized *union* time in the cell probe model.

Path compression requires two passes over the find path, one to find the tree root and another to perform the compression. Tarjan and Leeuwen [117] studied the one-pass variants, *path halving* and *path splitting*, and showed that they

[1]In the cell probe model of computation introduced by Yao [126] the cost of computation is measured by the total number of memory accesses to a random access memory with b bits word size. All other computations are considered to be free. In the lower bounds in this chapter $b = \lceil \log n \rceil$.

also run in $O(N + M\,\alpha(M + N, N))$ time when combined with union by rank. Path halving works by making every other node on the find path a child of its grandparent. In path splitting every node on the find path is made a child of its grandparent.

Amortized versus Worst-Case Time Bounds Recall that Alstrup *et al.* [3] showed that the optimal tradeoff between the worst-case *find* time t_q and the worst-case *union* time t_u is $t_q = \Omega(\log n / \log t_u)$. They also showed that only if $t_q > \alpha(M + N, N)$ can this tradeoff be achieved simultaneously with the optimal amortized time of $\Theta(\alpha(M + N, N))$.

Alstrup *et al.* also present union-find algorithms with simultaneously optimal amortized and worst-case bounds. The algorithm is a modified version of the standard union-find algorithm. By performing some of the path compressions at *union* operations, instead of just at *find* operations, the simultaneously optimal amortized and worst-case bounds are obtained.

Local Amortized Bounds To state some more local amortized bounds, we need a non-standard parameterization of the inverse Ackermann function. Let $A_k(j)$ be the Ackermann function and define $\bar{\alpha}(i, j) = \min\{k \geq 2 \mid A_k(i) > j\}$, for integers $i, j \geq 0$. Relating to the standard definition of α, we have $\alpha(M, N) = \Theta(\bar{\alpha}(\lfloor M/N \rfloor, N))$.

Kaplan *et al.* [75] refined the analysis of the union-find data structure with union by rank and path compression, and showed that the cost of each find is proportional to the size of the corresponding set instead of the size of the universe. More precisely, they showed that the amortized cost of *find*(x) operation is only $O(\bar{\alpha}(\lfloor \frac{M+N}{N} \rfloor, n)$, where n is the number of elements in the set containing x.

1.6.3 Data Structures for Union-Find with Deletions

The challenge in designing a data structure for the union-find with deletions problem is to keep the time for a *find* operation proportional to the number of elements in the set and not the number of elements there ever was in the set, while maintaining linear space in the number of current elements in the data structure. A simple way to deal with deletions would be to just mark each node containing a deleted element as deleted. In some applications this might work, but not in general. The space usage for this data structure would be $O(N)$, where N is the number of elements ever created, and, moreover, the *find* time would be dependent on the number of elements there ever was in the set.

Using an incremental background rebuilding technique for each set, Kaplan *et al.* [75] described a way of converting any data structure for the classical union-find problem into a union-find-delete data structure. The time bounds for *make-set*, *find* and *union* change by only a constant factor, while the time needed for *delete*(x) operation is the same as the time needed for a *find*(x) followed by a constant number of *union*s with a singleton set. As a *union* operation is usually much cheaper than a *find* operation, Kaplan *et al.* [75] thus showed that in both the amortized and the worst-case settings, a *delete* operation is not more expensive than a *find* operation. Combined with their refined amortized analysis of the classical union-find data structure, this provides a union-find-delete data structure that implements *makeset* and *union* in constant time, and *find*(x) and *delete*(x) in $O(\bar{\alpha}(\lfloor \frac{M+N}{N} \rfloor, n))$ amortized time and $O(\log n)$ worst-case time. Their data structure uses an incremental global rebuilding technique [95]. Each set has two counters, one counting the number of elements in the set, and one counting the number of deleted elements in the set. At each *delete* perform a *find* to find the set containing the element and then increment the number of deleted elements in the set by one. When at least $1/4$ of the elements in a set is deleted each delete is followed by a constant number of rebuilding operations. The background rebuilding is done by maintaining two trees, T_S^1 and T_S^2, for each set S in the data structure. In the background rebuilding 4 elements are moved from T_S^1 to T_S^2 when performing a delete. Kaplan *et al.* showed that at any time at most $1/4$ of the elements in T_S^2 and at most half of the elements in T_S^1 are deleted. That is, the number of items in T_S^1 and T_S^2 are within a constant factor of the number of undeleted elements in the set S, and therefore the time it takes to perform *find* and *union* in the union-find-delete data structure is proportional to the time it takes to perform *find* and *union* in the underlying union-find data structure without deletions.

Kaplan *et al.* also gave another data structure for union-find-delete in the worst-case setting. This is a modification of Smid's data structure [109] with the same performance for union and find as Smid's, i.e., $O(k)$ time for *union* and $O(\log_k n)$ time for *find*, that supports delete in $O(\log_k n)$ time.

Kaplan *et al.* posed the question whether *delete* operations can be implemented *faster* than *find* operations (while keeping the space and time bounds dependent on the number of current elements in the data structure and the number of elements in the set, respectively).

1.6.4 Our Results

We solve the open problem raised by Kaplan *et al.* [75] and show that *delete* can be performed in *constant* worst-case time, while still keeping the $O(\bar{\alpha}(\lfloor\frac{M+N}{N}\rfloor, n))$ amortized cost and the $O(\log n)$ worst-case cost of *find*, and the constant worst-case cost of *makeset* and *union*. Here N is the total number of elements ever created, M is the total number of *find* operations performed, and n is the number of elements in the set returned by the *find* operation. The data structure that we present uses linear space and is a relatively simple modification of the classical union-find data structure. It uses local rebuilding in contrast to the data structure by Kaplan *et al.* that uses global rebuilding.

We also obtain a very concise potential-based proof of the $O(\bar{\alpha}(\lfloor\frac{M+N}{N}\rfloor, n))$ bound, first obtained by Kaplan *et al.* [75], on the amortized cost of a *find* in the classical setting. We believe that our potential-based analysis is simpler than the one given by Kaplan *et al.* [75].

In the next section we discuss various analyzes of union-find data structures.

1.6.5 Analysis of Union-Find

In general when considering union-find data structures, the data structure itself is simple, whereas the analysis can be involved.

The first tight amortized analysis of the classical union-find data structure, by Tarjan [114] and Tarjan and van Leeuwen [117], uses *multiple partitions* and the so-called *accounting method*. The refined analysis of Kaplan *et al.* [75] is directly based on this method. The *accounting method* is one of the standard methods to analyze the amortized running time of an algorithm using credits and debits. To perform an operation we are given a certain number of credits to spend. If we complete the operation before running out of credits we can save the unused credits for future operations. If we run out of credits before completing an operation we can borrow credits by creating debit pairs and spending the created credits. The corresponding debits remain in existence to account for our borrowing. We can use surplus credits to pay off existing debits. The total time for a sequence of operations is proportional to the total number of credits allocated for the operations plus the number of debits remaining when all the operations are complete [115].

Kozen [85] gave a simplified analysis of the classical union-find data structure. Based on this Tarjan [116] gave an analysis using potential functions. This type of analysis is also used in Chapter 21 of Cormen *et al.* [39]. The *potential function* method is another standard method to analyze the amortized cost of an algorithm. Here a potential function Φ maps each version of the data structure to

a non-negative integer potential. Let D_i be the data structure after the ith operation. Let c_i be the actual cost of the ith operation. The amortized cost $\hat{c}_i$ of the ith operation is then $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, and the total amortized costs of n operations is $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$. To ensure that the total amortized cost is an upper bound on the actual total cost, the potential function Φ is defined such that $\Phi(D_i) \geq \Phi(D_0)$, for all i.

Seidel and Sharir [107] recently presented a top-down amortized analysis of the union-find data structure. The analysis uses a divide-and-conquer approach to get recurrence relations from which the bounds follow. The bound follows without having to introduce the inverse Ackermann function in the proof.

Analysis of Our Data Structure The analysis of our data structure uses two different potential functions. The first potential function is used to bound the *worst-case* cost of *find* operations. Both potential functions are needed to bound the *amortized* cost of *find* operations. The second potential function on its own can be used to obtain a simple derivation of the refined amortized bounds of Kaplan *et al.* [75] for union-find without deletions, since it is bounding the cost of an amortized operation in terms of the size of the set returned by the operation.

1.6.6 Our Techniques

Our union-find-delete data structure, like most other union-find data structures, maintains the elements of each set in a rooted tree. As elements can now be deleted, not all the nodes in these trees contain elements. Nodes that contain elements are said to be *occupied*, while nodes that do not contain elements are said to be *vacant*. When an element is deleted, the node containing it becomes vacant. If proper measures are not taken, a tree representing a set may contain too many vacant nodes. As a result, the space needed to store the tree, and the time needed to process a *find* operation may become too large. Our data structure uses a simple collection of local operations to *tidy up* a tree after each delete operation. This ensures that at most half of the nodes in a tree are vacant. More importantly, the algorithm employs local constant-time *shortcut* operations in which the grandparent, or a more distant ancestor, of a node becomes its new parent. These operations, which may be viewed as a local constant-time variant of the path compression technique, keep the trees relatively shallow to allow fast *find* operations.

1.6.7 Earlier Results

In a previous version of the paper we showed how to obtain deletions in worst case $O(\log^* n)$ time, while still keeping $O(\log n)$ worst-case cost of *find* operations, and constant worst-case cost of *makeset* and *union* operations. This paper is available as a technical report [4].

Chapter 2

Binary Dispatching

The *dispatching problem* for object oriented languages is the problem of determining the most specialized method to invoke for calls at run-time. This can be a critical component of execution performance. A number of results, including [Muthukrishnan and Müller SODA'96, Ferragina and Muthukrishnan ESA'96, Alstrup *et al.* FOCS'98], have studied this problem and in particular provided various efficient data structures for the *mono-method* dispatching problem. A paper of Ferragina, Muthukrishnan and de Berg [STOC'99] addresses the *multi-method* dispatching problem.

Our main result is a linear space data structure for *binary* dispatching that supports dispatching in logarithmic time. Using the same query time as Ferragina *et al.* this result improves the space bound with a logarithmic factor.

2.1 Introduction

In object oriented languages the modular units are abstract data types called *classes* and *selectors*. Each selector has possibly multiple implementations—denoted *methods*—each in a different class. The classes are arranged in a class hierarchy, and a class can inherit methods from its superclasses (classes above it in the class hierarchy). Therefore, when a selector s is invoked in a class c, the relevant method for s inherited by class c has to be determined. The *dispatching problem* for object oriented languages is to determine the most specialized method to invoke for a method call. This specialization depends on the actual arguments of the method call at run-time and can be a critical component of execution performance in object oriented languages. Most of the commercial object oriented languages rely on dispatching of methods with only one argument, the so-called *mono-method* or *unary dispatching problem*. A number of papers, see e.g.,[49, 93]

(for an extensive list see [50]), have studied the unary dispatching problem, and Ferragina and Muthukrishnan [49] provide a linear space data structure that supports unary dispatching in log-logarithmic time. However, the techniques in these papers do not apply to the more general *multi-method dispatching problem* in which more than one method argument is used for the dispatching. Multi-method dispatching has been identified as a powerful feature in object oriented languages supporting multi-methods such as Cecil [25], CLOS [24], Dylan [38], and MultiJava [36, 46]. Several recent results have attempted to deal with d-ary dispatching in practice (see [50] for an extensive list). Ferragina *et al.* [50] provided the first non-trivial data structures, and, quoting this paper, several experimental object oriented languages' "ultimately success and impact in practice depends, among other things, on whether multi-method dispatching can be supported efficiently".

Our result is a *linear space* data structure for the *binary dispatching* problem, i.e., multi-method dispatching for methods with at most two arguments. Our data structure uses *linear space* and supports dispatching in logarithmic time. Using the same query time as Ferragina *et al.* [50], this result improves the space bound with a logarithmic factor. Before we provide a precise formulation of our result, we will formalize the general d-ary dispatching problem.

Definition 2.1.1 (Multiple Dispatching Problem)**.** Let T be a rooted tree denoting the class hierarchy. Each node in T corresponds to a class, and T defines a partial order $\preceq$ on the set of classes:

$$A \preceq B \quad \Longleftrightarrow \quad A \text{ is an ancestor of B (not necessarily a proper ancestor).}$$

If A is a proper ancestor of B we write $A \prec B$. Similarly, $B \succeq A$ ($B \succ A$) if B is a (proper) descendant of A. Let $\mathcal{M}$ be the set of methods. Each method takes a number of classes as arguments. A method invocation is a query of the form $s(A_1, \ldots, A_d)$ where s is the name of a method in $\mathcal{M}$ and $A_1, \ldots, A_d$ are class instances. Let $s(A_1, \ldots, A_d)$ be such a query. We say that

$$s(B_1, \ldots, B_d) \text{ is } \textit{applicable} \text{ for } s(A_1, \ldots, A_d) \quad \Longleftrightarrow \quad B_i \preceq A_i \text{ for all } i \in \{1, \ldots, d\} .$$

The *most specialized method* for a query $s(A_1, \ldots, A_d)$ is the method $s(B_1, \ldots, B_d)$ such that

1. $s(B_1, \ldots, B_d)$ is applicable for $s(A_1, \ldots, A_d)$,
2. for every other method $s(C_1, \ldots, C_d)$ applicable for $s(A_1, \ldots, A_d)$ we have $C_i \preceq B_i$ for all i.

There might not be a most specialized method, i.e., we might have two applicative methods $s(B_1, \ldots, B_d)$ and $s(C_1, \ldots, C_d)$ where $B_i \prec C_i$ and $C_j \prec B_j$ for some indices $1 \leq i, j \leq d$. That is, neither method is more specialized than the other. Multi-method dispatching is to find the most specialized applicable method in $\mathcal{M}$ if it exists. If it does not exist or in case of ambiguity, "no applicable method" resp. "ambiguity" is reported instead.

The *d-ary dispatching problem* is to construct a data structure that supports multi-method dispatching with methods having up to d arguments, where $\mathcal{M}$ is static but queries are online.

The cases $d = 1$ and $d = 2$ are called the *unary* and *binary dispatching* problems, respectively. Let N denote the number of classes in the class hierarchy, m the number of methods in $\mathcal{M}$, and M the number of distinct method names in $\mathcal{M}$.

In this paper we focus on the binary dispatching problem which is of "particular interest" quoting Ferragina *et al.* [50].

We assume that the size of T is $O(m)$. If this is not the case we can map nodes that does not participate in any method to their closest ancestor that does participate in some method in $O(n)$ time.

Results

Our main result is a data structure for the binary dispatching problem using $O(m)$ space and query time $O(\log m)$ on a unit-cost RAM with word size logarithmic in N with $O(N + m\ (\text{loglog}\ m)^2)$ time for preprocessing. By the use of a reduction to a geometric problem, Ferragina *et al.* [50], obtain similar time bounds within space $O(m \log m)$. Furthermore they show how the case $d = 2$ can be generalized for $d > 2$ at the cost of factor $\log^{d-2} m$ in the time and space bounds.

Our result is obtained by a very different approach in which we employ a dynamic to static transformation technique. To solve the binary dispatching problem we turn it into a unary dispatching problem — a variant of the marked ancestor problem as defined by Alstrup *et al.* [7], in which we maintain a dynamic set of methods. The unary problem is then solved persistently. We solve the persistent unary problem combining the technique by Dietz [41] to make a data structure fully persistent and the technique from [7] to solve the tree color problem. The technique of using a persistent dynamic one-dimensional data structure to solve a static two-dimensional problem is a standard technique [104]. What is new in our technique is that we use the class hierarchy tree to denote the time (give the order on the versions) to get a fully persistent data structure. This gives a "branching" notion for time, which is the same as what one has in a fully persistent data

structure where it is called the version tree. This technique is different from the plane sweep technique where a plane-sweep is used to give a partially persistent data structure. A top-down tour of the tree corresponds to a plane-sweep in the partially persistent data structures.

Related and Previous Work

For the unary dispatching problem the best known bound is $O(N + m)$ space, $O(\text{loglog } N)$ query time and expected $O(N + m)$ preprocessing time [93]. The expectation in the preprocessing time is due to perfect hashing in a van Emde Boas predecessor data structure [120, 121]. Since the tree color data structure is static it is possible to get rid of the expectation using the deterministic dictionary by Hagerup *et al.* [63] together with a simple two-level approach (see e.g. [119]).

For the d-ary dispatching, $d \geq 2$, the result of Ferragina *et al.* [50] is a data structure using space $O(m\ (t\ \text{log}m/\text{log}t)^{d-1})$ and query time $O((\text{log}m/\text{log}t)^{d-1}\text{loglog}N)$, where t is a parameter $2 \leq t \leq m$. For the case $t = 2$ they are able to improve the query time to $O(\log^{d-1} m)$ using fractional cascading [30]. They obtain their results by reducing the d-ary dispatching problem to a point-enclosure problem in d dimensions: Given a point q, check whether there is a smallest rectangle containing q. In the context of the geometric problem, Ferragina *et al.* also present applications to approximate dictionary matching.

Eppstein and Muthukrishnan [47] looked at a similar problem called *packet classification*. Here there is a database of m filters available for preprocessing. A packet filter i in an IP network is a collection of d-dimensional ranges $[l_i^1, r_i^1] \times \cdots \times [l_i^d, r_i^d]$, an action A_i, and a priority p_i. An IP packet P is a d-dimensional vector of values $[P_1, \ldots, P_d]$. A filter i applies to packet P if $P_j \in [l_i^j, r_i^j]$ for $j = 1, \ldots, d$. The packet classification problem is given a packet P to determine the filter of highest priority that applies to P. The ranges of the different filters are typically nested [47]. That is, if two ranges intersect, one is completely contained in the other. In this case the packet classification problem is essentially the same as the multiple dispatching problem. For the case $d = 2$ Eppstein and Muthukrishnan gave an algorithm using space $O(m^{1+o(1)})$ and query time $O(\text{loglog}\, m)$, or $O(m^{1+\varepsilon})$ and query time $O(1)$. They reduced the problem to a geometric problem, very similar to the one in [50]. To solve the problem they used the plane-sweep approach to turn the static two-dimensional rectangle query problem into a partial persistent dynamic one-dimensional problem.

In 2004 Kwok and Poon [99] gave an algorithm for the binary dispatching problem with the same time and space bounds as ours. They reduce the problem

to a point enclosure problem on a 2-dimensional grid the same way as Ferragina *et al.* [50], and then apply the plane sweep technique. Kwok and Poon claim their algorithm is simpler than ours because they use partial persistence instead of full persistence.

2.2 Preliminaries

In this section we give some basic concepts which are used throughout the paper.

Let T be a rooted tree. The set of all nodes in T is denoted $V(T)$. Let $T(v)$ denote the subtree of T rooted at a node $v \in V(T)$. If $w \in V(T(v))$ then v is an ancestor of w, denoted $v \preceq w$, and if $w \in V(T(v)) \backslash \{v\}$ then v is a proper ancestor of w, denoted $v \prec w$. If v is a (proper) ancestor of w then w is a (proper) descendant of v. In the rest of the chapter all trees are rooted trees.

Let C be a set of colors. A labeling $l(v)$ of a node $v \in V(T)$ is a subset of C, i.e., $l(v) \subseteq C$. A labeling $l : V(T) \rightarrow 2^C$ of a tree T is a set of labelings for the nodes in T. Given a labeling of a tree T, the *first ancestor of* $w \in T$ *with color* c is the node $v \in T$ such that $v \preceq w$, $c \in l(v)$, and no node on the path between v and w is labeled c.

2.2.1 Persistent Data Structures

Data structures that one encounters in traditional algorithmic settings are *ephemeral*, i.e., previous states are lost when an update is made. In a *persistent* data structure also previous versions of the data structure can be queried. The concept of persistent data structures was introduced by Driscoll *et al.* [44].

Definition 2.2.1 (Persistence)**.** A data structure is *partially persistent* if all previous versions remain available for queries but only the newest version can be modified. A data structure is *fully persistent* if it allows both queries and updates of previous versions. An update may operate only on a single version at a time, that is, combining two or more versions of the data structure to form a new one is not allowed.

In addition to its ephemeral arguments a persistent update or query takes as an argument the version of the data structure to which the query or update refers. Let the version graph be a directed graph where each node corresponds to a version and there is an edge from node v_1 to a node v_2 if and only if V_2 was created by an update operation to V_1. The version graph for a partially persistent data structure is a path, and for a fully persistent data structure it is a tree.

Definition 2.2.2 (Version tree)**.** The *version tree* represents the temporal evolution of a fully persistent data structure. Each node in the version tree represents the result of an update on a version of the data structure.

Each node v of the version tree is assigned a pair $\langle l, val \rangle$, where l is the location of the persistent data structure that was updated at time v and *val* the value it was updated with. There are two operations in a version tree:

AddVersion(v,l,val): add a new leaf beneath node v and assign it the pair $\langle l, val \rangle$.

LookUp(v,l): find the first ancestor of node v (including v) which has a pair assigned whose first element is l, and return the associated value (or error if no such node exists).

Dietz [41] showed how to make any data structure fully persistent on a unit-cost RAM with logarithmic word size by an efficient implementation of the version tree.

Lemma 2.2.3 (Dietz [41])**.** *A data structure with worst case query time $O(Q(n))$ and update time $O(F(n))$ making worst case $O(U(n))$ memory modifications can be made fully persistent using $O(Q(n) \operatorname{loglog} n)$ worst case time per query and $O(F(n) \operatorname{loglog} n)$ expected amortized time per update using $O(U(n)\ n)$ space.*

It is important that the ephemeral data structure has worst case update time. If the updates times are amortized expensive operations might be repeated in many branches of the version tree.

For more about persistence and results see the surveys by Kaplan [72] and Italiano and Raman [71].

2.2.2 The Tree Color Problem

Definition 2.2.4 (Tree color problem)**.** Let T be a rooted tree with n nodes, where we associate a set of colors with each node of T. The *tree color problem* is to maintain a data structure with the following operations:

color(v, c): add c to v's set of colors, i.e., $l(v) \leftarrow l(v) \cup \{c\}$,

uncolor(v, c): remove c from v's set of colors, i.e., $l(v) \leftarrow l(v) \setminus \{c\}$,

firstcolor(v, c): find the first ancestor of v with color c (this may be v itself).

The *incremental* version of this problem does not support *uncolor*, the *decremental* problem does not support *color*, and the *fully dynamic* problem supports both update operations.

The unary dispatching problem is the same as the tree color problem if we let each color represent a method name.

Alstrup *et al.* [7] showed how to solve the tree color problem on a unit cost RAM with logarithmic word size in expected update time $O(\text{loglog } n)$ for both *color* and *uncolor*, and query time $O(\log n/\text{loglog } n)$, using linear space and preprocessing time. The expected update time is due to hashing. Thus the expectation can be removed at the cost of using more space. We need worst case time when we make the data structure persistent because data structures with amortized/expected time may perform poorly when made fully persistent, since expensive operations might be performed many times.

Querying and updating a version tree of a fully persistent data structure is an incremental version of the tree color problem. Dietz [41] showed how to solve the incremental tree color problem in $O(\text{loglog } n)$ amortized time per operation using linear space, when the nodes are colored top-down and each node has at most one color.

2.2.3 Predecessor Data Structure

In order to get linear space for our data structure we need a tree color data structure using only $O(1)$ worst case memory modifications per update. The bottleneck in the dynamic tree color data structure from [7] is the use of a van Emde Boas predecessor data structure [120, 121] (VEB). A VEB supports insert and predecessor queries on a set of integers from $\{1, \ldots, n\}$ in worst case $O(\text{loglog} n)$ time per operation on a RAM.

In Section 2.4.3 we show how to do modify this data structure such that it only uses worst case $O(1)$ memory modifications per update.

2.3 The Bridge Color Problem

The binary dispatching problem ($d = 2$) can be formulated as the following tree problem, which we call the *bridge color problem*.

Definition 2.3.1 (*Bridge Color Problem*)**.** Let T_1 and T_2 be two rooted trees. Between T_1 and T_2 there are a number of edges—called *bridges*—of different colors. Let C be the set of colors. A bridge is a triple $(c, v_1, v_2) \in C \times V(T_1) \times V(T_2)$ and is

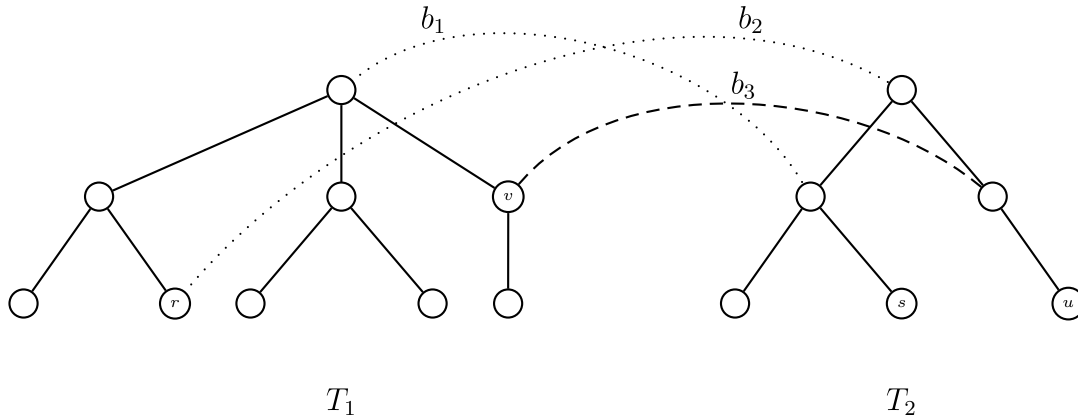


Figure 2.1: An example of the bridge color problem. The solid lines are tree edges and the dashed and dotted lines are bridges of color c and c', respectively. *firstcolorbridge*(c,v,u) returns b_3. *firstcolorbridge*(c',r,s) returns ambiguity since neither b_1 or b_2 is closer than the other.

denoted by $c(v_1, v_2)$. The *bridge color problem* is to construct a data structure which supports the query *firstcolorbridge*(c,v_1,v_2).

$\mathit{firstcolorbridge}(c, v_1, v_2)$ Find a bridge $c(w_1, w_2)$ such that:

1. $w_1 \preceq v_1$ and $w_2 \preceq v_2$.
2. There is no other bridge $c(w'_1, w'_2)$ such that $w_1 \prec w'_1 \preceq v_1$ or $w_2 \prec w'_2 \preceq v_2$.

If there is no bridge satisfying the first condition return NIL. If there is a bridge satisfying the first condition but not the second then return "ambiguity".

See Figure 2.1 for an example of the bridge color problem. The binary dispatching problem can be reduced to the bridge color problem the following way. Let T_1 and T_2 be copies of the tree T in the binary dispatching problem. For every method $s(v_1, v_2) \in \mathcal{M}$ make a bridge of color s between $v_1 \in V(T_1)$ and $v_2 \in V(T_2)$. The following lemma follows immediately from the definitions of the binary dispatching problem and the bridge color problem.

Lemma 2.3.2. *A method $s(B_1, B_2)$ is the most specialized method to an invocation $s(A_1, A_2)$ if and only if $s(B_1, B_2) = \mathit{firstcolorbridge}(c, A_1, A_2)$.*

The problem is now to construct a data structure that supports *firstcolorbridge*. The object of the remaining of this chapter is to show the following theorem:

Theorem 2.3.3. *Using expected $O(m \operatorname{loglog} m)$ time for preprocessing and $O(m)$ space, the query firstcolorbridge can be supported in worst case time $O(\log m)$ per operation, where m is the number of bridges.*

2.4 A Data Structure for the Bridge Color Problem

Let B be a set of bridges ($|B| = m$). As mentioned in the introduction we can assume that the number of nodes in the trees involved in the bridge color problem is $O(m)$, i.e., $|V(T_1)| + |V(T_2)| = O(m)$. In this section we present a data structure that supports *firstcolorbridge* in $O(\log m)$ time per query using $O(m)$ space for the bridge color problem.

2.4.1 Reduction to the Dynamic Tree Color Problem

We first reduce the static bridge color problem to the dynamic tree color problem. For each node $v \in V(T_1)$ we define the labeling l_v of T_2 as follows. The labeling of a node $w \in V(T_2)$ contains color c if w is the endpoint of a bridge of color c with the other endpoint among ancestors of v. Formally, $c \in l_v(w)$ if and only if there exists a node $u \prec v$ such that $c(u, w) \in B$. In addition to each labeling l_v, we need to keep the following extra information stored in a sparse array $H(v)$: For each pair $(w, c) \in V(T_2) \times C$, where $l_v(w)$ contains color c, we keep the first ancestor v' of v from which there is a bridge $c(v', w) \in B$. Note that this set is sparse, i.e., we can use a sparse array to store it. Similar define the symmetric labelings for T_1. See Figure 2.2 for an example.

For each labeling l_v of T_2, where $v \in V(T_1)$, we will construct a data structure for the static tree color problem. The query $\mathit{firstcolorbridge}(c, u, w)$ can then be answered by the following queries in this data structure.

First perform the query *firstcolor*(w, c) in the data structure for the labeling l_u of the tree T_2. If this query reports NIL there is no bridge to report, and we return NIL. Otherwise let w' be the reported node. We make a lookup in $H(u)$ to determine the bridge b such that $b = c(u', w') \in B$. By definition b is the bridge over (u, w') with minimal distance between w and w'. However, it is possible that there is a bridge (u'', w'') over (u, w) where *dist*(u,u″) < *dist*(u,u′). By a symmetric computation with the data structure for the labeling $l(w)$ of T_1 we can detect this. If so we return "ambiguity". Otherwise we return the unique first bridge b. See Figure 2.3 for an example.

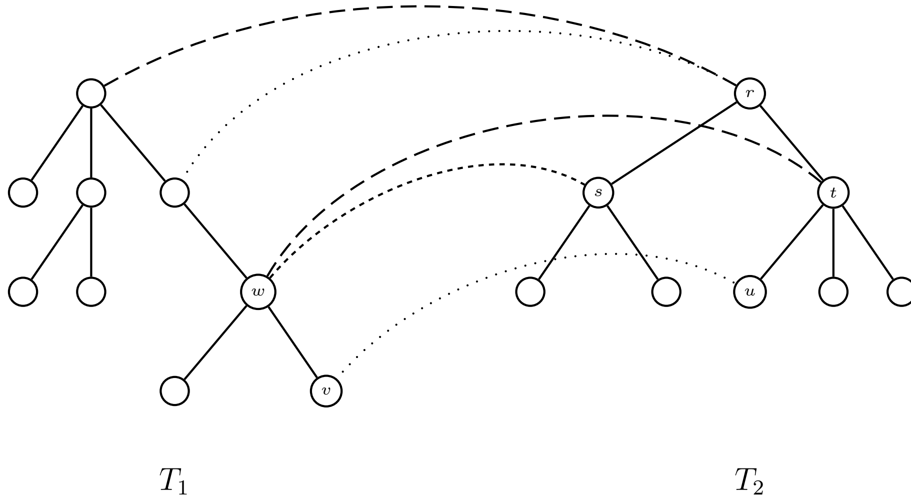


Figure 2.2: Example of labeling. The labeling for $v \in V(T_1)$, l_v: $l_v(r) = \{c_1, c_2\}$, $l_v(s) = \{c_3\}$, $l_v(t) = \{c_1\}$, $l_v(u) = \{c_2\}$. The labeling l_w for $w \in V(T_1)$ is the same as l_v except that $l_w(u)$ is empty.

2.4.2 Using Persistence to Save Space

Explicit representation of the tree color data structures for each of the labelings l_v for all nodes v in T_1 and T_2 would take up space $\Omega(m^2)$. Fortunately, the data structures overlap a lot: Let $v, w \in V(T_1)$, $u \in V(T_2)$, and let $v \preceq w$. Then $l_v(u) \subseteq l_w(u)$. We take advantage of this in a simple way. We make a fully persistent version of the *dynamic* tree color data structure. The idea is that the above set of $O(m)$ tree color data structures corresponds to a persistent version, each created by one of $O(m)$ updates in total.

Formally, suppose we have generated the data structure for the labeling l_v, for v in T_1. Let w be the child of node v in T_1. We construct the data structure for the labeling l_w by updating the persistent structure for l_v by inserting the color corresponding to all bridges with endpoint w (including updating $H(v)$). Since the data structure is fully persistent, we can repeat this for each child of v, and hence obtain data structures for all the labelings for children of v. In other words, we can form all the data structures for the labeling l_v for nodes $v \in V(T_1)$, by updates in the persistent structures according to a top-down traversal of T_1. Another way to see this, is that T_1 is denoting the time (giving the order of the versions). That is, the version tree has the same structure as T_1.

Similarly, we construct the labelings for T_1 by a traversal of T_2. We conclude with the following lemma:

Lemma 2.4.1. *A static data structure for the bridge color problem can be constructed by*

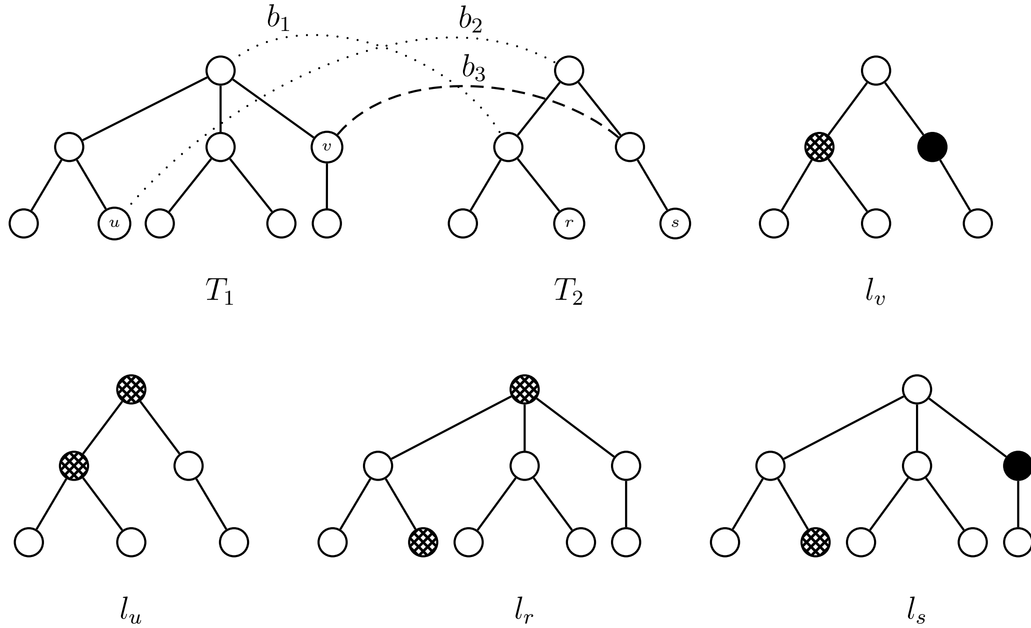


Figure 2.3: The query *firstcolorbridge*(c,v,s) returns b_3. To answer the query we perform the queries *firstcolor*(c, s) in the tree color data structure for l_v and *firstcolor*(c, v) in the tree color data structure for l_s. The query *firstcolorbridge*(c',u,r) returns ambiguity since neither b_1 or b_2 is closer than the other. To answer the query we perform the queries *firstcolor*(c', r) in the tree color data structure for l_u and *firstcolor*(c, u) in the tree color data structure for l_r.

$O(m)$ updates to a fully persistent version of the dynamic tree color problem.

2.4.3 Reducing the number of Memory Modifications in the Tree Color Problem

Alstrup *et al.* [7] gives the following upper bounds for the tree color problem for a tree of size m. Expected $O(\text{loglog}\, m)$ update time for both *color* and *uncolor*, and query time $O(\log m/\text{loglog}\, m)$, with linear space and preprocessing time.

For our purposes we need a slightly stronger result, i.e., updates that only make worst case $O(1)$ memory modifications. By inspection of the dynamic tree color algorithm, the bottle-neck in order to achieve this, is the use of the van Emde Boas predecessor data structure [120, 121] (VEB). Using a standard technique by Dietz and Raman [42] to implement a fast predecessor structure we get the following result.

Theorem 2.4.2. *Insert and predecessor queries on a set of integers from $\{1, \dots, n\}$ can*

be performed in $O(\text{loglog}\, n)$ worst case time per operation using worst case $O(1)$ memory modifications per update.

To prove the theorem we first show an amortized result[1]. The elements in our predecessor data structure is grouped into buckets $S_1, \ldots, S_k$, where we maintain the following invariants:

$$\begin{array}{lll} (1) & \max S_i < \min S_{i+1} & \text{for } i = 1, \ldots k-1, \text{ and} \\ (2) & 1/2 \log n < |S_i| \leq 2 \log n & \text{for all } i. \end{array}$$

We have $k \in O(n/\log n)$. Each S_i is represented by a balanced search tree with $O(1)$ worst case update time once the position of the inserted or deleted element is known and query time $O(\log m)$, where m is the number of nodes in the tree [51, 86]. This gives us update time $O(\text{loglog}\, n)$ in a bucket, but only $O(1)$ memory modifications per update. The minimum element s_i of each bucket S_i is stored in a VEB.

When a new element x is inserted it is placed in the bucket S_i such that $s_i < x < s_{i+1}$, or in S_1 if no such bucket exists. Finding the correct bucket is done by a predecessor query in the VEB. This takes $O(\text{loglog}\, n)$ time. Inserting the element in the bucket also takes $O(\text{loglog}\, n)$ time, but only $O(1)$ memory modifications. When a bucket S_i becomes to large it is split into two buckets of half size. This causes a new element to be inserted into the VEB and the binary trees for the two new buckets have to be build. An insertion into the VEB takes $O(\text{loglog}\, n)$ time and uses the same number of memory modifications. Building the binary search trees uses $O(\log n)$ time and the same number of memory modifications. When a bucket is split there must have been at least $\log n$ insertions into this bucket since it last was involved in a split. That is, splitting and inserting uses $O(1)$ amortized memory modifications per insertion.

Lemma 2.4.3. *Insert and predecessor queries on a set of integers from $\{1, \ldots, n\}$ can be performed in $O(\text{loglog}\, n)$ worst case time for predecessor and $O(\text{loglog}\, n)$ amortized time for insert using $O(1)$ amortized number of memory modifications per update.*

We can remove the amortization by the following technique by Raman [100] called thinning at the cost of making the bucket sizes $\Theta(\log^2 n)$.

Let $\alpha > 0$ be a sufficiently small constant. Define the *criticality* of a bucket to be:

$$\rho(b) = \frac{1}{\alpha \log n} \max\{0, \text{size}(b) - 1.8 \log^2 n\}.$$

[1]The amortized result (Lemma 2.4.3) was already shown by Mehlhorn and Näher [92], but in order to make the deamortization we give another implementation here.

A bucket b is called *critical* if $\rho(b) > 0$. We want to ensure that size$(b) \leq 2\log^2 n$. To maintain the size of the buckets every $\alpha \log n$ updates take the most critical bucket (if there is any) and move $\log n$ elements to a newly created empty adjacent bucket. A bucket rebalancing uses $O(\log n)$ memory modifications and we can thus perform it with $O(1)$ memory modifications per update spread over no more than $\alpha \log n$ updates.

We now show that the buckets never get too big. The criticality of all buckets can only increase by 1 between bucket rebalancings. We see that the criticality of the bucket being rebalanced is decreased, and no other bucket has its criticality increased by the rebalancing operations. We make use of the following lemma due to Raman:

Lemma 2.4.4 (Raman [100]). *Let $x_1, \ldots, x_n$ be real-valued variables, all initially zero. Repeatedly do the following:*

(1) Choose n non-negative real numbers $a_1, \ldots, a_n$ such that $\sum_{i=1}^{n} a_i = 1$, and set $x_i \leftarrow x_i + a_i$ for $1 \leq i \leq n$.

(2) Choose an x_i such that $x_i = \max_j\{x_j\}$, and set $x_i \leftarrow \max\{x_i - c, 0\}$ for some constant $c \geq 1$.

Then each x_i will always be less than $\ln n + 1$, even when $c = 1$.

Apply the lemma as follows: Let the variables of Lemma 2.4.4 be the criticalities of the buckets. The reals a_i are the increases in the criticalities between rebalancings and $c = 1/\alpha$. We see that if $\alpha \leq 1$ the criticality of a bucket will never exceed $\ln n + 1 = O(\log n)$. Thus for sufficiently small α the size of the buckets will never exceed $2\log^2 n$. This completes the proof of Theorem 2.4.2.

We need worst case update time for *color* in the tree color problem in order to make it persistent. The expected update time is due to hashing, and can be removed at the cost of using more space. We now use Theorem 2.4.2 to get the following lemma.

Lemma 2.4.5. *Using linear time for preprocessing, we can maintain a tree with complexity $O(\log\log m)$ for color and complexity $O(\log m/\log\log m)$ for firstcolor, using $O(1)$ memory modifications per update, where m is the number of nodes in the tree.*

2.4.4 Making the Data Structure Persistent

Using Dietz' method [41] to make a data structure fully persistent on the data structure from Lemma 2.4.5, we can construct a fully persistent version of the tree color data structure.

Taking advantage of the fact that the structure of the version tree is known from the beginning, we can get faster preprocessing time for our bridge color data structure.

Since the structure of the version tree is known from the beginning we can construct the tree color data structure for the version tree, by first to running through the whole version tree in an Euler tour, and remembering which updates are made in which node. Then we can construct the tree color data structure for the version tree using the data structure for the static tree color problem by Muthukrishnan and Müller [93]. This uses expected linear preprocessing time and linear space using worst-case $O(\text{loglog}\, m)$ time per query. As mentioned earlier the expectation in the preprocessing time can be removed.

Another possibility is to use a modified version of Dietz' method to make a data structure fully persistent. Recall that this method gives an expected amortized slowdown of $O(\text{loglog}\, m)$. The amortization comes from the problem of maintaining order in a list. Since we know the structure of the version tree from the beginning we can get rid of this amortization. This gives a significant simplification of the construction of our bridge color data structure. This data structure uses expected $O(\text{loglog}\, m)$ time per update, and the preprocessing time for our bridge color problem is thus a factor of $O(\text{loglog}\, m)$ bigger than in the approach using a static tree color data structure for the version tree.

2.4.5 Reducing the Space

Using the method described in the previous section we can construct a fully persistent version of the tree color data structure with complexity $O(\text{loglog}\, m)$ for *color* and *uncolor*, and complexity $O((\log m/\text{loglog}\, m) \cdot \text{loglog}\, m) = O(\log m)$ for *firstcolor*, using $O(m)$ memory modifications, where m is the number of nodes in the tree.

According to Lemma 2.4.1 a data structure for the bridge color problem can be constructed by $O(m)$ updates to a fully persistent version of the dynamic tree color problem. We can thus construct a data structure for the bridge color problem in time $O(m\, \text{loglog}\, m)$, which has query time $O(\log m)$, where m is the number of bridges. However, this data structure uses $O(c{\cdot}m)$ space, where c is the number of method names. Since we only use $O(m)$ memory modifications to construct the data structure we can use dynamic perfect hashing [43] to reduce the space. This gives a data structure for the bridge color problem using $O(m)$ space and expected preprocessing time $O(m\, \text{loglog}\, m)$. This completes the proof of Theorem 2.3.3.

If we use $O(N)$ time to reduce the class hierarchy tree to size $O(m)$ as men-

tioned in the introduction, we get the following corollary to Theorem 2.3.3.

Corollary 2.4.6. *Using expected* $O(N + m \operatorname{loglog} m)$ *time for preprocessing and* $O(m)$ *space, the* binary dispatching problem *can be solved in worst case time* $O(\log m)$ *per query. Here* N *is the number of classes and* m *is the number of methods.*

Chapter 3

Tree Inclusion

Given two rooted, ordered, and labeled trees P and T the tree inclusion problem is to determine if P can be obtained from T by deleting nodes in T. This problem has recently been recognized as an important query primitive in XML databases. Kilpeläinen and Mannila [*SIAM J. Comput. 1995*] presented the first polynomial time algorithm using quadratic time and space. Since then several improved results have been obtained for special cases when P and T have a small number of leaves or small depth. However, in the worst case these algorithms still use quadratic time and space. In this paper we present a new approach to the problem which leads to a new algorithm which uses optimal linear space and has subquadratic running time. Our algorithm improves all previous time and space bounds. Most importantly, the space is improved by a linear factor. This will make it possible to query larger XML databases and speed up the query time since more of the computation can be kept in main memory.

3.1 Introduction

Let T be a rooted tree. We say that T is *labeled* if each node is a assigned a symbol from an alphabet Σ and we say that T is *ordered* if a left-to-right order among siblings in T is given. All trees in this paper are rooted, ordered, and labeled. A tree P is *included* in T, denoted $P \sqsubseteq T$, if P can be obtained from T by deleting nodes of T. Deleting a node v in T means making the children of v children of the parent of v and then removing v. The children are inserted in the place of v in the left-to-right order among the siblings of v. The *tree inclusion problem* is to determine if P can be included in T and if so report all subtrees of T that include P. The tree P and T is often called the *pattern* and *target*, respectively.

Recently, the problem has been recognized as an important query primitive

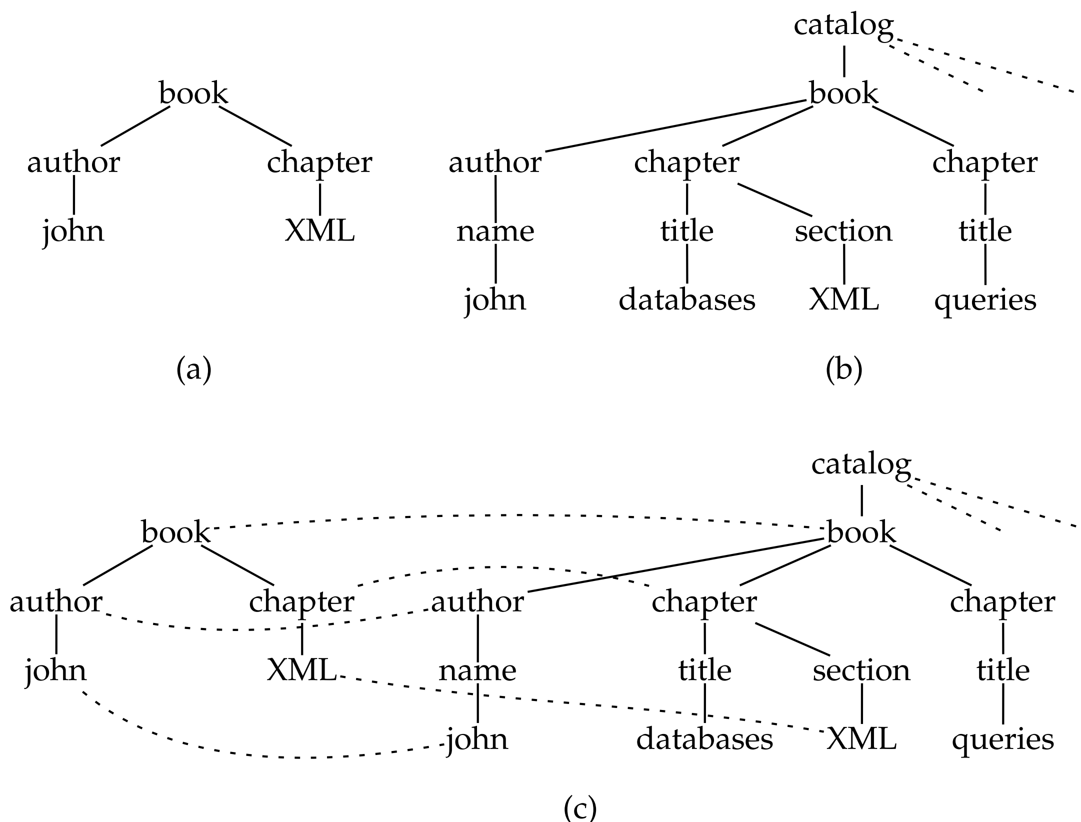


Figure 3.1: Can the tree (a) be included in the tree (b)? It can and the embedding is given in (c).

for XML data and has received considerable attention, see *e.g.*, [105, 125, 124, 127, 106, 118]. The key idea is that an XML document can be viewed as an ordered, labeled tree and queries on this tree correspond to a tree inclusion problem. As an example consider Figure 3.1. Suppose that we want to maintain a catalog of books for a bookstore. A fragment of the tree, denoted D, corresponding to the catalog is shown in (b). In addition to supporting full-text queries, such as find all documents containing the word "John", we can also utilize the tree structure of the catalog to ask more specific queries, such as "find all books written by John with a chapter that has something to do with XML". We can model this query by constructing the tree, denoted Q, shown in (a) and solve the tree inclusion problem: is $Q \sqsubseteq D$? The answer is yes and a possible way to include Q in D is indicated by the dashed lines in (c). If we delete all the nodes in D not touched by dashed lines the trees Q and D become isomorphic. Such a mapping of the nodes from Q to D given by the dashed lines is called an *embedding* (formally defined in

Section 3.3).

The tree inclusion problem was initially introduced by Knuth [83, exercise 2.3.2-22] who gave a sufficient condition for testing inclusion. Motivated by applications in structured databases [79, 90] Kilpeläinen and Mannila [80] presented the first polynomial time algorithm using $O(n_P n_T)$ time and space, where n_P and n_T is the number of nodes in a tree P and T, respectively. During the last decade several improvements of the original algorithm of Kilpeläinen and Mannila [80] have been suggested [78, 2, 103, 31]. The previously best known bound is due to Chen [31] who presented an algorithm using $O(l_P n_T)$ time and $O(l_P \min\{d_T, l_T\})$ space. Here, l_S and d_S denotes the number of leaves of and the maximum depth of a tree S, respectively. This algorithm is based on an algorithm of Kilpeläinen [78]. Note that the time and space is still $\Theta(n_P n_T)$ for worst-case input trees.

In this paper we improve all of the previously known time and space bounds. Combining the three algorithms presented in this paper we have:

Theorem 3.1.1. *For trees P and T the ordered tree inclusion problem can be solved in time $O(\min(\frac{n_P n_T}{\log n_T}, l_P n_T, n_P l_T \log\log n_T))$ using optimal $O(n_T + n_P)$ space.*

Hence, for worst-case input this improves the previous time and space bounds by a logarithmic and linear factor, respectively. When P has a small number of leaves the running time of our algorithm matches the previously best known time bound of [31] while maintaining linear space. In the context of XML databases the most important feature of our algorithms is the space usage. This will make it possible to query larger trees and speed up the query time since more of the computation can be kept in main memory.

3.1.1 Techniques

Most of the previous algorithms, including the best one by Chen [31], are essentially based on a simple dynamic programming approach from the original algorithm of Kilpeläinen and Mannila [80]. The main idea behind this algorithm is the following: Let $v \in V(P)$ and $w \in V(T)$ be nodes with children $v_1, \dots, v_i$ and $w_1, \dots, w_j$, respectively. To decide if $P(v)$ can be included $T(w)$ we try to find a sequence of numbers $1 \le x_1 < x_2 < \cdots < x_i \le j$ such that $P(v_k)$ can be included in $T(w_{x_k})$ for all k, $1 \le k \le i$. If we have already determined whether or not $P(v_s) \sqsubseteq T(w_t)$, for all s and t, $1 \le s \le i$, $1 \le t \le j$, we can efficiently find such a sequence by scanning the children of v from left to right. Hence, applying this approach in a bottom-up fashion we can determine, if $P(v) \sqsubseteq T(w)$, for all pairs $(v, w) \in V(P) \times V(T)$.

In this paper we take a different approach. The main idea is to construct a data structure on T supporting a small number of procedures, called the *set procedures*, on subsets of nodes of T. We show that any such data structure implies an algorithm for the tree inclusion problem. We consider various implementations of this data structure which all use linear space. The first simple implementation gives an algorithm with $O(l_P n_T)$ running time. As it turns out, the running time depends on a well-studied problem known as the *tree color problem*. We show a direct connection between a data structure for the tree color problem and the tree inclusion problem. Plugging in a data structure of Dietz [41] we obtain an algorithm with $O(n_P l_T \log\log n_T)$ running time.

Based on the simple algorithms above we show how to improve the worst-case running time of the set procedures by a logarithmic factor. The general idea used to achieve this is to divide T into small trees or forests, called *micro trees* or *clusters* of logarithmic size which overlap with other micro trees in at most 2 nodes. Each micro tree is represented by a constant number of nodes in a *macro tree*. The nodes in the macro tree are then connected according to the overlap of the micro trees they represent. We show how to efficiently preprocess the micro trees and the macro tree such that the set procedures use constant time for each micro tree. Hence, the worst-case running time is improved by a logarithmic factor to $O(\frac{n_P n_T}{\log n_T})$.

Throughout the paper we assume a standard RAM model of computation with word size $\Omega(\log n)$. We use a standard instruction set such as bitwise boolean operations, shifts, and addition.

3.1.2 Related Work

For some applications considering *unordered* trees is more natural. However, in [91, 80] this problem was proved to be NP-complete. The tree inclusion problem is closely related to the *tree pattern matching problem* [69, 84, 45, 37]. The goal is here to find an injective mapping f from the nodes of P to the nodes of T such that for every node v in P the ith child of v is mapped to the ith child of $f(v)$. The tree pattern matching problem can be solved in $O(n \log^{O(1)} n)$ time, where $n = n_P + n_T$. Another similar problem is the *subtree isomorphism* problem [33, 108], which is to determine if T has a subgraph which is isomorphic to P. The subtree isomorphism problem can be solved efficiently for ordered and unordered trees. The best algorithms for this problem use $O(n_P^{1.5} n_T / \log n_P)$ for unordered trees and $O(n_P n_T / \log n_P)$ time ordered trees [33, 108]. Both use $O(n_P n_T)$ space. The tree inclusion problem can be considered a special case of the *tree edit distance*

problem [111, 128, 81]. Here one wants to find the minimum sequence of insert, delete, and relabel operations needed to transform P into T. The currently best worst-case algorithm for this problem uses $O(n_P^2 n_T \log n_T)$ time. For more details and references see the survey [21].

3.1.3 Outline

In Section 3.2 we give notation and definitions used throughout the paper. In Section 3.3 a common framework for our tree inclusion algorithms is given. Section 3.4 present two simple algorithms and then, based on these result, we show how to get a faster algorithm in Section 3.5.

3.2 Notation and Definitions

In this section we define the notation and definitions we will use throughout the paper. For a graph G we denote the set of nodes and edges by $V(G)$ and $E(G)$, respectively. Let T be a rooted tree. The root of T is denoted by $\mathrm{root}(T)$. The *size* of T, denoted by n_T, is $|V(T)|$. The *depth* of a node $v \in V(T)$, $\mathrm{depth}(v)$, is the number of edges on the path from v to $\mathrm{root}(T)$ and the depth of T, denoted d_T, is the maximum depth of any node in T. The set of children of a node v is denoted $\mathrm{child}(v)$. A node with no children is a leaf and otherwise an internal node. The set of leaves of T is denoted $L(T)$ and we define $l_T = |L(T)|$. We say that T is *labeled* if each node v is a assigned a symbol, denoted $l(v)$, from an alphabet Σ and we say that T is *ordered* if a left-to-right order among siblings in T is given. All trees in this paper are rooted, ordered, and labeled.

Let $T(v)$ denote the subtree of T rooted at a node $v \in V(T)$. If $w \in V(T(v))$ then v is an ancestor of w, denoted $v \preceq w$, and if $w \in V(T(v))\backslash\{v\}$ then v is a proper ancestor of w, denoted $v \prec w$. If v is a (proper) ancestor of w then w is a (proper) descendant of v. A node z is a common ancestor of v and w if it is an ancestor of both v and w. The nearest common ancestor of v and w, $\mathrm{nca}(v, w)$, is the common ancestor of v and w of largest depth. The *first ancestor of w labeled α*, denoted $\mathrm{fl}(w, \alpha)$, is the node v such that $v \preceq w$, $l(v) = \alpha$, and no node on the path between v and w is labeled α. If no such node exists then $\mathrm{fl}(w, \alpha) = \perp$, where $\perp \notin V(T)$ is a special *null node*.

For any set of pairs U, let $U|_1$ and $U|_2$ denote the *projection* of U to the first and second coordinate, that is, if $(u_1, u_2) \in U$ then $u_1 \in U|_1$ and $u_2 \in U|_2$.

Lists A *list*, X, is a finite sequence of objects $X = [v_1, \dots, v_k]$. The *length* of the list, denoted $|X|$, is the number of objects in X. The ith element of X, $X[i]$, $1 \leq i \leq |X|$ is the object v_i and $v \in X$ iff $v = X[j]$ for some $1 \leq j \leq |X|$. For any two lists $X = [v_1, \dots, v_k]$ and $Y = [w_1, \dots, w_k]$, the list obtained by *appending* Y to X is the list $X \circ Y = [v_1, \dots, v_k, w_1, \dots, w_k]$. We extend this notation such that for any object u, $X \circ u$ denotes the list $X \circ [u]$. For simplicity in the notation we will sometimes write $[v_i \mid 1 \leq i \leq k]$ to denote the list $[v_1, \dots, v_k]$. A *pair list* is a list of pairs of object $Y = [(v_1, w_1), \dots, (v_k, w_k)]$. Here the first and second element in the pair is denoted by $Y[i]_1 = v_i$ and $Y[i]_2 = w_i$. The projection of pair lists is defined by $Y|_1 = [v_1, \dots, v_k]$ and $Y|_2 = [w_1, \dots, w_k]$.

Orderings Let T be a tree with root v and let $v_1, \dots, v_k$ be the children of v from left-to-right. The *preorder traversal* of T is obtained by visiting v and then recursively visiting $T(v_i)$, $1 \leq i \leq k$, in order. Similarly, the *postorder traversal* is obtained by first visiting $T(v_i)$, $1 \leq i \leq k$, in order and then v. The *preorder number* and *postorder number* of a node $w \in T(v)$, denoted by $\mathrm{pre}(w)$ and $\mathrm{post}(w)$, is the number of nodes preceding w in the preorder and postorder traversal of T, respectively. The nodes to the *left* of w in T is the set of nodes $u \in V(T)$ such that $\mathrm{pre}(u) < \mathrm{pre}(w)$ and $\mathrm{post}(u) < \mathrm{post}(w)$. If u is to the left of w, denoted by $u \lhd w$, then w is to the *right* of u. If $u \lhd w$, $u \preceq w$, or $w \prec u$ we write $u \unlhd w$. The null node $\perp$ is not in the ordering, i.e., $\perp \ntriangleleft v$ for all nodes v.

Deep Sets A set of nodes $V \subseteq V(T)$ is *deep* iff no node in V is a proper ancestor of another node in V.

Minimum Ordered Pair Let $V_1, \dots, V_k$ be deep sets of nodes. Let $\Phi(V_1, \dots, V_k) \subseteq (V_1 \times \cdots \times V_k)$, be the set such that $(v_1, \dots, v_k) \in \Phi(V_1, \dots, V_k)$ iff $v_1 \lhd \cdots \lhd v_k$. If $(v_1, \dots, v_k) \in \Phi(V_1, \dots, V_k)$ and there is no $(v'_1, \dots, v'_k) \in \Phi(V_1, \dots, V_k)$, where either $v_1 \lhd v'_1 \lhd v'_k \unlhd v_k$ or $v_1 \unlhd v'_1 \lhd v'_k \lhd v_k$ then the pair (v_1, v_k) is a *minimum ordered pair*. The set of minimum ordered pairs for $V_1, \dots, V_k$ is denoted by $\mathrm{mop}(V_1, \dots, V_k)$. Figure 3.2 illustrates mop on a small example. The following lemma shows that $\mathrm{mop}(V_1, \dots, V_k)$ can be computed iteratively by first computing $\mathrm{mop}(V_1, V_2)$ and then $\mathrm{mop}(\mathrm{mop}(V_1, V_2)|_2, V_3)$ and so on.

Lemma 3.2.1. *For any deep sets of nodes $V_1, \dots, V_k$ we have, $(v_1, v_k) \in \mathrm{mop}(V_1, \dots, V_k)$ if and only if there exists a node v_{k-1} such that $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \dots, V_{k-1})$ and $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \dots, V_{k-1})|_2, V_k)$.*

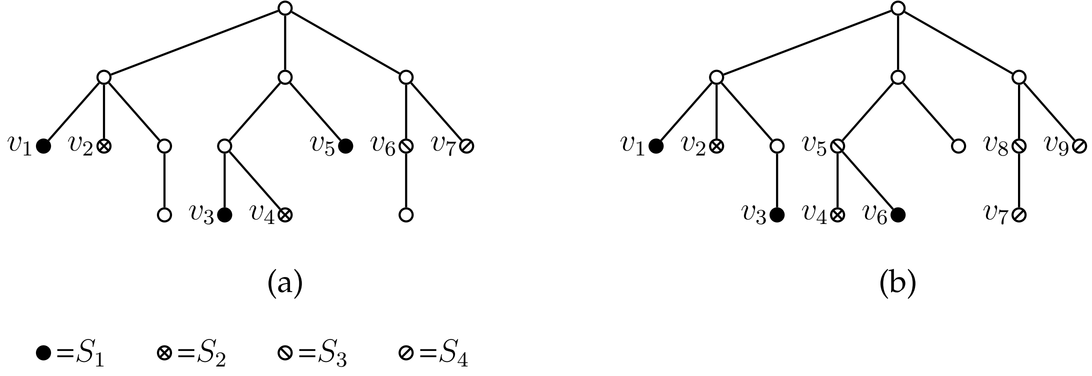


Figure 3.2: Two examples of mop. In (a) we have $\Phi(S_1, S_2, S_1, S_3, S_4) = \{(v_1, v_2, v_3, v_6, v_7), (v_1, v_2, v_5, v_6, v_7), (v_1, v_4, v_5, v_6, v_7), (v_3, v_4, v_5, v_6, v_7)\}$ and thus $\mathrm{mop}(S_1, S_2, S_1, S_3, S_4) = \{(v_3, v_7)\}$. In (b) we have $\mathrm{mop}(S_1, S_2, S_1, S_3, S_4) = \{(v_1, v_7), (v_3, v_9)\}$ since $\Phi(S_1, S_2, S_1, S_3, S_4) = \{(v_1, v_2, v_3, v_5, v_7), (v_1, v_2, v_6, v_8, v_9), (v_1, v_2, v_3, v_8, v_9), (v_1, v_2, v_3, v_5, v_9), (v_1, v_4, v_6, v_8, v_9), (v_3, v_4, v_6, v_8, v_9)\}$.

Proof. We will start by showing $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k) \Rightarrow \exists v_{k-1}$ such that $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ and $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$.

First note that $(w_1, \ldots, w_k) \in \Phi(V_1, \ldots, V_k) \Rightarrow (w_1, \ldots, w_{k-1}) \in \Phi(V_1, \ldots, V_{k-1})$. Since $(v, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$ there must be a minimum node v_{k-1} such that $\Phi(V_1, \ldots, V_{k-1})$ contains the tuple $(v_1, \ldots, v_{k-1})$. It follows immediately that the pair $(v, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$. It remains to show that the pair $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$. Since $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$ there exists no $w \in V_k$ such that $v_{k-1} \lhd w \lhd v_k$. For the sake of contradiction assume there exists a $w \in \mathrm{mop}(V_1, \ldots, V_{k-1})|_2$ such that $v_{k-1} \lhd w \lhd v_k$. Since $(v, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ this implies that there is a $w' \rhd v_1$ such that $(w', w) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$. But this implies that there is a tuple $(w', \ldots, w, v_k) \in \Phi(V_1, \ldots, V_k)$ contradicting that $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$.

We now show that if there exists a v_{k-1} such that $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ and $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$ then $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$.

Clearly, there exists a tuple $(v_1, \ldots, v_{k-1}, v_k) \in \Phi(V_1, \ldots, V_k)$. Assume that there exists a tuple $(w_1, \ldots, w_k) \in \Phi(V_1, \ldots, V_k)$ such that $v_1 \lhd w_1 \lhd w_k \unlhd v_k$. Since $w_{k-1} \unlhd v_{k-1}$ this contradicts that $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$. Assume that there exists a tuple $(w_1, \ldots, w_k) \in \Phi(V_1, \ldots, V_k)$ such that $v_1 \unlhd w_1 \lhd w_k \lhd v_k$. Since $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ we have $v_{k-1} \unlhd w_{k-1}$ and thus $w_k \rhd v_{k-1}$ contradicting $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$. $\square$

3.3 Computing Deep Embeddings

In this section we will present a general framework for answering tree inclusion queries. As Kilpeläinen and Mannila [80] we solve the equivalent *tree embedding problem*. Let P and T be rooted labeled trees. An *embedding* of P in T is an injective function $f : V(P) \to V(T)$ such that for all nodes $v, u \in V(P)$,

(i) $l(v) = l(f(v))$. (label preservation condition)

(ii) $v \prec u$ iff $f(v) \prec f(u)$. (ancestor condition)

(iii) $v \lhd u$ iff $f(v) \lhd f(u)$. (order condition)

An example of an embedding is given in Figure 3.1(c).

Lemma 3.3.1 (Kilpeläinen and Mannila[80]). *For any trees P and T, $P \sqsubseteq T$ iff there exists an embedding of P in T.*

We say that the embedding f is *deep* if there is no embedding g such that $f(\text{root}(P)) \prec g(\text{root}(P))$. The *deep occurrences* of P in T, denoted $\text{emb}(P, T)$ is the set of nodes,

$$\text{emb}(P, T) = \{f(\text{root}(P)) \mid f \text{ is a deep embedding of } P \text{ in } T\}.$$

Note that $\text{emb}(P, T)$ must be a deep set in T. Furthermore, by definition the set of ancestors of nodes in $\text{emb}(P, T)$ is the set of subtrees $T(u)$ such that $P \sqsubseteq T(u)$. Hence, to solve the tree inclusion problem it is sufficient to compute $\text{emb}(P, T)$ and then, using additional $O(n_T)$ time, report all ancestors (if any) of this set.

We show how to compute deep embeddings. The key idea is to construct a data structure that allows a fast implementation of the following procedures. For all $V \subseteq V(T)$, $U \subseteq V(T) \times V(T)$, and $\alpha \in \Sigma$ define:

$\textsc{Parent}_T(V)$. Return the set $R := \{\text{parent}(v) \mid v \in V\}$.

$\textsc{Nca}_T(U)$. Return the set $R := \{\text{nca}(u_1, u_2) \mid (u_1, u_2) \in U\}$.

$\textsc{Deep}_T(V)$. Return the set $R := \{v \in V \mid \nexists w \in V \text{ such that } v \prec w\}$.

$\textsc{Mop}_T(U, V)$. Return the set of pairs R such that for any pair $(u_1, u_2) \in U$, $(u_1, v) \in R$ iff $(u_2, v) \in \text{mop}(U|_2, V)$.

$\textsc{Fl}_T(V, \alpha)$. Return the set $R := \{\text{fl}(v, \alpha) \mid v \in V\}$.

Collectively we call these procedures the *set procedures*. With the set procedures we can compute deep embeddings. The following procedure $\textsc{Emb}_T(v)$, $v \in V(P)$ recursively computes the set of deep occurrences of $P(v)$ in T.

$\text{EMB}_T(v)$ Let $v_1, \ldots, v_k$ be the sequence of children of v ordered from left to right. There are three cases:

1. $k = 0$ (v is a leaf). Set $R := \text{DEEP}_T(\text{FL}_T(L(T), l(v)))$.
2. $k = 1$. Recursively compute $R_1 := \text{EMB}_T(v_1)$.
   Set $R := \text{DEEP}_T(\text{FL}_T(\text{DEEP}_T(\text{PARENT}_T(R_1)), l(v)))$.
3. $k > 1$. Compute $R_1 := \text{EMB}_T(v_1)$ and $U_1 := \{(r, r) \mid r \in R_1\}$. For i, $1 \leq i \leq k$, compute $R_i := \text{EMB}_T(v_i)$ and $U_i := \text{MOP}_T(U_{i-1}, R_i)$. Finally, compute $R := \text{DEEP}_T(\text{FL}_T(\text{DEEP}_T(\text{NCA}_T(U_k)), l(v)))$.

If $R = \emptyset$ stop and report that there is no deep embedding of $P(v)$ in T. Otherwise return R.

Figure 3.3 illustrates how EMB works on a small example.

Lemma 3.3.2. *For any two trees T and P, $\text{EMB}_T(v)$ computes the set of deep occurrences of $P(v)$ in T.*

Proof. By induction on the size of the subtree $P(v)$. If v is a leaf we immediately have that $\text{emb}(v, T) = \text{DEEP}_T(\text{FL}_T(L(T), l(v)))$ and thus case 1 follows. Suppose that v is an internal node with $k \geq 1$ children $v_1, \ldots, v_k$. We show that $\text{emb}(P(v), T) = \text{EMB}_T(v)$. Consider cases 2 and 3 of the algorithm.

If $k = 1$ we have that $w \in \text{EMB}_T(v)$ implies that $l(w) = l(v)$ and there is a node $w_1 \in \text{EMB}_T(v_1)$ such that $\text{fl}(\text{parent}(w_1), l(v)) = w$, that is, no node on the path between w_1 and w is labeled $l(v)$. By induction $\text{EMB}_T(v_1) = \text{emb}(P(v_1), T)$ and therefore w is the root of an embedding of $P(v)$ in T. Since $\text{EMB}_T(v)$ is the deep set of all such nodes it follows that $w \in \text{emb}(P(v), T)$. Conversely, if $w \in \text{emb}(P(v), T)$ then $l(w) = l(v)$, there is a node $w_1 \in \text{emb}(P(v_1), T)$ such that $w \prec w_1$, and no node on the path between w and w_1 is labeled $l(v)$, that is, $\text{fl}(w_1, l(v)) = w$. Hence, $w \in \text{EMB}_T(v)$.

Before considering case 3 we show that $U_j = \text{mop}(\text{EMB}_T(v_1), \ldots, \text{EMB}_T(v_j))$ by induction on j, $2 \leq j \leq k$. For $j = 2$ it follows from the definition of MOP_T that $U_2 = \text{mop}(\text{EMB}_T(v_1), \text{EMB}_T(v_2))$. Hence, assume that $j > 2$. By the induction hypothesis we have $U_j = \text{MOP}_T(U_{j-1}, \text{EMB}_T(v_j)) = \text{MOP}_T(\text{mop}(\text{EMB}_T(v_1), \ldots, \text{EMB}_T(v_{j-1})), R_j)$. By the definition of MOP_T, U_j is the set of pairs such that for any pair $(r_1, r_{j-1}) \in \text{mop}(\text{EMB}_T(v_1), \ldots, \text{EMB}_T(v_{j-1}))$, we have that $(r_1, r_j) \in U_j$ if and only if $(r_{j-1}, r_j) \in \text{mop}(\text{mop}(\text{EMB}_T(v_1), \ldots, \text{EMB}_T(v_{j-1}))|_2, R_j)$. By Lemma 3.2.1 it follows that $(r_1, r_j) \in U_j$ if and only if $(r_1, r_j) \in \text{mop}(\text{EMB}_T(v_1), \ldots, \text{EMB}_T(v_j))$.

Next consider case 3 ($k > 1$). If $w \in \text{EMB}_T(v)$ we have $l(w) = l(v)$ and there are nodes $(w_1, w_k) \in \text{mop}(\text{emb}(P(v_1), T), \ldots, \text{emb}(P(v_k), T))$ such that $w =$

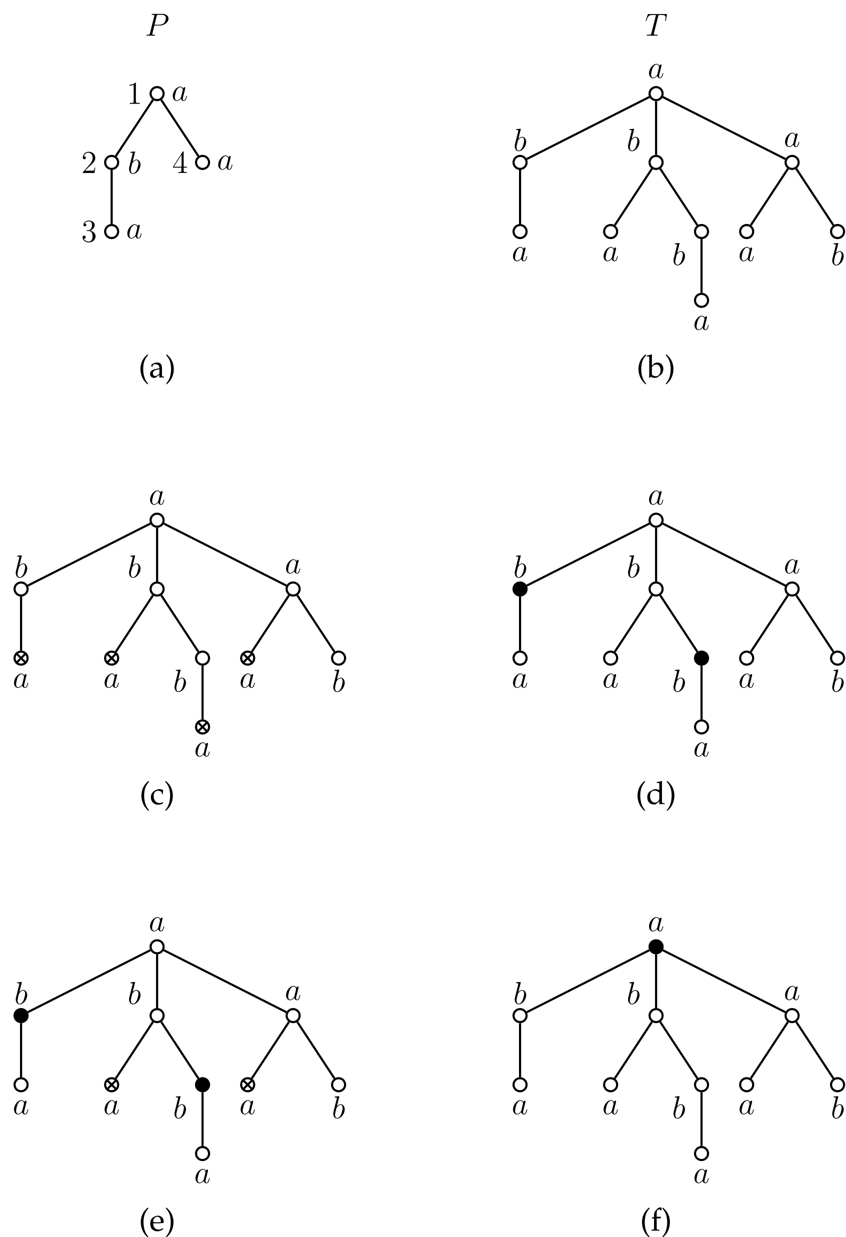


Figure 3.3: Computing the deep occurrences of P into T depicted in (a) and (b) respectively. The nodes in P are numbered 1–4 for easy reference. (c) Case 1 of EMB: The set $\text{EMB}_T(3)$. Since 3 and 4 are leaves and $l(3) = l(4)$ we have $\text{EMB}_T(3) = \text{EMB}_T(4)$. (d) Case 2 of EMB. The set $\text{EMB}_T(2)$. Note that the middle child of the root of T is not in the set since it is not a deep occurrence. (e) Case 3 of EMB: The two minimal ordered pairs of (d) and (c). (f) The nearest common ancestors of both pairs in (e) give the root node of T which is the only (deep) occurrence of P.

$\mathrm{fl}(\mathrm{nca}(w_1, w_k), l(v))$. Clearly, w is the root of an embedding of $P(v)$ in T. Assume for contradiction that w is not a deep embedding, that is, $w \prec u$ for some node $u \in \mathrm{emb}(P(v), T)$. Since $w = \mathrm{fl}(\mathrm{nca}(w_1, w_k), l(v))$ there must exist nodes $u_1 \lhd \cdots \lhd u_k$, such that $u_i \in \mathrm{emb}(P(v_i), T)$ and $u = \mathrm{fl}(\mathrm{nca}(u_1, u_k), l(v))$. Since $w \prec u$ we must have $w_1 \lhd u_1$ or $u_k \lhd w_k$. However, this contradicts the fact that $(w_1, w_k) \in \mathrm{mop}(\mathrm{emb}(P(v_1), T), \ldots, \mathrm{emb}(P(v_k), T))$. If $w \in \mathrm{emb}(P(v), T)$ a similar argument implies that $w \in \textsc{Emb}_T(v)$. □

When the tree T is clear from the context we may not write the subscript T in the procedure names. Note that since the $\textsc{Emb}_T(v)$ is a deep set we can assume in an implementation of EMB that PARENT, FL, NCA, and MOP take deep sets as input. We will use this fact in the following sections.

3.4 A Simple Tree Inclusion Algorithm

In this section we a present a simple implementation of the set procedures which leads to an efficient tree inclusion algorithm. Subsequently, we modify one of the procedures to obtain a family of tree inclusion algorithms where the complexities depend on the solution to a well-studied problem known as the *tree color problem*.

3.4.1 Preprocessing

To compute deep embeddings efficiently we require a data structure for T which allows us, for any $v, w \in V(T)$, to compute $\mathrm{nca}_T(v, w)$ and determine if $v \prec w$ or $v \lhd w$. In linear time we can compute $\mathrm{pre}(v)$ and $\mathrm{post}(v)$ for all nodes $v \in V(T)$, and with these it is straightforward to test the two conditions. Furthermore,

Lemma 3.4.1 (Harel and Tarjan[66]). *For any tree T there is a data structure using $O(n_T)$ space and preprocessing time which supports nearest common ancestor queries in $O(1)$ time.*

Hence, our data structure uses linear preprocessing time and space.

3.4.2 Implementation of the Set Procedures

To answer tree inclusion queries we give an efficient implementation of the set procedures. The idea is to represent the node sets in a left-to-right order. For this purpose we introduce some helpful notation. A *node list*, X, is a list of nodes. If $v_i \lhd v_{i+1}$, $1 \leq i < |X|$ then X is *ordered* and if $v_1 \unlhd v_{i+1}$, $1 \leq i < |X|$ then X is

semiordered. A *node pair list*, Y, is a list of pairs of nodes. We say that Y is ordered if $Y|_1$ and $Y|_2$ are ordered, and semiordered if $Y|_1$ and $Y|_2$ are semiordered.

The set procedures are implemented using node lists and node pair lists below. All lists used in the procedures are either ordered or semiordered. As noted in Section 3.3 we may assume that the input to all of the procedures, except DEEP, represent a deep set, that is, the corresponding node list or node pair list is ordered. We assume that the input list given to DEEP is semiordered and the output, of course, is ordered. Hence, the output of all the other set procedures must be semiordered.

$\textsc{Parent}_T(X)$. Return the list $Z := [\text{parent}(X[i]) \mid 1 \leq i \leq |X|]$.

$\textsc{Nca}_T(Y)$. Return the list $Z := [\text{nca}(Y[i]) \mid 1 \leq i \leq |Y|]$.

$\textsc{Deep}_T(X)$. Initially, set $v := X[1]$ and $Z := []$. For each i, $2 \leq i \leq k$, compare v and $X[i]$: If $v \lhd X[i]$ set $Z := Z \circ v$ and $v := X[i]$. If $v \prec X[i]$, set $v := X[i]$ and otherwise ($X[i] \prec v$) do nothing.

Finally, set $Z := Z \circ v$ and return Z.

$\textsc{Mop}_T(X, Y)$. Initially, set $Z := []$. Find the minimum j such that $X[1]_2 \lhd Y[j]$ and set $x := X[1]_1$, $y := Y[j]$, and $h := j$. If no such j exists stop.

As long as $h \leq |Y|$ do the following: For each i, $2 \leq i \leq |X|$, do: Set $h := h+1$ until $X[i]_2 \lhd Y[h]$. Compare $Y[h]$ and y: If $y = Y[h]$ set $x := X[i]_1$. If $y \lhd Y[h]$ set $Z := Z \circ (x, y)$, $x := X[i]_1$, and $y := Y[h]$.

Finally, set $Z := Z \circ (x, y)$ and return Z.

$\textsc{Fl}_T(X, \alpha)$. Initially, set $Y := X$, $Z := []$, and $S := []$. Repeat until $Y := []$: For $i = 1, \ldots, |Y|$ if $l(Y[i]) = \alpha$ set $Z := \textsc{Insert}(Y[i], Z)$ and otherwise set $S := S \circ \text{parent}(Y[i])$.

Set $S := \textsc{Deep}_T(S)$, $Y := \textsc{Deep}^*_T(S, Z)$, $S := []$.

Return Z.

The procedure FL calls two auxiliary procedures: $\textsc{Insert}(v, Z)$ that takes an ordered list Z and insert the node v such that the resulting list is ordered, and $\textsc{Deep}^*(S, Z)$ that takes two ordered lists and returns the ordered list representing the set $\textsc{Deep}(S \cup Z) \cap S$, i.e., $\textsc{Deep}^*(S, Z) = [s \in S | \nexists z \in Z : s \prec z]$. Below we describe in more detail how to implement FL together with the auxiliary procedures.

We use one doubly linked list to represent all the lists Y, S, and Z. For each element in Y we have pointers Pred and Succ pointing to the predecessor and successor in the list, respectively. We also have at each element a pointer Next pointing to the next element in Y. In the beginning Next $=$ Succ for all elements, since all elements in the list are in Y. When going through Y in one iteration we simple follow the Next pointers. When FL calls INSERT$(Y[i], Z)$ we set Next(Pred$(Y[i])$) to Next$(Y[i])$. That is, all nodes in the list not in Y, i.e., nodes not having a Next pointer pointing to them, are in Z. We do not explicitly maintain S. Instead we just set save PARENT$(Y[i])$ at the position in the list instead of $Y[i]$. Now DEEP(S) can be performed following the Next pointers and removing elements from the doubly linked list accordingly to procedure DEEP. It remains to show how to calculate DEEP$^*(S, Z)$. This can be done by running through S following the Next pointers. At each node s compare Pred(s) and Succ(s) with s. If one of them is a descendant of s remove s from the doubly linked list.

Using this linked list implementation DEEP$^*(S, Z)$ takes time $O(|S|)$, whereas using DEEP to calculate this would have used time $O(|S| + |Z|)$.

3.4.3 Correctness of the Set Procedures

Clearly, PARENT and NCA are correct. The following lemmas show that DEEP, FL, and MOP are also correctly implemented.

Lemma 3.4.2. *Procedure* DEEP(X) *is correct.*

Proof. Let u be an element in X. We will first prove that if $X \cap V(T(u)) = \emptyset$ then $u \in Z$. Since $X \cap V(T(u)) = \emptyset$ we must at some point during the procedure have $v = u$, and v will not change before u is added to Z. If u occurs several times in X we will have $v = u$ each time we meet a copy of u (except the first) and it follows from the implementation that u will occur exactly once in Z.

We will now prove that if $X \cap V(T(u)) \neq \emptyset$ then $u \notin Z$. Let w be the rightmost and deepest descendant of u in X. There are two cases:

1. u is before w in X. Look at the time in the execution of the procedure when we look at w. There are two cases.

   (a) $v = u$. Since $u \prec w$ we set $v = w$ and proceed. It follows that $u \notin Z$.

   (b) $v = x \neq u$. Since any node to the left of u also is to the left of w and X is an semiordered list we must have $x \in V(T(u))$ and thus $u \notin Z$.

2. u is after w in X. Since w is the rightmost and deepest ancestor of u and X is semiordered we must have $v = w$ at the time in the procedure where we look at u. Therefore $u \notin Z$.

If u occurs several times in X, each copy will be taken care of by either case 1. or 2. □

To show that FL is correct we need the following proposition.

Proposition 3.4.3. *Let X be an ordered list and let v be an ancestor of $X[i]$ for some $i \in \{1, \ldots, k\}$. If v is an ancestor of some node in X other than $X[i]$ then v is an ancestor of $X[i-1]$ or $X[i+1]$.*

Proof. Assume for the sake of contradiction that $v \not\prec X[i-1]$, $v \not\prec X[i+1]$, and $v \prec w$, where $w \in X$. Since X is ordered either $w \lhd X[i-1]$ or $X[i+1] \lhd w$. Assume $w \lhd X[i-1]$. Since $v \prec X[i]$ and $X[i-1]$ is the left of $X[i]$, $X[i-1]$ is to the left of v contradicting $v \prec w$. Assume $X[i+1] \lhd w$. Since $v \prec X[i]$ and $X[i+1]$ is the right of $X[i]$, $X[i-1]$ is to the right of v contradicting $v \prec w$. □

Proposition 3.4.3 shows that the doubly linked list implementation of DEEP* is correct. Clearly, INSERT is implemented correct be the doubly linked list representation, since the nodes in the list remains in the same order throughout the execution of the procedure.

Lemma 3.4.4. *Procedure* FL(V, α) *is correct.*

Proof. Let $F = \{\mathrm{fl}(v, \alpha) \mid v \in X\}$. It follows immediately from the implementation of the procedure that FL$(X, \alpha) \subseteq F$. It remains to show that DEEP$(F) \subseteq$ FL(X, α). Let v be a node in DEEP(F)), let $w \in X$ be the node such that $v = \mathrm{fl}(w, \alpha)$, and let $w = v_1, v_2, \ldots, v_k = v$ be the nodes on the path from w to v. In each iteration of the algorithm we have $v_i \in Y$ for some i unless $v \in Z$. □

Lemma 3.4.5. *Procedure* MOP(X, Y) *is correct.*

Proof. We want to show that for $1 \leq j < |X|$, $1 \leq t < |Y|$, $(X[j]_1, Y[t]) \in Z$ iff $(X[j]_2, Y[t]) \in \mathrm{mop}(X|_2, Y)$. Since $X|_2$ and Y are ordered lists

$$(X[j]_2, Y[t]) \in \mathrm{mop}(X|_2, Y) \iff Y[t-1] \trianglelefteq X[j]_2 \lhd Y[t] \trianglelefteq X[j+1]_2. \quad (3.1)$$

First we show that $(X[j]_1, Y[t]) \in Z \Rightarrow (X[j]_2, Y[t]) \in \mathrm{mop}(X|_2, Y)$. We will break the proof into three parts, each showing one of the inequalities from the right hand side of (3.1).

- $Y[t-1] \trianglelefteq X[j]_2$: We proceed by induction on j. Base case $j = 1$: Immediately from the implementation of the procedure. $j > 1$: We have $x = X[j-1]_1$ and $y = Y[h]$ for some $h \leq t$. By the induction hypothesis $Y[j-1] \trianglelefteq X[j-1]_2$. If $X[j]_2 \triangleleft Y[h]$ then $h = t$ since $Y[h-1] \trianglelefteq X[j-1]_2 \triangleleft X[j]_2$ and thus $Y[t-1] \triangleleft X[j]_2$. If $X[j]_2 \trianglerighteq Y[h]$ then $h \leq t-1$ and thus $Y[t-1] \trianglelefteq X[j]_2$.

- $X[j]_2 \triangleleft Y[t]$: Follows immediately from the implementation of the procedure.

- $X[j+1]_2 \trianglerighteq Y[t]$: Assume $X[j+1]_2 \triangleleft Y[t]$. Consider the time in the procedure when we look at $X[j+1]_2$. We have $y = Y[t]$ and thus set $x := X[j+1]_1$ contradicting $(X[j]_1, Y[t]) \in Z$.

It follows immediately from the implementation of the procedure, that if $X[j]_2 \triangleleft Y[t]$, $Y[t-1] \trianglelefteq X[j]_2$, and $X[j+1]_2 \trianglerighteq Y[t]$ then $(X[j]_1, Y[t]) \in Z$. □

3.4.4 Complexity of the Set Procedures

For the running time of the node list implementation observe that, given the data structure described in Section 3.4.1, all set procedures, except FL, perform a single pass over the input using constant time at each step. Hence we have,

Lemma 3.4.6. *For any tree T there is a data structure using $O(n_T)$ space and preprocessing which supports each of the procedures* PARENT*,* DEEP*,* MOP*, and* NCA *in linear time (in the size of their input).*

The running time of a single call to FL might take time $O(n_T)$. Instead we will divide the calls to FL into groups and analyze the total time used on such a group of calls. The intuition behind the division is that for a path in P the calls made to FL by EMB is done bottom up on disjoint lists of nodes in T.

Lemma 3.4.7. *For disjoint ordered node lists $V_1, \ldots, V_k$ and labels $\alpha_1, \ldots, \alpha_k$, such that any node in V_{i+1} is an ancestor of some node in* DEEP*$(\mathrm{FL}_T(V_i, \alpha_i))$, $2 \leq i < k$, all of the calls $\mathrm{FL}_T(V_1, \alpha_1), \ldots, \mathrm{FL}_T(V_k, \alpha_k)$ can be computed in $O(n_T)$ total time.*

Proof. Let Y, Z, and S be as in the implementation of the procedure. Since DEEP and DEEP* takes time $O(S)$, we only need to show that the total length of the lists S—summed over all the calls—is $O(n_T)$ to analyze the total time usage of DEEP and DEEP*. We note that in one iteration $|S| \leq |Y|$. INSERT takes constant time and it is thus enough to show that any node in T can be in Y at most twice during all calls to FL.

Consider a call to FL. Note that Y is ordered at all times. Except for the first iteration, a node can be in Y only if one of its children were in Y in the last iteration. Thus in one call to FL a node can be in Y only once.

Look at a node u the first time it appears in Y. Assume that this is in the call $\text{FL}(V_i, \alpha_i)$. If $u \in X$ then u cannot be in Y in any later calls, since no node in V_j where $j > i$ can be a descendant of a node in V_i. If $u \notin Z$ in this call then u cannot be in Y in any later calls. To see this look at the time when u removed from Y. Since the set $Y \cup Z$ is deep at all times no descendant of u will appear in Y later in this call to FL, and no node in Z can be a descendant of u. Since any node in V_j, $j > i$, is an ancestor of some node in $\text{DEEP}(\text{FL}(V_i, \alpha_i))$ neither u or any descendant of u can be in any V_j, $j > i$. Thus u cannot appear in Y in any later calls to FL. Now if $u \in Z$ then we might have $u \in V_{i+1}$. In that case, u will appear in Y in the first iteration of the procedure call $\text{FL}(V_{i+1}, \alpha_i)$, but not in any later calls since the lists are disjoint, and since no node in V_j where $j > i + 1$ can be a descendant of a node in V_{i+1}. If $u \in Z$ and $u \notin V_{i+1}$ then clearly u cannot appear in Y in any later call. Thus a node in T is in Y at most twice during all the calls. ☐

3.4.5 Complexity of the Tree Inclusion Algorithm

Using the node list implementation of the set procedures we get:

Theorem 3.4.8. *For trees P and T the tree inclusion problem can be solved in $O(l_P n_T)$ time and $O(n_P + n_T)$ space.*

Proof. By Lemma 3.4.6 we can preprocess T in $O(n_T)$ time and space. Let $g(n)$ denote the time used by FL on a list of length n. Consider the time used by $\text{EMB}_T(\text{root}(P))$. We bound the contribution for each node $v \in V(P)$. From Lemma 3.4.6 it follows that if v is a leaf the cost of v is at most $O(g(l_T))$. Hence, by Lemma 3.4.7, the total cost of all leaves is $O(l_P g(l_T)) = O(l_P n_T)$. If v has a single child w the cost is $O(g(|\text{EMB}_T(w)|))$. If v has more than one child the cost of MOP, NCA, and DEEP is bounded by $\sum_{w \in \text{child}(v)} O(|\text{EMB}_T(w)|)$. Furthermore, since the length of the output of MOP (and thus NCA) is at most $z = \min_{w \in \text{child}(v)} |\text{EMB}_T(w)|$ the cost of FL is $O(g(z))$. Hence, the total cost for internal nodes is,

$$\sum_{v \in V(P) \setminus L(P)} O\left(g(\min_{w \in \text{child}(v)} |\text{EMB}_T(w)|) + \sum_{w \in \text{child}(v)} |\text{EMB}_T(w)| \right) \leq \sum_{v \in V(P)} O(g(|\text{EMB}_T(v)|)). \tag{3.2}$$

Next we bound (3.2). For any $w \in \text{child}(v)$ we have that $\text{EMB}_T(w)$ and $\text{EMB}_T(v)$ are disjoint ordered lists. Furthermore we have that any node in $\text{EMB}_T(v)$ must be an ancestor of some node in $\text{DEEP}_T(\text{FL}_T(\text{EMB}_T(w), l(v)))$. Hence, by Lemma 3.4.7, for

any leaf to root path $\delta = v_1, \ldots, v_k$ in P, we have that $\sum_{u \in \delta} g(|\textsc{Emb}_T(u)|) \leq O(n_T)$. Let Δ denote the set of all root to leaf paths in P. It follows that,

$$\sum_{v \in V(T)} g(|\textsc{Emb}_T(v)|) \leq \sum_{p \in \Delta} \sum_{u \in p} g(|\textsc{Emb}_T(u)|) \leq O(l_P n_T).$$

Since this time dominates the time spent at the leaves the time bound follows. Next consider the space used by $\textsc{Emb}_T(\mathrm{root}(P))$. The preprocessing described in Section 3.4.1 uses only $O(n_T)$ space. Furthermore, by induction on the size of the subtree $P(v)$ it follows immediately that at each step in the algorithm at most $O(\max_{v \in V(P)} |\textsc{Emb}_T(v)|)$ space is needed. Since $\textsc{Emb}_T(v)$ a deep embedding, it follows that $|\textsc{Emb}_T(v)| \leq l_T$. □

3.4.6 An Alternative Algorithm

In this section we present an alternative algorithm. Since the time complexity of the algorithm in the previous section is dominated by the time used by $\textsc{Fl}$, we present an implementation of this procedure which leads to a different complexity. Define a *firstlabel data structure* as a data structure supporting queries of the form $\mathrm{fl}(v, \alpha)$, $v \in V(T)$, $\alpha \in \Sigma$. Maintaining such a data structure is known as the *tree color problem*. This is a well-studied problem, see e.g. [41, 93, 49, 7]. With such a data structure available we can compute $\textsc{Fl}$ as follows,

$\textsc{Fl}(X, \alpha)$ Return the list $Z := [\mathrm{fl}(X[i], \alpha) \mid 1 \leq i \leq |X|]$.

Theorem 3.4.9. *Let P and T be trees. Given a firstlabel data structure using $s(n_T)$ space, $p(n_T)$ preprocessing time, and $q(n_T)$ time for queries, the tree inclusion problem can be solved in $O(p(n_T) + n_P l_T \cdot q(n_T))$ time and $O(n_P + s(n_T) + n_T)$ space.*

Proof. Constructing the firstlabel data structures uses $O(s(n_T))$ and $O(p(n_T))$ time. As in the proof of Theorem 3.4.8 the total time used by $\textsc{Emb}_T(\mathrm{root}(P))$ is bounded by $\sum_{v \in V(P)} g(|\textsc{Emb}_T(v)|)$, where $g(n)$ is the time used by $\textsc{Fl}$ on a list of length n. Since $\textsc{Emb}_T(v)$ is a deep embedding and each fl takes $q(n_T)$ we have,

$$\sum_{v \in V(P)} g(|\textsc{Emb}_T(v)|) \leq \sum_{v \in V(P)} g(l_T) = n_P l_T \cdot q(n_T).$$

□

Several firstlabel data structures are available, for instance, if we want to maintain linear space we have,

Lemma 3.4.10 (Dietz [41]). *For any tree T there is a data structure using $O(n_T)$ space, $O(n_T)$ expected preprocessing time which supports firstlabel queries in $O(\log\log n_T)$ time.*

The expectation in the preprocessing time is due to perfect hashing. Since our data structure does not need to support efficient updates we can remove the expectation by using the deterministic dictionary by Hagerup et. al. [63]. This gives a worst-case preprocessing time of $O(n_T \log n_T)$, however, using a simple two-level approach this can be reduced to $O(n_T)$ (see e.g. [119]). Plugging in this data structure we obtain,

Corollary 3.4.11. *For trees P and T the tree inclusion problem can be solved in $O(n_P + n_T)$ space and $O(n_P\, l_T \log\log n_T)$ time.*

3.5 A Faster Tree Inclusion Algorithm

In this section we present a new tree inclusion algorithm which has a worst-case subquadratic running time. As discussed in the introduction the general idea is cluster T into small trees of logarithmic size which we can efficiently preprocess and then use this to speedup the computation with a logarithmic factor.

3.5.1 Clustering

In this section we describe how to divide T into micro trees and how the macro tree is created. For simplicity in the presentation we assume that T is a binary tree. If this is not the case it is straightforward to construct a binary tree B, where $n_B \leq 2n_T$, and a mapping $g : V(T) \to V(B)$ such that for any pair of nodes $v, w \in V(T)$, $l(v) = l(g(v))$, $v \prec w$ iff $g(v) \prec g(w)$, and $v \lhd w$ iff $g(v) \lhd g(w)$. If the nodes in the set $U = V(B) \backslash \{g(v) \mid v \in V(T)\}$ is assigned a special label $\beta \notin \Sigma$ it follows that for any tree P, $P \sqsubseteq T$ iff $P \sqsubseteq B$.

Let C be a connected subgraph of T. A node in $V(C)$ incident to a node in $V(T)\backslash V(C)$ is a *boundary* node. The boundary nodes of C are denoted by δC. A *cluster* of C is a connected subgraph of C with at most two boundary nodes. A set of clusters CS is a *cluster partition* of T iff $V(T) = \cup_{C \in CS} V(C)$, $E(T) = \cup_{C\in CS} E(C)$, and for any $C_1, C_2 \in CS$, $E(C_1) \cap E(C_2) = \emptyset$, $|E(C_1)| \geq 1$, $\mathrm{root}(T) \in \delta C$ if $\mathrm{root}(T) \in V(C)$. If $|\delta C| = 1$ we call C a *leaf cluster* and otherwise an *internal cluster*.

We use the following recursive procedure $\textsc{Cluster}_T(v, s)$, adopted from Alstrup and Rauhe [8], which creates a cluster partition CS of the tree $T(v)$ with the

property that $|CS| = O(s)$ and $|V(C)| \leq \lceil n_T/s \rceil$. A similar cluster partitioning achieving the same result follows from [6, 5, 52].

$\text{CLUSTER}_T(v, s)$. For each child u of v there are two cases:

1. $|V(T(u))| + 1 \leq \lceil n_T/s \rceil$. Let the nodes $\{v\} \cup V(T(u))$ be a leaf cluster with boundary node v.
2. $|V(T(u))| > \lceil n_T/s \rceil$. Pick a node $w \in V(T(u))$ of maximum depth such that $|V(T(u))| + 2 - |V(T(w))| \leq \lceil n_T/s \rceil$.

   Let the nodes $V(T(u)) \backslash V(T(w)) \cup \{v, w\}$ be an internal cluster with boundary nodes v and w. Recursively, compute $\text{CLUSTER}_T(w, s)$.

Lemma 3.5.1. *Given a tree T with $n_T > 1$ nodes, and a parameter s, where $\lceil n_T/s \rceil \geq 2$, we can build a cluster partition CS in $O(n_T)$ time, such that $|CS| = O(s)$ and $|V(C)| \leq \lceil n_T/s \rceil$ for any $C \in CS$.*

Proof. The procedure $\text{CLUSTER}_T(\text{root}(T), s)$ clearly creates a cluster partition of T and it is straightforward to implement in $O(n_T)$ time. Consider the size of the clusters created. There are two cases for u. In case 1, $|V(T(u))| + 1 \leq \lceil n_T/s \rceil$ and hence the cluster $C = \{v\} \cup V(T(u))$ has size $|V(C)| \leq \lceil n_T/s \rceil$. In case 2, $|V(T(u))| + 2 - |V(T(w))| \leq \lceil n_T/s \rceil$ and hence the cluster $C = V(T(u)) \backslash V(T(w)) \cup \{v, w\}$ has size $|V(C)| \leq \lceil n_T/s \rceil$.

Next consider the size of the cluster partition. We say that a cluster C is *bad* if $|V(C)| \leq c/2$ and *good* otherwise. We will show that at least a constant fraction of the clusters in the cluster partition are good. Let $c = \lceil n_T/s \rceil$. It is easy to verify that the cluster partition created by procedure CLUSTER has the following properties:

(i) Let C be a bad internal cluster with boundary nodes v and w ($v \prec w$). Then w has two children with at least $c/2$ descendants each.

(ii) Let C be a bad leaf cluster with boundary node v. Then the boundary node v is contained in a good cluster.

By (ii) the number of bad leaf clusters is no larger than twice the number of good internal clusters. By (i) each bad internal cluster C is sharing its lowest boundary node of C with two other clusters, and each of these two clusters are either internal clusters or good leaf clusters. This together with (ii) shows that number of bad clusters is at most a constant fraction of the total number of clusters. Since a good cluster is of size more than $c/2$, there can be at most $2s$ good clusters and thus $|CS| = O(s)$. □

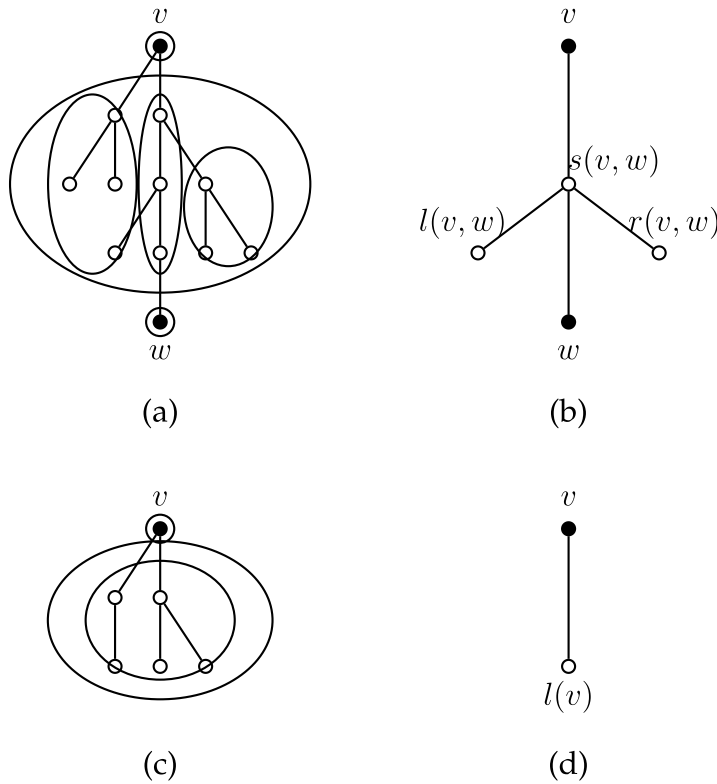


Figure 3.4: The clustering and the macro tree. (a) An internal cluster. The black nodes are the boundary node and the internal ellipses correspond to the boundary nodes, the right and left nodes, and spine path. (b) The macro tree corresponding to the cluster in (a). (c) A leaf cluster. The internal ellipses are the boundary node and the leaf nodes. (d) The macro tree corresponding to the cluster in (c).

Let $C \in CS$ be an internal cluster $v, w \in \delta C$. The *spine path* of C, $\pi(C)$, is the path between v, w excluding v and w. A node on the spine path is a *spine node*. A node to the left and right of v, w, or any node on $\pi(C)$ is a *left node* and *right node* respectively. If C is a leaf cluster with $v \in \delta C$ then any proper descendant of v is a *leaf node*.

Let CS be a cluster partition of T as described in Lemma 3.5.1. We define an ordered *macro tree* T^M. Our definition of T^M may be viewed as an "ordered" version of the macro tree given by Alstrup and Rauhe [8]. For each internal cluster $C \in CS$, $v, w \in \delta C$, $v \prec w$, we have the node $s(v, w)$ and edges $(v, s(v, w)), (s(v, w), w)$. Furthermore, we have the nodes $l(v, w)$ and $r(v, w)$ and edges $(l(v, w), s(v, w))$ and $(r(v, w), s(v, w))$ ordered such that $l(v, w) \lhd w \lhd r(v, w)$. If C is a leaf cluster and $v \in \delta C$ we have the node $l(v)$ and edge $(l(v), v)$. Since $\mathrm{root}(T)$ is a boundary node T^M is rooted at $\mathrm{root}(T)$. Figure 3.4 illustrates these definitions.

To each node $v \in V(T)$ we associate a unique macro node denoted $i(v)$. If $u \in V(C)$ and $C \in CS$, then

$$i(u) = \begin{cases} u & \text{If } u \text{ is boundary,} \\ s(v,w) & \text{if } u \text{ is a spine node and } v,w \in \delta C, \\ l(v,w) & \text{if } u \text{ is a left node and } v,w \in \delta C, \\ r(v,w) & \text{if } u \text{ is a right node and } v,w \in \delta C, \\ l(v) & \text{if } u \text{ is a leaf node and } v \in \delta C. \end{cases}$$

Conversely, for any macro node $x \in V(T^M)$ define the *macro-induced subgraph*, denoted $I(x)$, as the induced subgraph of T of the set of nodes $\{v \mid v \in V(T), x = i(v)\}$. We also assign a *set* of labels to x given by $l(x) = \{l(v) \mid v \in V(I(x))\}$. If x is spine node or a boundary node the unique node in $V(I(x))$ of greatest depth is denoted by $\text{first}(x)$. Finally, for any set of nodes $\{x_1, \ldots, x_k\} \subseteq V(T^M)$ we define $I(x_1, \ldots, x_k)$ as the induced subgraph of the set of nodes $V(I(x_1)) \cup \cdots \cup V(I(x_k))$.

The following propositions states useful properties of ancestors, nearest common ancestor, and the left-to-right ordering in the cluster partition and in T. The propositions follows directly from the definition of the clustering.

Proposition 3.5.2. *For any pair of nodes $v, w \in V(T)$, the following hold*

(i) If $i(v) = i(w)$ then $v \prec_T w$ iff $v \prec_{I(i(v))} w$.

(ii) If $i(v) \neq i(w)$, $i(v) \in \{s(v', w'), v'\}$ and $i(w) \in \{l(v', w'), r(v', w')\}$, then $v \prec_T w$ iff $v \prec_{I(i(v), s(v', w'), v')} w$.

(iii) In all other cases, $w \prec_T v$ iff $i(w) \prec_{T^M} i(v)$.

Proposition 3.5.3. *For any pair of nodes $v, w \in V(T)$, the following hold*

(i) If $i(v) = i(w)$ then $v \lhd w$ iff $v \lhd_{I(i(v))} w$.

(ii) If $i(v) = l(v', w')$, $i(w) \in \{s(v', w'), v'\}$ then $v \lhd w$ iff $v \lhd_{I(l(v', w'), s(v', w'), v')} w$.

(iii) If $i(v) = r(v', w')$, $i(w) \in \{s(v', w'), v'\}$ then $w \lhd v$ iff $w \lhd_{I(r(v', w'), s(v', w'), v')} v$.

(iv) In all other cases, $v \lhd w$ iff $i(v) \lhd_{T^M} i(w)$.

Proposition 3.5.4. *For any pair of nodes $v, w \in V(T)$, the following hold*

(i) If $i(v) = i(w) = l(v')$ then $\text{nca}_T(v, w) = \text{nca}_{I(i(v), v')}(v, w)$.

(ii) If $i(v) = i(w) \in \{l(v', w'), r(v', w')\}$ then $\text{nca}_T(v, w) = \text{nca}_{I(i(v), s(v', w'), v')}(v, w)$.

(iii) *If* $i(v) = i(w) = s(v', w')$ *then* $\mathrm{nca}_T(v, w) = \mathrm{nca}_{I(i(v))}(v, w)$.

(iv) *If* $i(v) \neq i(w)$ *and* $i(v), i(w) \in \{l(v', w'), r(v', w'), s(v', w')\}$ *then* $\mathrm{nca}_T(v, w) = \mathrm{nca}_{I(i(v),i(w),s(v',w'),v')}(v, w)$.

(v) *If* $i(v) \neq i(w)$, $i(v) \in \{l(v', w'), r(v', w'), s(v', w')\}$, *and* $i(w) \preceq_{T^M} w'$ *then* $\mathrm{nca}_T(v, w) = \mathrm{nca}_{I(i(v),s(v',w'),w')}(v, w')$.

(vi) *If* $i(v) \neq i(w)$, $i(w) \in \{l(v', w'), r(v', w'), s(v', w')\}$, *and* $i(v) \preceq_{T^M} w'$ *then* $\mathrm{nca}_T(v, w) = \mathrm{nca}_{I(i(w'),s(v',w'),w')}(w, w')$.

(vii) *In all other cases,* $\mathrm{nca}_T(v, w) = \mathrm{nca}_{T^M}(i(v), i(w))$.

3.5.2 Preprocessing

In this section we describe how to preprocess T. First we make a cluster partition CS of the tree T with clusters of size s, to be fixed later, and the corresponding macro tree T^M in $O(n_T)$ time. The macro tree is preprocessed as in 3.4.1. However, since nodes in T^M contain a set of labels, we store for each node $v \in V(T^M)$ a dictionary of $l(v)$. Using perfect hashing the total time to compute all these dictionaries is $O(n_T)$ expected time. Furthermore, we modify the definition of fl such that $\mathrm{fl}_{T^M}(v, \alpha)$ is the nearest ancestor w of v such that $\alpha \in l(w)$.

Next we show how to preprocess the micro trees. For any labeled, ordered, forest S and $M, N \subseteq V(S)$ we define, in addition, to the set procedures the following useful procedures.

ANCESTOR$_S(M)$. Return the set of all ancestors of nodes in M.

LEFTOF$_S(M, N)$. Return a boolean indicating whether there is at least one node $v \in M$ such that for all nodes $w \in S$, $v \trianglelefteq w$.

LEFT$_S(M)$. Return the leftmost node in M.

RIGHT$_S(M)$. Return the rightmost node in M.

MATCH$_S(M, N, O)$, where $M = \{m_1 \lhd \cdots \lhd m_k\}$, $N = \{v_1 \lhd \cdots \lhd v_k\}$, $O = \{o_1 \lhd \cdots \lhd o_l\}$, and $o_i = v_j$ for some j. Return the set $R := \{m_j \mid o_i = v_j, 1 \leq i \leq l\}$.

MOP$_S(M, N)$ Return the triple $(R_1, R_2, \mathsf{bool})$. Where $R_1 = \mathrm{mop}(M, N)|_1$ and $R_2 = \mathrm{mop}(M, N)|_2$, and bool indicates whether there is any node in $v \in M$ such that for all nodes $w \in N$, $v \trianglerighteq w$.

MASK$_S(\alpha)$, $\alpha \in \Sigma$. Return the set of nodes with label α.

We show how to implement these procedures on all macro induced subforest S of each cluster $C \in CS$. Note that a cluster contains at most a constant number of such subforests. For the procedures PARENT_S, ANCESTOR_S, DEEP_S, NCA_S, LEFTOF_S, LEFT_S, RIGHT_S, MATCH_S, and MOP_S we will simply precompute and store the result of any input for all forests S with at most s nodes. We assume that the input and output node sets of the above procedures is given as a bitstring of length s. Hence, for any forest S the total of number of distinct node sets is at most 2^s. Since at most 3 input sets occur in the procedures and the total number of forest of size at most s is $O(2^{2s})$ it follows that there are $2^{O(s)}$ distinct inputs to each procedure to precompute and store. Furthermore, it is straightforward to compute all results within the same time bound. If c is the constant hidden in the O notation we set $s = \frac{1}{c} \log n_T$ and the total preprocessing time and space used becomes $2^{cs} = 2^{\log n_T} = n_T$. Furthermore, since the size of the input to procedures is logarithmic we can lookup the result of any input in constant time.

Next we show how to compute the remaining procedures MASK_S and FL_S. Note that since the size of the alphabet is potentially $\Omega(n_T)$, we cannot precompute all values for these procedures in $O(n_T)$ time. Instead we implement MASK_S using a dictionary for each subforest S indexed by the labels in S. Again, using perfect hashing we can build all such tables in $O(n_T)$ excepted time using linear space. Hence, we can lookup MASK_S in constant time. Finally, we can compute FL in constant time with the other procedures since

$$\text{FL}_S(M, \alpha) = \text{DEEP}_S(\text{ANCESTOR}_S(M) \texttt{ and } \text{MASK}_S(\alpha)),$$

where `and` denotes a bitwise and operation.

As discussed in Section 3.4.6, if we require worst-case running times instead of the expected $O(n_T)$ time above we may instead use a deterministic dictionary without changing the overall running time of our tree inclusion algorithm.

3.5.3 A Compact Representation of Node Sets

In this section we show how to implement the set procedures in sublinear time using the clustering and preprocessing defined in the previous section.

First we define a compact representation of node sets, which we call micro-macro node sets. A *micro-macro node set* (mm-node set) $\mathcal{V}$ for a tree T with macro tree T^M is a set of pairs $\mathcal{V} = \{(x_1, M(x_k)), \dots, (x_k, M(x_k))\}$, such that for any pair $(x, M(x)) \in \mathcal{V}$:

(i) $x \in V(T^M)$,

(ii) $M(x) \subseteq V(I(x))$,

(iii) $M(x) \neq \emptyset$.

Additionally, if for any pairs $(x, M(x)), (y, M(y)) \in \mathcal{X}$:

(iv) $x \neq y$,

we say that $\mathcal{V}$ is *canonical*. For any mm-node set $\mathcal{V}$ there is a corresponding set of nodes $S(\mathcal{V}) \subseteq V(T)$ given by $S(\mathcal{V}) = \cup_{(x,M(x))\in\mathcal{V}} M(x)$. Conversely, given a set of nodes V there is a unique canonical mm-node set $\mathcal{V}$ given by:

$$\mathcal{V} = \{(x, M(x)) \mid M(x) = V(I(x)) \cap X \neq \emptyset\}.$$

We say that $\mathcal{V}$ is deep iff the set $S(\mathcal{V})$ is deep. Note that by Lemma 3.5.2(ii) an mm-node set $\mathcal{V}$ may be deep even though the node set $\mathcal{V}|_1$ is not. Since the size of the macro tree is $O(n_T/\log n_T)$ we have that,

Lemma 3.5.5. *For any canonical mm-node set $\mathcal{V}$, $|\mathcal{V}| \leq O(n_T/\log n_T)$.*

As with node lists in the simple implementation, we define a *micro-macro node list* (mm-node list), $\mathcal{X} = [(x_1, M(x_k)), \ldots, (x_k, M(x_k))]$, as a list where each element is an element of an mm-node set. We say that $\mathcal{X}$ is ordered if $x_1 \lhd_{T^M} \cdots \lhd_{T^M} x_k$ and semiordered if $x_1 \unlhd_{T^M} \cdots \unlhd_{T^M} x_k$.

In the following we show how to implement the set procedures using mm-node lists. As before we assume that the input to each of the procedures is deep. Each of the procedures, except DEEP, accept as input mm-node lists which are semiordered, canonical, and deep and return as output semiordered mm-node list(s). The input for DEEP is semiordered and canonical and the output is semiordered, canonical, and deep. Since the output of the procedures is not necessarily canonical and DEEP requires canonical input we need the following additional procedure to make EMB work:

CANONICAL$(\mathcal{X})$, where $\mathcal{X}$ is a semiordered mm-node list. Return a semiordered canonical mm-list $\mathcal{R}$ such that $S(\mathcal{R}) = S(\mathcal{X})$.

We simply run this procedure on any input mm-node list to DEEP immediately before executing DEEP.

3.5.4 Implementation of the Set Procedures

The implementation of all set procedures is described in this section.

PARENT($\mathcal{X}$). Initially, set $\mathcal{R} := []$. For each i, $2 \leq i \leq |\mathcal{X}|$, set $(x, M(x)) := \mathcal{X}[i]$. There are three cases:

1. $x \in \{l(v,w), r(v,w)\}$. Compute $N = \text{PARENT}_{I(x,s(v,w),v)}(M(x))$. For each macro node $s \in \{x, s(v,w), v\}$ (in semiorder) set $\mathcal{R} := \mathcal{R} \circ (s, N \cap V(I(s)))$ if $N \cap V(I(s)) \neq \emptyset$.
2. $x = l(v)$. Compute $N = \text{PARENT}_{I(x,v)}(M(x))$. For each macro node $s \in \{x, v\}$ (in semiorder) set $\mathcal{R} := \mathcal{R} \circ (s, N \cap V(I(s)))$ if $N \cap V(I(s)) \neq \emptyset$.
3. $x \notin \{l(v,w), r(v,w), l(v)\}$. If $N = \text{PARENT}_{I(x)}(M(x)) \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x, N)$. Otherwise, if $\text{parent}_{TM}(x) \neq \bot$ set $\mathcal{R} := \mathcal{R} \circ (\text{parent}_{TM}(x), \text{first}(\text{parent}_{TM}(x)))$.

Return $\mathcal{R}$.

Consider the three cases of procedure PARENT. Case 1 handles the fact that left and right nodes may have a spine node or a boundary node as parent. Since no left or right node can have a parent outside their cluster there is no need to compute parents in the macro tree. Case 2 handles the fact that the nodes in a leaf node may have the boundary node as parent. Since none of the nodes in the leaf node can have a parent outside their cluster there is no need to compute parents in the macro tree. Case 3 handles boundary and spine nodes. Since the input to PARENT is deep there is either a parent within the micro tree or we can use the macro tree to compute the parent of the root of the micro tree.

NCA($\mathcal{X}$). Initially, set $\mathcal{R} := []$. For each i, $1 \leq i \leq |\mathcal{X}|$, set $(x, M(x)) := \mathcal{X}[i]_1$ and $(y, M(y)) := \mathcal{X}[i]_2$ and compare x and y. There are two cases:

1. $x = y$: Let $z := x$. There are two subcases:

   If z is a boundary node then set $\mathcal{R} := \mathcal{R} \circ (z, z)$. Otherwise set

$$S := \begin{cases} I(z, v), & \text{if } z = l(v), \\ I(z, s(v,w), v), & \text{if } z \in \{l(v,w), r(v,w)\}, \\ I(z), & \text{if } z = s(v,w). \end{cases}$$

   Compute $M := \text{NCA}_S(M(x), M(y))$. For each macro node s in S (in semiorder) we set $\mathcal{R} := \mathcal{R} \circ (s, M \cap V(I(s)))$ if $M \cap V(I(s)) \neq \emptyset$.

2. $x \neq y$: Compute $z := \text{NCA}_{TM}(x, y)$. There are two subcases:

   If z is a boundary node then set $\mathcal{R} := \mathcal{R} \circ (z, z)$. Otherwise z must be a spine node $s(v,w)$. There are three cases:

(a) If $x \in \{l(v,w), s(v,w)\}$ and $y \in \{s(v,w), r(v,w)\}$ compute $M := \text{NCA}_{I(x,y,s(v,w),v)}(M(x), M(y))$.

(b) If $x = l(v,w)$ and $y \preceq_T w$ compute $M := \text{NCA}_{I(x,s(v,w),w)}(M(x), w)$.

(c) If $y = r(v,w)$ and $x \preceq_T w$ compute $M := \text{NCA}_{I(y,s(v,w),w)}(w, M(y))$.

Set $\mathcal{R} := \mathcal{R} \circ (z, M \cap V(I(z)))$.

Return $\mathcal{R}$.

Consider the two cases of procedure NCA. Case 1 handles the cases (i), (ii), and (iii) from Proposition 3.5.4. Case 2 handles the cases (iv), (v), (vi) and (vii) from Proposition 3.5.4.

DEEP$(\mathcal{X})$. Initially, set $(x, M(x)) := \mathcal{X}[1]$ and $\mathcal{R} := []$. For each i, $2 \leq i \leq |\mathcal{X}|$, set $(x_i, M(x_i)) := \mathcal{X}[i]$ and compare x and x_i:

1. $x \lhd x_i$: Set $\mathcal{R} := \mathcal{R} \circ (x, \text{DEEP}_{I(x)} M(x))$, and $(x, M(x)) := (x_i, M(x_i))$.
2. $x \prec x_i$: If $x_i \in \{l(v,w), r(v,w)\}$ and $x = s(v,w)$ compute $N := \text{DEEP}_{I(x_i, s(v,w))}(M(x) \cup M(x_i))$.
   Then, set $(x, M(x)) := (x, N(x))$ and if $N(x_i) := N \cap I(x_i) \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x_i, N(x_i))$. Otherwise ($x_i \notin \{l(v,w), r(v,w)\}$ or $x \neq s(v,w)$) set $(x, M(x)) := (x_i, M(x_i))$.
3. $x_i \prec x$: As above, with x and x_i replaced by each other.

Return $\mathcal{R}$.

The above DEEP procedure resembles the previous DEEP procedure implemented on the macro tree. The biggest difference is that a mm-node set $\mathcal{X}$ may be deep even though the set $\mathcal{X}|_1$ is not deep in T^M. However, this can only happen for nodes in the same cluster which is straightforward to handle (see Proposition 3.5.2(i) and (ii)).

MOP$(\mathcal{X}, \mathcal{Y})$. Initially, set $\mathcal{R} := []$,$\mathcal{X}' := \mathcal{X}|_1$, $\mathcal{Z} := \mathcal{X}|_2$, $r := \perp$, $s := \perp$, $i := 1$, and $j := 1$. Repeat the following until $i > |\mathcal{X}|$ or $h > |\mathcal{Y}|$:

If $\mathcal{Z}[i]_1 = l(v,w)$ set $j := j + 1$ until $\mathcal{Z}[i]_1 \unlhd \mathcal{Y}[j]_1$ or $\mathcal{Y}[j]_1 = s(v,w)$. If $\mathcal{Z}[i]_1 = s(v,w)$ set $j := j + 1$ until $\mathcal{Z}[i]_1 \unlhd \mathcal{Y}[j]_1$ or $\mathcal{Y}[j]_1 = r(v,w)$. Otherwise set $j := j + 1$ as long as $\mathcal{Z}[i]_1 \rhd \mathcal{Y}[j]_1$.

Set $(x, M(x)) := \mathcal{X}'[i]$, $(z, M(z)) := \mathcal{Z}[i]$, and $(y, M(y)) := \mathcal{Y}[j]$. There are two cases:

1. $z \lhd y$: If $s \lhd y$ set $\mathcal{R} := \mathcal{R} \circ (r, s)$. Set $r := (x, \text{RIGHT}_{I(x)}(M(x)))$, $s := (y, \text{LEFT}_{I(y)}(M(y)))$, and $i := i + 1$.
2. Either
   (a) $z = y$,
   (b) $z = l(v, w)$ and $y = s(v, w)$,
   (c) $z = s(v, w)$ and $y = r(v, w)$.
   If $s \lhd y$ then set $\mathcal{R} := \mathcal{R} \circ (r, s)$.
   If $s = y$ and $\text{LEFTOF}_{I(z)}(M(z), M(y)) = \text{true}$ then set $\mathcal{R} := \mathcal{R} \circ (r, s)$.
   Compute $(M_1, M_2, \text{match_x}) := \text{MOP}_{I(z,y)}(M(z), M(y))$.
   If $M_1 \neq []$ then compute $M := \text{MATCH}(M(x), M(z), M_1)$, and set $\mathcal{R} := \mathcal{R} \circ ((x, M), (y, M_2))$. Set $r := \perp$, $s := \perp$, and $j := j + 1$. If $\text{match_x} = \text{false}$ set $i := i + 1$.

Return $\mathcal{R}$.

The above MOP procedure resembles the previous MOP procedure implemented on the macro tree in one of the cases. Case 1 in the above iteration is almost the same as the previous implementation of the procedure. Case 2(a) are due to the fact that we can have nearest neighbor pairs within a macro-induced subtree $I(x)$. Cases 2(b) and 2(c) takes care of the special cases caused by the spine nodes.

$\text{FL}(\mathcal{X}, \alpha)$. Initially, set $\mathcal{R} := []$ and $S := []$. For each $(x, M(x)) := \mathcal{X}[i]$, $1 \leq i \leq |\mathcal{X}|$ there are 2 cases:

1. $x \in \{l(v, w), r(v, w)\}$. Compute $N = \text{FL}_{I(x,s(v,w),v)}(M(x), \alpha)$. If $N \neq \emptyset$, then for each macro node $s \in \{x, s(v, w), v\}$ (in semiorder) set $\mathcal{R} := \mathcal{R} \circ (s, N \cap V(I(s)))$ if $N \cap V(I(s)) \neq \emptyset$. Otherwise, set $U := U \circ \text{parent}(v)$.
2. $x \notin \{l(v, w), r(v, w)\}$. Compute $N = \text{FL}_{I(x)}(M(x), \alpha)$. If $N \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x, N)$ and otherwise set $U := U \circ \text{parent}(x)$.

Subsequently, compute $S := \text{FL}_{T^M}(U, \alpha)$, and use this result to compute the mm-node list $\mathcal{S} := [(S[i], \text{FL}_{I(S[i])}(\text{first}(S[i]), \alpha)) \mid 1 \leq i \leq |S|]$. Merge the mm-node lists $\mathcal{S}$ and $\mathcal{R}$ with respect to semiorder and return the result.

The FL procedure is similar to PARENT. The cases 1 and 2 compute FL on a micro tree. If the result is within the micro tree we add it to $\mathcal{R}$ and otherwise we store the node in the macro tree which contains parent of the root of the micro tree in a node list S. We then compute FL in the macro tree on the list S and use this to compute the final result.
Finally, we give the trivial CANONICAL procedure.

CANONICAL($\mathcal{X}$). For each node $x \in V(T^M)$ maintain a set $N(x) \subseteq I(V(x))$ initially empty. For each i, $1 \leq i \leq |\mathcal{X}|$ set $N(\mathcal{X}[i]_1) := N(\mathcal{X}[i]_1) \cup \mathcal{X}[i]_2$. Then, set $\mathcal{R} := []$ and traverse T^M in any semiordering. For each node $x \in V(T^M)$, if $N(x) \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x, N(x))$.

Return $\mathcal{R}$.

3.5.5 Correctness of the Set Procedures

In this section we show the correctness of the mm-node set implementation of the set procedures.

Lemma 3.5.6. *Procedure* PARENT($\mathcal{X}$) *is correct.*

Proof. Follows immediately by looking at all different kinds of macro nodes, and by the comments below the implementation of the procedure. □

Lemma 3.5.7. *Procedure* NCA($\mathcal{X}$) *is correct.*

Proof. Let $(x, M(x)) := \mathcal{X}[i]_1$ and $(y, M(y)) := \mathcal{X}[i]_2$. We will show that $v \in S(\mathcal{R})$ if and only if $v \in \mathrm{nca}_T(M(x), M(y))$. We first show $v \in \mathrm{nca}_T(M(x), M(y)) \Rightarrow v \in S(\mathcal{R})$. There must exist $u \in M(x)$ and $w \in M(y)$ such that $v = \mathrm{nca}_T(u, w)$. Consider the cases of Proposition 3.5.4. In case (i), (ii), and (iii) we have $x = y$. This is Case 1 in the procedure. It follows immediately from the implementation that $v \in \mathcal{R}$. Case (iv)-(vi). This is Case 2(a)-(c) in the procedure since the input is semiordered. Case (vii) is taken care of by both case 1 and 2 in the procedure (z is a boundary node).

That $v \in \mathrm{nca}_T(M(x), M(y)) \Leftarrow v \in S(\mathcal{R})$ follows immediately from the implementation and Proposition 3.5.4. □

Lemma 3.5.8. *Procedure* DEEP($\mathcal{X}$) *is correct.*

Proof. The input to DEEP is canonical and semiordered. Let $u \in S(\mathcal{X})$ and $M = S(\mathcal{X}) \cap V(T(u))$. We will show $M = \emptyset$ iff $u \in S(\mathcal{R})$. At some point during the execution of the procedure we have $u \in M(x_i)$.

We first prove $M = \emptyset \Rightarrow u \in S(\mathcal{R})$. Consider the iteration where $u \in M(x_i)$. It is easy to verify that either $u \in S(\mathcal{R})$ after this iteration or $u \in M(x)$. Now assume $u \in M(x)$. It is easy to verify that we have $u \in M(x)$ until $(x, \mathrm{DEEP}_{I(x)}(M(x)))$ is appended to $\mathcal{R}$. Since $M = \emptyset$ we have $u \in \mathrm{DEEP}_{I(x)}(M(x))$ and thus $u \in S(\mathcal{R})$. □

To prove the correctness of procedure MOP we need the following proposition.

Proposition 3.5.9. *Let* $\mathcal{R} = [(r_i, M(r_i)) \mid 1 \le i \le k]$ *and* $\mathcal{S} = [(s_i, M(s_i)) \mid 1 \le i \le l]$ *be deep, canonical lists. For any pair of nodes* $r \in M(r_i)$, $s \in M(s_j)$ *for some* i *and* j, *then* $(r, s) \in \text{mop}_T(S(\mathcal{R}), S(\mathcal{S}))$ *iff one of the following cases are true:*

(i) $r_i = s_j$ *and* $(r, s) \in \text{mop}_{I(r_i)}(M(r_i), M(s_j))$.

(ii) $r_i = l(v, w)$, $s_j = s(v, w)$ *and* $(r, s) \in \text{mop}_{I(r_i, s_j)}(M(r_i), M(s_j))$.

(iii) $r_i = s(v, w)$, $s_j = r(v, w)$ *and* $(r, s) \in \text{mop}_{I(r_i, s_j)}(M(r_i), M(s_j))$.

(iv) $r_i = l(v, w)$, $s_j = r(v, w)$, $r_{i+1} \neq s(v, w)$, $s_{j-1} \neq s(v, w)$, $r = \text{RIGHT}(M(r_i))$, $s = \text{LEFT}(M(s_j))$, *and* $(r_i, s_j) \in \text{mop}_{T^M}(\mathcal{R}|_1, \mathcal{S}|_1)$.

(v) $r_i, s_j \in C \in CS$, $r_i \neq s_j$, *either* r_i *or* s_j *is the bottom boundary node* w *of* C, $r = \text{RIGHT}(M(r_i))$, $s = \text{LEFT}(M(s_j))$, *and* $(r_i, s_j) \in \text{mop}_{T^M}(\mathcal{R}|_1, \mathcal{S}|_1)$.

(vi) $r_i \in C_1 \in CS$, $s_j \in C_2 \in CS$, $C_1 \neq C_2$, $r = \text{RIGHT}(M(r_i))$, $s = \text{LEFT}(M(s_j))$, *and* $(r_i, s_j) \in \text{mop}_{T^M}(\mathcal{R}|_1, \mathcal{S}|_1)$.

The proposition follows immediately, by considering all cases for r_i and s_j, *i.e.*, $r_i = s_j$, r_i and s_j are in the same cluster, and r_i and s_j are not in the same cluster. Using Proposition 3.5.9 we get

Lemma 3.5.10. *Procedure* MOP$(\mathcal{X}, \mathcal{Y})$ *is correct.*

Proof. Let $(x, M(x)) = \mathcal{X}'[i]$ and $(z, M(z)) = \mathcal{Z}[i]$. We call r, t a corresponding pair in $(M(x), M(z))$ iff r and t are the ith node in the left to right order of $M(x)$ and $M(z)$, respectively. Let

$$\begin{aligned} S \;:=\; & \{(r, s) \mid r, t \text{ is a corresponding pair in } (M(x), M(z)), \text{ and} \\ & (t, s) \in \text{mop}_T(S(\mathcal{Z}), S(\mathcal{Y}))\}. \end{aligned}$$

We first show $(v_x, v_y) \in S \Rightarrow (v_x, v_y)$ is a corresponding pair in $(\mathcal{R}[i]_1, \mathcal{R}[i]_2)$. Let (v_z, v_y) be the pair in $\text{mop}_T(S(\mathcal{Z}), S(\mathcal{Y}))$, where $v_z \in M(z_i)$ and $v_y \in M(y_j)$, and look at each of the cases from Proposition 3.5.9.

- Case (i), (ii), and (iii) of the proposition. This is case 2 in the procedure. We have $v_x \in M$ and $v_y \in M_2$, which are both added to $\mathcal{R}$.

- Case (iv), (v), and (vi) of the proposition. This is case 1 in the procedure. Here we set $r := (x, \text{RIGHT}_{I(x)}(M(x)))$ and $s := (y, \text{LEFT}_{I(y)}(M(y)))$, where such $v_x \in M(x)$ and $v_y \in M(y)$. We need to show that (r, s) is added to $\mathcal{R}$ before r and s are changed. If the next case is (i) again then it follows from the fact that $(z_i, y_j) \in \text{mop}_{T^M}(\mathcal{Z}|_1, \mathcal{Y}|_1)$. If the next case is (ii) then we must have $s \lhd y$ or $s = y$ and $\text{LEFTOF}_{I(z)}(M(z), M(y)) = \text{true}$ since $(z_i, y_j) \in \text{mop}_{T^M}(\mathcal{Z}|_1, \mathcal{Y}|_1)$.

We now show if (v_x, v_y) is a corresponding pair in $(\mathcal{R}[i]_1, \mathcal{R}[i]_2)$ then $(v_x, v_y) \in S$. Consider each of the two cases from the procedure. In case 1 we set $r := (x, \text{RIGHT}_{I(x)}(M(x)))$, $s := (y, \text{LEFT}_{I(y)}(M(y)))$ because $z \lhd y$. The pair (r, s) is only added to $\mathcal{R}$ if there is no other $z' \in Z|_1$, $z \lhd z'$ such that $z' \lhd y$, or if $z' = \lhd y$ and there are nodes in $M(y)$ to the left of all nodes in $M(z')$. This corresponds to case (iv), (v), or (vi) in Proposition 3.5.9. In case 2 it is straightforward to verify that it corresponds to one of the cases (i), (ii), or (iii) in Proposition 3.5.9. □

Lemma 3.5.11. *Procedure* FL$(\mathcal{X}, \alpha)$ *is correct.*

Proof. We only need to show that case 1 and 2 correctly computes FL on a micro tree. That the rest of the procedures is correct follows from case (iii) in Proposition 3.5.2 and the comments after the implementation.

That case 1 and 2 are correct follows from Proposition 3.5.2. Since we always call DEEP on the output from FL$(\mathcal{X}, \alpha)$ there is no need to compute FL in the macro tree if N is nonempty. □

Lemma 3.5.12. *Procedure* CANONICAL$(\mathcal{X})$ *is correct.*

Proof. Follows immediately from the implementation of the procedure. □

3.5.6 Complexity of the Tree Inclusion Algorithm

For the running time of the macro-node list implementation observe that, given the data structure described in Section 3.5.2, all set procedures, except FL, perform a single pass over the input using constant time at each step. Procedure FL$(|\mathcal{X}|)$ uses $O(|\mathcal{X}|)$ time to compute $\mathcal{R}$ and U since each step takes constant time. Computing S takes time $O(n_T / \log n_T)$ and computing $\mathcal{S}$ takes time $O(|S|)$. Merging $\mathcal{R}$ and $\mathcal{S}$ takes time linear in the length of the two lists. It follows that FL runs in $O(n_T / \log n_T)$ time. To summarize we have shown that,

Lemma 3.5.13. *For any tree T there is a data structure using $O(n_T)$ space and $O(n_T)$ expected preprocessing time which supports all of the set procedures in $O(n_T / \log n_T)$ time.*

Next consider computing the deep occurrences of P in T using the procedure EMB of Section 3.3 and Lemma 3.5.13. Since each node $v \in V(P)$ contributes at most a constant number of calls to set procedures it follows immediately that,

Theorem 3.5.14. *For trees P and T the tree inclusion problem can be solved in $O(\frac{n_P n_T}{\log n_T})$ time and $O(n_P + n_T)$ space.*

Combining the results in Theorems 3.4.8, 3.5.14 and Corollary 3.4.11 we have the main result of Theorem 3.1.1.

Chapter 4

Union-Find with Deletions

A union-find data structure maintains a collection of disjoint sets under the operations *makeset*, *union* and *find*. Kaplan, Shafrir and Tarjan [SODA 2002] designed data structures for an extension of the union-find problem in which elements of the sets maintained may be deleted. The cost of *delete* in their implementations is the same as the cost of a *find*. They left open the question whether *delete* can be implemented more efficiently than *find*. We resolve this open problem by presenting a relatively simple modification of the classical union-find data structure that supports *delete*, as well as *makeset* and *union*, in *constant* time, while still supporting *find* in $O(\log n)$ worst-case time and $O(\alpha(\lfloor (M+N)/N \rfloor, n))$ amortized time. Here n is the number of elements in the set returned by the *find* operation, and $\alpha(\cdot,\cdot)$ is a functional inverse of Ackermann's function.

4.1 Introduction

A union-find data structure maintains a collection of disjoint sets under the operations *makeset*, *union* and *find*. A *makeset* operation generates a singleton set. A *union* unites two sets into a new set, destroying the original sets. A *find* operation on an element returns an identifier of the set currently containing it.

The union-find problem has many applications in a wide range of areas. For an extensive list of such applications, and for a wealth of information on the problem and many of its variants, see the survey of Galil and Italiano [58].

A simple union-find data structure (attributed to McIlroy and Morris by Aho *et al.* [1]), which employs two simple heuristics, *union by rank* and *path compression*, was shown by Tarjan [114] (see also Tarjan and van Leeuwen [117]) to be very efficient. It performs a sequence of M *find*s and N *makeset* and *union*s in $O(N + M\,\alpha(M,N))$ total time. Here $\alpha(\cdot,\cdot)$ is an extremely slowly growing func-

tional inverse of Ackermann's function. In other words, the *amortized* cost of each *makeset* and *union* is $O(1)$, while the amortized cost of each *find* is $O(\alpha(M+N, N))$, only marginally more than a constant. Fredman and Saks [57] obtained a matching lower bound in the cell probe model of computation, showing that this data structure is essentially optimal in the amortized setting.

The union by rank heuristics on its own implies that *find* operations take $O(\log n)$ worst-case time. Here n is the number of elements in the set returned by the *find* operation. All other operations take constant worst-case time. It is possible to trade a slower *union* for a faster *find*. Smid [109], building on a result of Blum [23], gave for any k a data structure that supports *union* in $O(k)$ time and *find* in $O(\log_k n)$ time. When $k = \log n / \log\log n$, both *union* and *find* take $O(\log n / \log\log n)$ time. Fredman and Saks [57] (see also Ben-Amram and Galil [19]) again show that this tradeoff is optimal, establishing an interesting gap between the amortized and worst-case complexities of the union-find problem. Alstrup *et al.* [3] present union-find algorithms with simultaneously optimal amortized and worst-case bounds.

4.1.1 Local Amortized Bounds

As noted by Kaplan *et al.* [75], the standard amortized bounds for *find* are global in terms of the total number N of elements ever created whereas the worst-case bounds are local in terms of the number n of elements in the set we are finding. Obviously n may be much smaller than N. To state more local amortized bounds, we need a non-standard parameterization of the inverse Ackermann function. For integers $k \geq 0$ and $j \geq 1$, define Ackermann's function $A_k(j)$ as follows

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1. \end{cases}$$

Here $f^{(i)}(x)$ is the function f iterated i times on x. In the standard definition of the inverse Ackermann function we have $\alpha(i,j) = \min\{k \mid A_k(\lfloor i/j \rfloor) > i\}$ for integers $i, j \geq 0$ [115]. To state more local amortized bounds we will use the following definition of the inverse Ackermann function

$$\bar{\alpha}(i,j) = \min\{k \geq 2 \mid A_k(i) > j\} ,$$

for integers $i, j \geq 0$. (For technical reasons, $\bar{\alpha}(i,j)$ is defined to be at least 2 for every $i, j \geq 0$.) Relating to the standard definition of α, we have $\alpha(M,N) = \Theta(\bar{\alpha}(\lfloor M/N \rfloor, N))$.

Kaplan *et al.* [75] present a refined analysis of the classical union-find data structure showing that the amortized cost of *find* is only $O(\bar{\alpha}(\lfloor(M+N)/N\rfloor, n)$. They state their results equivalently in terms of a three parameter function that we will not define here. To get a purely local amortized cost for *find*, we note that

$$\bar{\alpha}(\lfloor(M+N)/N\rfloor, n) \;\leq\; \bar{\alpha}(1,n) \;=\; O(\alpha(n,n))\;.$$

4.1.2 Union-Find with Deletions

In the traditional version of the union-find problem elements are created using *makeset*. Once created, however, elements are never destroyed. Kaplan *et al.* [75] consider a very natural extension of the union-find problem in which elements may be *deleted*. We refer to this problem as the *union-find with deletions* problem, or *union-find-delete* for short.

Let N^* be the current number of elements in the whole data structure. Using relatively straightforward ideas (see, e.g., [75]) it is possible to design a union-find-delete data structure that uses only $O(N^*)$ space, handles *makeset*, *union* and *delete* in $O(1)$ worst-case time, and *find* in $O(\log N^*)$ worst-case time and $O(\alpha(N^*))$ amortized time. The challenge in the design of union-find-delete data structures is to have an efficient *find*(x) in terms of n, the size of the set currently containing x, and not N^*.

Using an incremental background rebuilding technique for each set, Kaplan *et al.* [75] describe a way of converting any data structure for the classical union-find problem into a union-find-delete data structure. The time bounds for *makeset*, *find* and *union* change by only a constant factor, while the time needed for *delete*(x) is the same as the time needed for *find*(x) followed by *union* with a singleton set. As *union* is usually much cheaper than *find*, Kaplan *et al.* thus show that in both the amortized and the worst-case settings, *delete* is not more expensive than *find*. Combined with their refined amortized analysis of the classic union-find data structure, this provides, in particular, a union-find-delete data structure that implements *makeset* and *union* in $O(1)$ time, and *find* and *delete* in $O(\bar{\alpha}(\lfloor(M+N)/N\rfloor, n)$ amortized time and $O(\log n)$ worst-case time. They leave open, however, the question whether *delete* can be implemented *faster* than *find*.

4.1.3 Our Results

We solve the open problem and show that *delete* can be performed in *constant* worst-case time, while still keeping the $O(\bar{\alpha}(\lfloor(M+N)/N\rfloor, n)) = O(\alpha(n,n))$ amortized cost and the $O(\log n)$ worst-case cost of *find*, and the constant worst-case cost

of *makeset* and *union*. We recall here that N is the total number of elements ever created, M is the total number of *find*s performed, and n is the number of elements in the set returned by the *find*. The data structure that we present uses linear space and is a relatively simple modification of the classic union-find data structure. It is at least as simple as the data structures presented by Kaplan *et al.*[75].

As a by-product we obtain a concise potential-based proof of the $O(\bar{\alpha}(\lfloor (M + N)/N \rfloor, n))$ bound on the amortized cost of a *find* in the classical setting. We believe that our potential-based analysis is simpler than the one given by Kaplan *et al.*

4.1.4 Our Techniques

Our union-find-delete data structure, like most other union-find data structures, maintains the elements of each set in a rooted tree. As elements can now be deleted, not all the nodes in these trees contain elements. Nodes that contain elements are said to be *occupied*, while nodes that do not contain elements are said to be *vacant*. When an element is deleted, the node containing it becomes vacant. If proper measures are not taken, a tree representing a set may contain too many vacant nodes. As a result, the space needed to store the tree, and the time needed to process a *find* may become too large. Our data structure uses a simple collection of local operations to *tidy up* a tree after each *delete*. This ensures that at most half of the nodes in a tree are vacant. More importantly, the algorithm employs local constant-time *shortcut* operations in which the grandparent, or a more distant ancestor, of a node becomes its new parent. These operations, which may be viewed as a local constant-time variant of the path compression technique, keep the trees relatively shallow to allow a fast *find*.

As with the simple standard union-find, the analysis is the most non-trivial part. The analysis of the new data structure uses two different potential functions. The first potential function is used to bound the *worst-case* cost of *find*. Both potential functions are needed to bound the *amortized* cost of *find*. The second potential function on its own can be used to obtain a simple derivation of the refined amortized bounds of Kaplan *et al.* [75] for union-find without deletions.

We end this section with a short discussion of the different techniques used to analyze union-find data structures. The first tight amortized analysis of the classical union-find data structure, by Tarjan [114] and Tarjan and van Leeuwen [117], uses *collections of partitions* and the so-called *accounting method*. The refined analysis of Kaplan *et al.* [75] is directly based on this method.

A much more concise analysis of the union-find data structure based on po-

tential functions can be found in Kozen [85] and Chapter 21 of Cormen *et al.* [39]. The amortized analysis of our new union-find-delete data structure is based on small but crucial modifications of the potential function used in this analysis. As a by product we get, as mentioned above, a simple proof of the local amortized bounds of Kaplan *et al.* [75].

Seidel and Sharir [107] recently presented an intriguing top-down amortized analysis of the union-find data structure. Our analysis is no less concise, though perhaps less intuitive, and has the additional advantage of bounding the cost of an amortized operation in terms of the size of the set returned by the operation.

4.2 Preliminaries

4.2.1 The Union-Find and Union-Find-Delete Problems

A classical union-find data structure supports the following operations:

- *makeset(x)*: Create a singleton set containing x.
- *union(A,B)*: Combine the sets identified by A and B into a new set, destroying the old sets.
- *find(x)*: Return an identifier of the set containing x.

The only requirement from the identifier returned by a *find* is that the calls *find*(x) and *find*(y) return the same identifier if and only if the two elements x and y are currently contained in the same set. A union-find-delete data structure supports, in addition to the above operations, a *delete* operation

- *delete(x)*: Delete x from the set containing it.

A *delete* should not change the identifier attached to the set from which the element was deleted. It is important to note that *delete* does not receive a reference to the set currently containing x. It only receives a pointer to the element x itself.

Definitions

Let T be a rooted tree. The root of T is denoted by $\mathrm{root}(T)$. The set of all nodes in T is denoted $V(T)$, and the *size* of T is $|V(T)|$. The *height* of a node v, denoted by $h(v)$, is defined to be 0, if v is a leaf, and $\max\{h(w)|w$ is a child of $v\}+1$, otherwise. The height of a tree is the height of its root. For a node v let $p(v)$ denote the *parent* of v. A node $x \in V(T)$ is an *ancestor* of a node $y \in V(T)$ if x is on the path from y

to the root of T—both y and the root included. Node x is a *descendant* of node y if y is an ancestor of x.

4.2.2 Standard Worst-Case Bounds for Union-Find

We briefly review here the simple standard union-find data structure that supports *makeset* and *union* in constant time and *find* in $O(\log n)$ time. It forms the basis of our new data structure for the union-find-delete problem.

The elements of a set are maintained in a rooted tree and the identifier of the set is the root of the tree. When we refer to set A, A denotes the identifer of the set, i.e., the root of the tree representing it. We will use T_A to denote the tree representing the set A. Each node $v \in V(T_A)$ has an assigned integer rank $\textit{rank}(v)$. An important invariant is that the parent of a node always has strictly higher rank than the node itself. The rank of a tree is defined to be the rank of the root of the tree.

Operations We implement the operations as follows:

find(x): Follow parent pointers from x all the way to the root. Return the root as the identifier of the set.

makeset(x): Create a new node x. Let $p(x) \leftarrow x$, $\textit{rank}(x) \leftarrow 0$.

union(A,B): Recall that A and B are root nodes. If $\textit{rank}(A) \geq \textit{rank}(B)$ make B a child of A. Otherwise make A a child of B. If $\textit{rank}(A)=\textit{rank}(B)$, increase $\textit{rank}(A)$ by one.

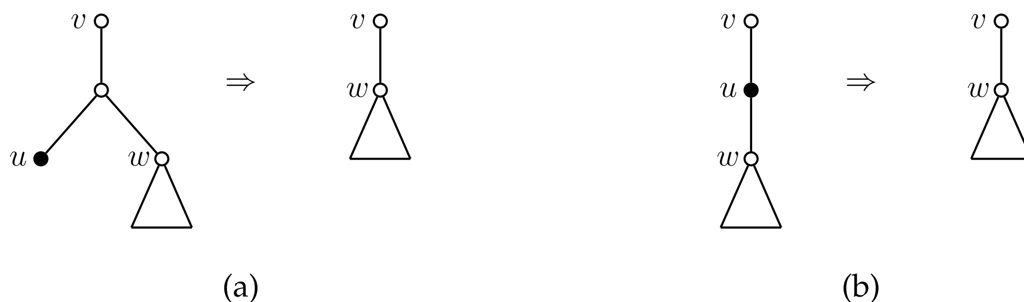
Analysis Trivially, *makeset* and *union* take constant time. Since ranks are strictly increasing when following parent pointers, the time of *find* applied to an element in a set A is proportional to $\textit{rank}(A)$. We prove, by induction, that $\textit{rank}(A) \leq \log_2 |A|$, or equivalently, that

$$|A| \geq 2^{\textit{rank}(A)} \ . \tag{4.1}$$

When A is created with *makeset(x)*, it has rank 0 and $2^0 = 1$ elements. If C is the set created by *union(A,B)*, then $|C| = |A| + |B|$. If C has the same rank as A, or the same rank as B, we are trivially done. Otherwise, we have $\textit{rank}(A) = \textit{rank}(B) = k$ and $\textit{rank}(C) = k + 1$, and then

$$|C| = |A| + |B| \geq 2^k + 2^k = 2^{k+1} \ .$$

This completes the standard analysis of union-find with worst-case bounds.



(a) (b)

Figure 4.1: Deletion of u and tidying up the tree. Black nodes are occupied. In (a) the parent of u is vacant, and thus bypassed when u is deleted. In (b) u gets bypassed after the deletion since it has a single child.

4.3 Augmenting Worst-Case Union-Find with Deletions

Each set in the data structure is again maintained in a rooted tree. In the standard union-find data structure, reviewed in Section 4.2.2, the nodes of each tree were identified with the elements of the set. In the new data structure, elements are attached to nodes, not identified with them. Some nodes in a tree are *occupied*, i.e., have an element attached to them, while others are *vacant*, i.e., have no element attached to them. An element can then be deleted by simply removing it from the node it was attached to. This node then becomes vacant. The identifier of a set is taken to be its root node. As the identifier of a set is a node, and not an element, identifiers do not change as a result of *delete*.

A problem with this approach is that if we never remove vacant nodes from the trees, we may end up consuming non-linear space. To avoid this, we require our union-find trees to be *tidy*:

Definition 4.3.1. A tree is said to be *tidy* if it satisfies the following properties:

- Every vacant non-root node has at least two children,
- Every leaf is occupied and has rank 0.

It is easy to tidy up a tree (see Figure 4.1). First, we remove vacant leaves. When a node becomes a leaf, its rank is reduced to 0. Next, if a vacant non-root node v has a single child w, we make the parent of v the parent of w and remove v. We call this *bypassing* v. The following lemma follows from the definition of a tidy tree.

Lemma 4.3.2. *At most half of the nodes in a tidy tree may be vacant.*

Proof. Since every vacant non-root node has at least two children, the number of leaves in the tree is at least the number of vacant nodes. Since all leaves are occupied, at most half the nodes in a tidy tree can be vacant. □

Tidy trees thus use linear space. However, tidyness on it own does not yield a sublinear time bound on *find*. (Note, for example, that a path of occupied nodes is tidy.) Our next goal is to make sure that the height of a tree is logarithmic in the number of occupied nodes contained in it. Ideally, we would want all trees to be *reduced*:

Definition 4.3.3. A tree is said to be *reduced* if it is either

- A tree composed of a single occupied node of rank 0, or
- A tree of height 1 with a root of rank 1 and occupied leaves of rank 0.

We will not manage to keep our trees reduced at all times. Reduced trees form, however, the base case for our analysis.

4.3.1 Keeping the Trees Shallow during Deletions

This section contains our main technical contribution. We show how to implement deletions so that for any set A,

$$|A| \geq (2/3)(6/5)^{\mathit{rank}(A)} . \tag{4.2}$$

Consequently, $\mathit{rank}(A) \leq \log_{6/5}(3|A|/2) = O(\log|A| + 1)$. As the rank of a tree is always an upper bound on its height, we thus need to follow at most $O(\log|A|+1)$ parent pointers to get from any element of A to the root identifier.

The key idea is to associate the following value with each node v:

Definition 4.3.4. The value $val(v)$ of a node v is defined as

$$val(v) = \begin{cases} (5/3)^{\mathit{rank}(p(v))} & \text{if } v \text{ is occupied,} \\ (1/2)(5/3)^{\mathit{rank}(p(v))} & \text{if } v \text{ is vacant.} \end{cases}$$

Here, if v is a root, $p(v) = v$. The value of a set A is defined as the sum the values of all nodes in the tree T_A representing A:

$$\mathit{VAL}(A) = \sum_{v \in T_A} val(v) .$$

The value $5/3$ in the definition of $val(v)$ is chosen to satisfy Equation 4.2, Lemma 4.3.5, Lemma 4.3.7, and Lemma 4.4.6 below. In fact, we could have chosen any constant value in $[(1+\sqrt{5})/2, 2)$.

We are going to implement deletions in such that we maintain the following invariant.

$$VAL(A) \geq 2^{rank(A)} \ . \tag{4.3}$$

Since the tree representing a set A contains exactly $|A|$ occupied nodes, each of value at most $(5/3)^{rank(A)}$, and at most $|A|$ vacant nodes in T_A, each of value at most $(5/3)^{rank(A)}/2$, it will follow that

$$|A| \geq \frac{2^{rank(A)}}{(3/2)(5/3)^{rank(A)}} = (2/3)(6/5)^{rank(A)},$$

so (4.3) will imply (4.2).

The essential operation used to maintain inequality (4.3)—and thus keep the trees shallow—is to *shortcut* from a node v, giving v a parent higher up over v in the tree. For example, path compression shortcuts from all nodes in a search path directly to the root. Since ranks are strictly increasing up through the tree, shortcutting from v increases the value of v by a factor of at least $5/3$. This suggests that we can make up for the loss of a deleted node by a constant number of shortcuts from nearby nodes of similar rank.

Before proceeding, let us check that reduced trees satisfy (4.3).

Lemma 4.3.5. *If the tree representing a set A is reduced then $VAL(A) \geq 2^{rank(A)}$.*

Proof. If A is of height 0, then $VAL(A) = (5/3)^0 = 1$ and $2^{rank(A)} = 1$. If A is of height 1, then $VAL(A) \geq (5/3)^1 + (1/2)(5/3)^1 = 5/2$ while $2^{rank(A)} = 2$. □

Let us for a moment assume that we have an implementation of *delete* that preserves, i.e., does not decrease, value, and let us check that the other operations preserve (4.3). A *makeset* creates a reduced tree, so (4.3) is satisfied by Lemma 4.3.5. Also, when we set $C := union(A, B)$, we get $VAL(C) \geq VAL(A) + VAL(B)$, and hence (4.3) follows just like (4.1).

4.3.2 Paying for a Deletion via Local Rebuilding

We now show how we can implement *delete* in constant time, while maintaining (4.3). We will implement *delete* such that either the value of the set from which the element is deleted is not decreased, or otherwise we end up with a reduced tree representing the set.

Suppose we delete an element of A attached to a node u. As u becomes vacant, we immediately loose half its value. Before u was vacant the tree was tidy, but now we may have to tidy the tree (see also Figure 4.1). If u is not a leaf, the only required tidying up is to bypass u if it has a single child. If instead u was a leaf, we first delete u. If $p(u)$ is now a leaf, its rank is reduced to zero, but that in itself does not affect any value. If $p(u)$ is vacant and now has only one child, we bypass $p(u)$. This completes the tidying up.

Lemma 4.3.6. *Let v be the parent of the highest node affected by a* delete(x)*, including tidying up. If rank$(v) = k$, then VAL(A) is decreased by at most $(9/10)(5/3)^k$, where A is the set containing x.*

Proof. It is easy to see that the worst-case is when $v = p(p(u))$, where u is a deleted leaf and $p(u)$ is bypassed. Now u lost at most $(5/3)^{k-1}$ and $p(u)$ lost $(5/3)^k/2$, while the other child of $p(u)$ gained at least $((5/3)^k - (5/3)^{k-1})/2$ from the bypass. Adding up, the total loss is $(9/10)(5/3)^k$. □

Below we show how to regain the loss from a *delete* using a pointer to the node v from Lemma 4.3.6. To find nearby nodes to shortcut from, we maintain two doubly linked lists for each node v; namely $C(v)$ containing the children of v, and $G(v)$ containing the children of v that themselves have children. Thus, to find a grandchild of v, we take a child of a child in $G(v)$. Both lists are easily maintained as children are added and deleted: if a child u is added to v, it is added to $C(v)$. If u is the first child of v, we add v to $G(p(v))$. Finally, we add u to $G(v)$ if $C(u)$ is non-empty. Deleting a child is symmetric. Using these lists, we implement a procedure to gain the lost value as follows:

PROCEDURE GAIN(v) Set $x := v$. Repeat the following until the value of the set containing v is increased by $(9/10)(5/3)^{rank(v)}$.

1. While $G(x)$ is non-empty do: Find a child y of x that have children. There are two main cases.
   (a) y is occupied.
   Take any child z of y and shortcut to x (see Figure 4.2 (a)). If z is the last child of y, remove y from $G(x)$.
   (b) y is vacant.
   First note that since the tree is tidy, $|C(y)| \geq 2$. There are two cases.
      i. If $|C(y)| > 2$, take any child z of y and shortcut to x as above (see Figure 4.2 (b)).

ii. $C(y) = \{z, z'\}$. If both z and z' are occupied, shortcut both z and z' to x and remove y (see Figure 4.2 (c)).
Otherwise, one of them, say z is vacant. Tidyness implies that z has at least two children. If more than two, any one of them can be shortcut to x (see Figure 4.2 (d)). If exactly two, then one of them is shortcut to y and the other to x while z is removed (see Figure 4.2 (e)).

2. If x is not the root, set $x = p(x)$.
If x is the root, set $rank(x) = h(x)$ and stop.

It is easy to verify that the procedure preserves tidyness. The following lemma shows that procedure GAIN either regains the value lost due to a deletion or returns a reduced tree in constant time.

Lemma 4.3.7. *Let v be a node in a tidy tree. Then in $O(1)$ time procedure* GAIN(v) *either increases the value of the tree with $(9/10)(5/3)^{rank(v)}$ or returns a reduced tree.*

Proof. If we stop in case 2 because x is the root, then the tree is a reduced tree. In the other case we stop because we have gained enough value. We now show that one of these cases will happen within constant time. First note that we cannot get to case 2 twice in a row without getting to case 1 since $p(x) \in G(p(p(x)))$. Clearly, one iteration of case 1 takes constant time. It remains to show that one iteration of case 1 increases the value with $\Omega((5/3)^{rank}(v))$. Let $k = rank(v)$. Consider each of the cases in case one:

- Case 1a. The value of z increases by at least

$$(1/2)((5/3)^k - (5/3)^{k-1}) = (1/5)(5/3)^k .$$

  We note that y may have rank much lower than $k - 1$, but that would only increase our gain.

- Case 1(b)i. The gain is the same as in Case 1a.

- Case 1(b)ii. There are three cases:

  In the first case (both z and z' occupied), we get a gain of at least

$$2((5/3)^k - (5/3)^{k-1}) - (1/2)(5/3)^k = (3/10)(5/3)^k .$$

  In the second case (z is vacant and has more than two children) we gain at least

$$(1/2)((5/3)^k - (5/3)^{k-2}) = (8/25)(5/3)^k .$$

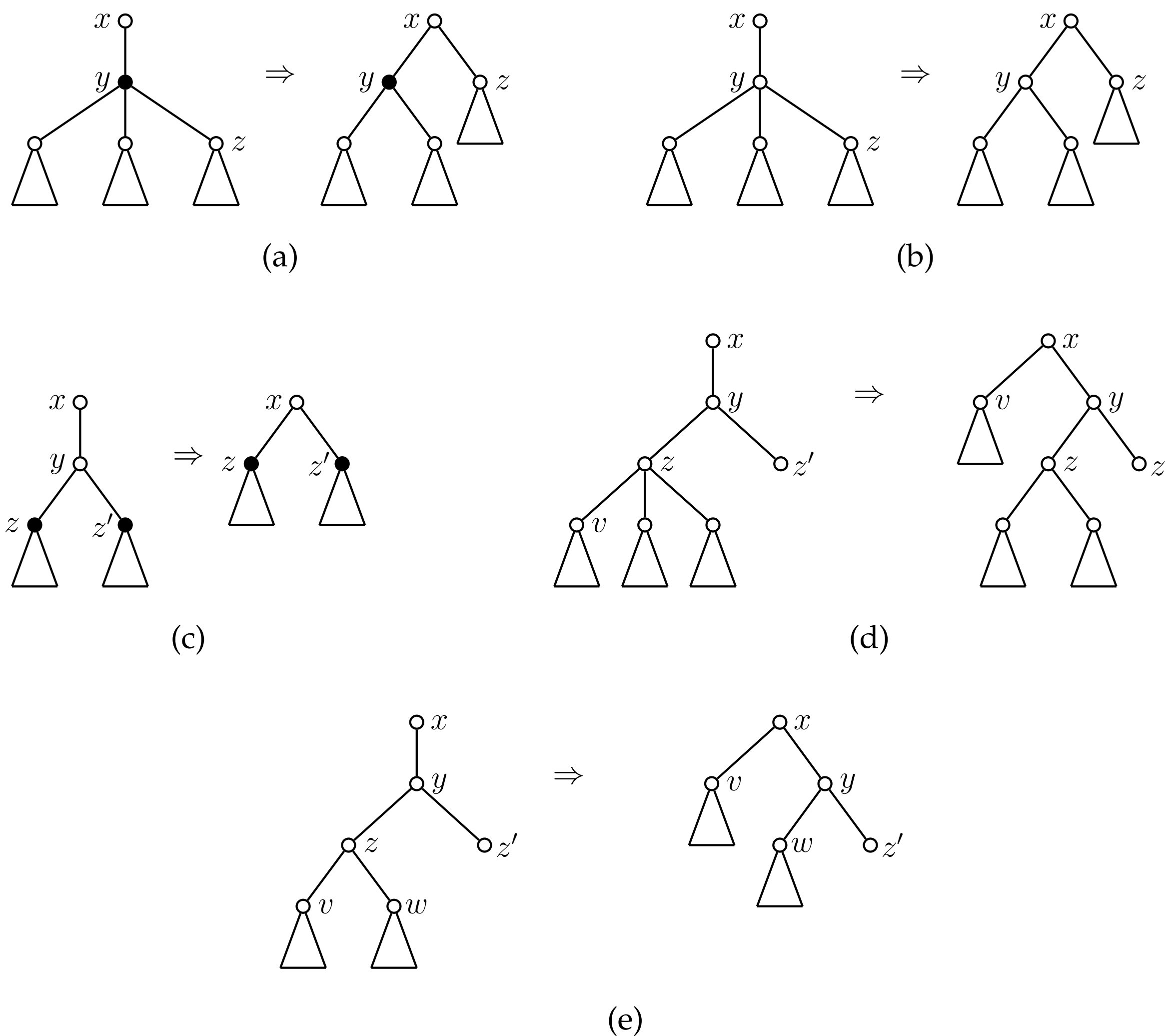


Figure 4.2: The figure shows the cases from procedure GAIN. In (a) y is occupied. In (b) y is vacant and has more than two children. In (c),(d), and (e) y is vacant and has exactly two children. In (c) the children of y are occupied. In (d) and (e) not both children of y are occupied. In (d) z has more than two children and in (e) it has exactly two children.

In the second case (z is vacant and has exactly two children) we gain at least

$$(1/2)((5/3)^k + 2(5/3)^{k-2}) = (7/50)(5/3)^k \ .$$

Since we gain at least $(7/50)(5/3)^k$ value at each iteration, we need at most 7 iterations of case 1. □

Combining Lemmas 4.3.5, 4.3.6, and 4.3.7 we implement a deletion in constant time so that either we have no loss, meaning that (4.3) is preserved, or obtaining a reduced tree that satisfies (4.3) directly. Thus we have proved

Theorem 4.3.8. *There is a data structure for the union-find-delete problem supporting* makeset, union, *and* delete *in constant time, and* find *in $O(\log n)$ time. The size of the data structure is proportional to the number of current elements in it.*

4.4 Faster Amortized Bounds

We will now show that we can get fast amortized bounds for *find*, yet preserve the previous worst-case bounds. All we have to do is to use *path compression* followed by tidying up operations. Path compression of a path from node $v \in T$ to node $u \in T$ makes every node on the path a child of u. When we perform a *find* from a node v, we compress the path to the root. Our analysis is a new potential based analysis that obtains local amortized bounds.

Before going further, we note that path compression consists of shortcuts that increase value of the previous section, so intuitively, the path compression can only help the deletions. Below, we first present our new analysis without the deletions, and then we observe that deletions are only helpful.

Analysis

We assign a potential $\phi(x)$ to each node x in the forest. To define the potential we need some extra functions. Define $Q = \lfloor \frac{M+N}{N} \rfloor$ and $\alpha'(n) = \bar{\alpha}(Q, n)$. Our goal is to prove that the amortized cost of *find* is $O(\alpha'(n))$ where n is the cardinality of the set found. We also define $\mathit{rank}'(v) = \mathit{rank}(v) + Q$.

Definition 4.4.1. For a non-root node x we define

$$\text{level}(x) = \max\{k \geq 0 \mid A_k(\mathit{rank}'(x)) \leq \mathit{rank}'(p(x))\} \ ,$$

and

$$\text{index}(x) = \max\{i \geq 1 \mid A^{(i)}_{\text{level}(x)}(\mathit{rank}'(x)) \leq \mathit{rank}'(p(x))\} \ .$$

We have

$$0 \leq \text{level}(x) < \bar{\alpha}(\mathit{rank}'(x), \mathit{rank}'(p(x))) \leq \alpha'(\mathit{rank}'(p(x))) \;, \tag{4.4}$$

and

$$1 \leq \text{index}(x) \leq \mathit{rank}'(x) \;. \tag{4.5}$$

Definition 4.4.2. The potential $\phi(x)$ of a node x is defined as

$$\phi(x) = \begin{cases} \alpha'(\mathit{rank}'(x)) \cdot (\mathit{rank}'(x) + 1) & \text{if } x \text{ root,} \\ (\alpha'(\mathit{rank}'(x)) - \text{level}(x)) \cdot \mathit{rank}'(x) - \text{index}(x) + 1 & \\ \qquad \text{if } x \text{ not root and } \alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}'(p(x))), & \\ 0 & \text{otherwise.} \end{cases}$$

The potential $\Phi(x)$ of a set A is defined as the sum of the potentials of the nodes in the tree T_A representing the set A:

$$\Phi(A) = \sum_{x \in T_A} \phi(x) \;.$$

At first sight the potential function looks very similar to the standard one from [39], but there are important differences. Using $\alpha(\mathit{rank}(x))$ instead of $\alpha(N)$ we get a potential function that is more locally sensitive. To get this change to work, we use the trick that the potential of a node is only positive if $\alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}'(p(x)))$. Note that from Equation (4.4) and Equation (4.5) it follows that the potential of such a node is strictly positive. We also note that the only potentials that can increase are those of roots. All other nodes keep their ranks while the ranks of their parents increase and that can only decrease the potential.

We will now analyze the change in potential due to the operations.

Lemma 4.4.3. *The total increase in potential due to* makeset *operations is $O(M+N)$.*

Proof. When we create a new set A with rank 0, it gets potential

$$\alpha'(Q)(Q+1) = \bar{\alpha}(Q,Q)(Q+1) = 2(Q+1) = O((M+N)/N) \;.$$

Over N *makeset* operations, this adds up to a total increase of $O(M+N)$. □

Lemma 4.4.4. *The operation* union(A, B) *increases the potential by at most one.*

Proof. Suppose we make A the parent of B. If the rank of A is not increased, there is no node that increases potential, so assume that $\mathit{rank}'(A)$ is increased from k to $k+1$. Then k was also the rank of B. If $\alpha'(k+1) > \alpha'(k)$, then B gets zero

potential along with any previous child of A. The potential of B is reduced by $\alpha'(k) \cdot (k+1)$. On the other hand, the potential of A is increased by

$$(\alpha'(k)+1) \cdot (k+2) - \alpha'(k) \cdot (k+1) = \alpha'(k) + k + 2 \ ,$$

which is less than $\alpha'(k) \cdot (k+1)$ if $k \geq 2$, since $\alpha'(k) \geq 2$. (Here we use the fact that $\bar{\alpha}(j,i) \geq 2$, for every $i,j \geq 0$.) If $k=1$ the increase in potential is one.

Finally, if $\alpha'(k+1) = \alpha'(k)$, then the potential of A increases by $\alpha'(k)$ while the potential of B decreases by at least $\alpha'(k)$, since B was a root with potential $\alpha'(k) \cdot (k+1)$ and now becomes a child with potential at most $\alpha'(k) \cdot k$. □

Lemma 4.4.5. *A path compression of length ℓ from a node v up to some node u decreases the potential by at least $\ell - (2 \cdot \alpha'(\mathit{rank}'(u)) + 1)$. In particular, the amortized cost is at most $O(\alpha'(\mathit{rank}'(u)))$.*

Proof. The potential of the root does not change due to the path compression. We will show that at least $\max\{0, l - (2 \cdot \alpha'(\mathit{rank}'(u)) + 2)\}$ nodes have their potential decreased by at least one.

There can be at most $\alpha'(\mathit{rank}'(u))$ nodes x on the compressed path that had $\alpha'(\mathit{rank}'(x)) < \alpha'(\mathit{rank}'(p(x)))$ before the operation. The potentials of these nodes do not change.

If node x had $\alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}'(p(x))) < \alpha'(\mathit{rank}'(u))$, then its potential drops to 0, and the decrease in x's potential is therefore at least one.

It remains to account for the nodes x with $\alpha'(\mathit{rank}'(x)) = \alpha'(\mathit{rank}(u))$. Let x be a node on the path such that x is followed somewhere on the path by a node $y \neq u$ with level(y) = level$(x) = k$. There can be at most $\alpha'(\mathit{rank}'(u)) + 1$ nodes on the path that do not satisfy these constraints: The last node before u, u, and the last node on the path for each level, since level$(y) < \alpha'(\mathit{rank}'(u))$. Let x be a node that satisfies the conditions. We show that the potential of x decreases by at least one. Before the path compression we have

$$\begin{aligned} \mathit{rank}'(p(y)) &\geq A_k(\mathit{rank}'(y)) \\ &\geq A_k(\mathit{rank}'(p(x))) \\ &\geq A_k(A_k^{(\text{index}(x))}(\mathit{rank}'(x))) \\ &= A_k^{(\text{index}(x)+1)}(\mathit{rank}'(x)). \end{aligned}$$

After the path compression $\mathit{rank}'(p(x)) = \mathit{rank}'(p(y))$ and thus $\mathit{rank}'((p(x)) \geq A_k^{(\text{index}(x)+1)}(\mathit{rank}'(x))$, since $\mathit{rank}'(x)$ does not change and $\mathit{rank}'(p(y))$ does not decrease. This means that either index(x) or level(x) must increase by at least one. Thus $\phi(x)$ decreases by at least one. □

We conclude that the amortized cost of *find* in a set A is

$$O(\alpha'(\mathit{rank}'(A))) = O(\bar{\alpha}(Q, \mathit{rank}(A) + Q + c)) = O(\bar{\alpha}(Q, \mathit{rank}(A))).$$

The last step follows because $\bar{\alpha}$ is defined to be at least 2. Recall that $Q = \lfloor \frac{M+N}{N} \rfloor$ and that $\mathit{rank}(A) \leq \log_2 |A|$, so without deletions, this is the desired bound.

A simpler analysis To get a simpler potential based proof of the local amortized bound we can replace $\alpha'(n)$ by $\hat{\alpha}(n) = \alpha(n, n)$ and use $\mathit{rank}(x)$ instead of $\mathit{rank}'(x)$ in the potential function. This gives a proof more in style with the one in Cormen *et al.* [39], but still obtaining the local amortized bound $\hat{\alpha}(n)$ for *find* instead of $\hat{\alpha}(N)$. With this definition of the potential function *makeset* no longer increases the potential. A *union* might now increase the potential by 2, and Lemma 4.4.5 still hold.

4.4.1 Deletion and Path Compression

We now combine the path compression and amortized analysis with deletions. The potential used in the amortization is identical for vacant and occupied nodes. It is clear that deletions and tidying up can only decrease this potential, so they have no extra amortized cost. Likewise, a path compression can only increase value as it only performs shortcuts. However, after a path compression, there may be some cleaning to do if some vacant nodes go down to 0 or 1 children. We start the path compression from a tidy tree where each vacant node has at least two children, and the compression takes at most one child from each node on the path. Hence the only relevant tidying up is to bypass some of the nodes on the path. The tidying up takes time proportional to the length of the path, so the cost of a *find* is unchanged.

The tidying up does decrease value, but the loss turns out less than the gain from the compression.

Lemma 4.4.6. *Path compression followed by tidying up operations does not decrease the value of a tree.*

Proof. The path compression involves nodes $v_0, ..., v_\ell$ starting in some occupied node v_0 and ending in the root which has some rank k. After the compression, all nodes $v_0, .., v_{\ell-1}$ are children of the root v_ℓ. If node v_i is not bypassed when tidying up, its value gain is at least $((5/3)^{\mathit{rank}(v_\ell)} - (5/3)^{\mathit{rank}(v_{i+1})})/2$. If v_i is bypassed, then $0 < i < \ell$, and v_i is vacant, so the loss is $(5/3)^{\mathit{rank}(v_{i+1})}/2$. However, then v_i has a child w_i which gains at least $((5/3)^{\mathit{rank}(v_\ell)} - (5/3)^{\mathit{rank}(v_i)})/2$, so the total change is

$$((5/3)^{\mathit{rank}(v_\ell)} - (5/3)^{\mathit{rank}(v_{i+1})} - (5/3)^{\mathit{rank}(v_i)})/2 \ .$$

Since ranks are strictly increasing along a path, this change is positive for all but $i = \ell - 1$. On the other hand, the first node v_0 is always occupied, and has a gain of at least $(5/3)^{rank(v_\ell)} - (5/3)^{rank(v_1)}$, where $1 \leq \ell - 1$. We can use the value gained by v_0 to pay for the value lost by bypassing both v_1 and v_{l-1}. There are two cases.

If both v_{l-1} and v_1 is bypassed we must have $l \geq 4$. Combining the changes in potential for the nodes v_0, v_1, and v_{l-1} we get,

$$(5/3)^{rank(l)} - (5/3)^{rank(v_1)} - (1/2)(5/3)^{rank(v_2)} - (1/2)(5/3)^{rank(v_{l-1})} > 0 \ .$$

If v_1 is not bypassed, we get that the total gain for v_0 and v_{l-1} is at least,

$$(5/3)^{rank(v_l)} - (5/3)^{rank(v_1)} - (1/2)(5/3)^{rank(v_l)} \ ,$$

which is always positive. Thus the overall change in value is positive, or zero if the path has length 0 or 1 and no compression happens. □

Since our values and hence (4.3) are preserved, for any set A, we get $rank(A) = O(\log |A|)$. Thus our amortized cost of a *find* is

$$O(\bar{\alpha}(Q, O(\log |A|)) = \Theta(\bar{\alpha}(\lfloor \frac{M+N}{N} \rfloor, |A|) \ .$$

Note that our replacing $O(\log |A|)$ by $|A|$ in the second argument of $\bar{\alpha}$ has no asymptotic consequence. Summing up, we have proved

Theorem 4.4.7. *A sequence of M* finds *and at most N* makeset, union, *and* deletes *takes $O(N + \sum_{i=1}^{M} \bar{\alpha}(\lfloor \frac{M+N}{N} \rfloor, n_i)$ time, where n_i is the number of elements in the set returned by the ith find. Meanwhile, the worst-case bounds of Theorem 4.3.8 are preserved. The size of data structure at any time during the sequence is proportional to the number of current elements in it.*

Part II

Approximation Algorithms

Chapter 5

Introduction to Part II

The results in this part of the dissertation are concerning approximation algorithms on graphs. Two problems are studied in this part: The *asymmetric k-center problem* and its variants (Chapter 6) and the *finite capacity dial-a-ride problem* (Chapter 7).

5.1 Overview

In this section we will give a short overview of the problems studied in this part of the dissertation.

Asymmetry in k-Center Variants Given a complete graph on n vertices with nonnegative (but possibly infinite) edge costs, and a positive integer k, the *k-center problem* is to find a set of k vertices, called *centers*, minimizing the maximum distance to any vertex and from its nearest center. In Chapter 6 we examine variants of the *asymmetric* k-center problem.

We provide an $O(\log^* n)$-approximation algorithm for the asymmetric *weighted* k-center problem. Here, the vertices have weights and we are given a total budget for opening centers. In the *p-neighbor* variant each vertex must have p (unweighted) centers nearby: we give an $O(\log^* k)$-bicriteria algorithm using $2k$ centers, for small p. In k-center with minimum coverage, each center is required to serve a minimum of clients. We give an $O(\log^* n)$-approximation algorithm for this problem. Finally, the following three versions of the asymmetric k-center problem we show to be inapproximable: *priority k-center*, *k-supplier*, and *outliers with forbidden centers*.

Finite Capacity Dial-a-Ride Given a collection of objects in a metric space, a specified destination point for each object, and a vehicle with a capacity of at most k objects, the *finite capacity dial-a-ride problem* is to compute a shortest tour for the vehicle in which all objects can be delivered from their sources to their destinations while ensuring that the vehicle carries at most k objects at any point in time. In the *preemptive* version of the problem an object may be dropped at intermediate locations and then picked up later by the vehicle and delivered.

We study the hardness of approximation of the preemptive finite capacity dial-a-ride problem. Let N denote the number of nodes in the input graph, i.e., the number of points that are either sources or destinations. We show that the preemptive Finite Capacity Dial-a-Ride problem has no $\min\{O(\log^{1/4-\varepsilon} N), k^{1-\varepsilon}\}$-approximation algorithm for any $\varepsilon > 0$ unless all problems in NP can be solved by randomized algorithms with expected running time $O(n^{\text{polylog} n})$.

5.2 Approximation Algorithms

Many interesting optimization problems turns out to be NP-hard. This, in the words of Garey and Johnson, means "I can't find an efficient algorithm, but neither can all these famous people" ([59, p. 3]).

Although the optimal solution to NP-hard optimization problems cannot be found efficiently (unless P = NP), it might still be possible to find near-optimal solutions efficiently. The goal in approximation algorithms is to find provably good approximate solutions for optimization problems that are hard to solve exactly.

Given an instance of a minimization problem, an δ-approximation algorithm returns a feasible solution whose objective value is at most a factor δ greater than the optimum. We say that the algorithm has approximation factor δ. Sometimes approximation ratio or approximation guarantee is used instead of approximation factor.

5.3 Prior Publication

The results in Chapter 6, Section 4.1-Section 6.6 has previously appeared in:

4. "Asymmetry in k-Center Variants".
   Inge Li Gørtz and Anthony Wirth.
   In *Proceedings of the 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX), 2003.*

> An extended version has been accepted for publication in *Theoretical Computer Science, Special Issue on Approximation and Online Algorithms*.

Chapter 7 is unpublished work.

5.4 On Chapter 6: Asymmetry in k-Center Variants

In Chapter 6 we investigate the asymmetric variants of the *k-center* problem.

Section 4.0-6.6 is a minor revision of the version of paper 4 accepted for TCS. In this section we formally define the problem and relate our results to other work.

5.4.1 The k-Center Problem

Imagine you have a delivery service. You want to place your k delivery hubs at locations that minimize the maximum distance between customers and their nearest hubs. This is the *k-center* problem. Formally, given a complete graph on n vertices with nonnegative (but possibly infinite) edge costs, and a positive integer k, the k-center problem is to find a set of k vertices, called *centers*, minimizing the maximum distance to any vertex and from its nearest center.

The k-center problem is NP-hard [76]. Without the triangle inequality the problem is NP-hard to approximate within any factor. This can be shown by a reduction from the dominating set problem. We henceforth assume that the edge costs satisfy the triangle inequality. Hsu and Nemhauser [70], using the same reduction, showed that the metric k-center problem cannot be approximated within a factor of $(2 - \epsilon)$ unless P = NP. In 1985 Hochbaum and Shmoys [67] provided a (best possible) factor 2 algorithm for the metric k-center problem.

In the asymmetric k-center problem the graph is a complete digraph. The edge costs still obey the triangle inequality, but the cost of the edge from a node u to a node v might not be the same as the cost of the edge from v to u. The motivation for the *asymmetric k*-center problem, in our example, is that traffic patterns or one-way streets might cause the travel time from one point to another to differ depending on the direction of travel.

In 1996 Panigrahy and Vishwanathan [123, 96] gave the first approximation algorithm for the asymmetric problem, with factor $O(\log^* n)$. Archer [13] proposed two $O(\log^* k)$ algorithms based on many of the ideas of Panigrahy and Vishwanathan. The complementary $\Omega(\log^* n)$ hardness result [35, 65, 34] shows that these approximation algorithms are asymptotically optimal.

Variants of the k-Center Problem

A number of variants of the k-center problem have already been explored in the context of symmetric graphs.

Weighted k-Center Instead of a restriction on the number of centers, each vertex has a weight and we have a budget W, that limits the total weight of centers. Hochbaum and Shmoys [68] produced a factor 3 algorithm for this *weighted k-center problem*, which has recently been shown to be tight [35, 34].

We give an $O(\log^* n)$-approximation algorithm for the asymmetric version of the problem. Since the weighted k-center problem is a generalization of the standard k-center problem (set all weights to one) this is asymptotically optimal.

Priority k-Center In some cases some demand points might be more important than others. Plesnik [98] studied the *priority k-center problem*, in which the effective distance to a demand point is enlarged in proportion to its specified priority. Plesnik approximates the symmetric version within a factor of 2.

We show that the asymmetric version of the problem cannot be approximated within any factor unless P $=$ NP.

k-Center with Minimum Coverage Lim *et al.* [87] studied k-center with minimum coverage problems, where each center is required to serve a minimum of clients. This problem is motivated by trying to balance the workload and allow for economies of scale. Lim *et al.* defined two problems: the *q-all-coverage k-center problem*, where each center must cover at least q vertices (including itself), and the *q-coverage k-center problem*, where each center must cover at least q non-center nodes. For the q-all-coverage k-center problem Lim *et al.* gave a 2-approximation algorithm, and a 3-approximation algorithm for the weighted and priority versions of the problem. For the q-coverage k-center problem they gave a 2-approximation algorithm, and a 4-approximation algorithm for the weighted and priority versions of the problem.

We give a $O(\log^* n)$-approximation algorithm for the asymmetric version of both the q-all-coverage k-center problem and the q-coverage k-center problem. We also give a $O(\log^* n)$-approximation algorithm for the asymmetric weighted version of both problems. Since the k-center with minimum coverage problems are generalizations of the standard k-center problem this is asymptotically optimal.

p-Neighbor k-Center Khuller *et al.* [77] investigated the *p-neighbor k-center problem* in which each vertex must have p centers nearby. This problem is motivated by the need to account for facility failures: even if up to $p - 1$ facilities fail, every demand point has a functioning facility nearby. They gave a 3-approximation algorithm for all p, and a best possible 2-approximation algorithm when $p < 4$, noting that the case where p is small is "perhaps the practically interesting case".

For the asymmetric p-neighbor k-center problem we provide an $O(\log^* k)$-bicriteria algorithm using $2k$ centers, for $p \leq n/k$.

k-Supplier Hochbaum and Shmoys [68] studied the *k-supplier problem*, where the vertex set is segregated into suppliers and customers. Only supplier vertices can be centers and only the customer vertices need to be covered. Hochbaum and Shmoys gave a 3-approximation algorithm and showed that it is the best possible.

We show that the asymmetric k-supplier problem cannot be approximated within any factor unless P $=$ NP.

Outliers and Forbidden Centers Charikar *et al.* [26] noted that a disadvantage of the standard k-center formulation is that a few distant clients, *outliers*, can force centers to be located in isolated places. They suggested a variant of the problem, the k-center problem with *outliers and forbidden centers*, where a small subset of clients may be denied service, and some points are forbidden from being centers. Charikar *et al.* gave a (best possible) 3-approximation algorithm for the symmetric version of this problem.

We show that the asymmetric k-center problem with outliers and forbidden centers cannot be approximated within any factor unless P $=$ NP.

k-Center with Dynamic Distances Bhatia *et al.* [20] considered the problem in a network model, such as a city street network, in which the traversal times change as the day progresses. This is known as the *k-center problem with dynamic distances*: we wish to assign the centers such that the objective criteria are met at all times.

Bhatia *et al.* gave a $1 + \beta$-approximation algorithm for the symmetric k-center problem with dynamic distances, where β is the maximum ratio of an edge's greatest length to its smallest length. For the asymmetric version of the problem Bhatia *et al.* gave a $O(\log^* n + \nu)$-bicriteria algorithm using $k(1+3/(\nu+1))$ centers.

5.4.2 Our Techniques

The approximation algorithms for the weighted k-center, k-center with minimum coverage, and priority k-center problems are obtained by adapting the techniques for the standard k-center problem by Panigrahy and Vishwanathan [96]. The inapproximability results are shown by reductions from the max coverage problem.

5.4.3 Asymmetry in Other Graph Problems

Asymmetry is also a significant impediment to approximation in many other graph problems, such as facility location, k-median and the TSP.

The facility location and k-median problems are clustering problems similar to the k-center problem. In the *facility location* problem we are given a set of facilities and a set of cities. There is a cost of opening a facility and a cost of connecting a city to a facility. The problem is to find a subset of facilities to open and an assignment of all the cities to open facilities, such that the total cost of opening facilities and connecting cities is minimized. In the *k-median* problem we again have a set of facilities and a set of cities, and a cost of connecting cities to facilities, but instead of a cost for opening a facility we instead have a bound k on the number of facilities we are allowed to open. The problem is now to find at most k facilities to open and an assignment of all the cities to open facilities, such as to minimize the total connecting cost.

Both the facility location and k-median problems admit constant factor approximation algorithms in the symmetric case [89, 16], but are provably $\Omega(\log n)$ hard to approximate in the asymmetric case [12].

In the *traveling salesman problem* (TSP) we are given a graph with nonnegative edge costs, and the problem is to find a minimum cost cycle visiting every vertex exactly once.

There exists a $3/2$-approximation algorithm by Christofedes [32] for the symmetric traveling salesman problem (STSP), whereas for the asymmetric traveling salesman problem (ATSP) the best known approximation algorithm by Kaplan *et al.* [73] achieves a factor of $\frac{4}{3}\log n$. The best inapproximability results by Papadimitriou and Vempala [97], shows that there is no α-approximation algorithm for ATSP for $\alpha = \frac{117}{116} - \epsilon$ and for STSP for $\alpha = \frac{220}{219} - \epsilon$ unless P = NP.

5.5 On Chapter 7: Dial-a-Ride

In Chapter 7 we look at the dial-a-ride problem. In this section we formally define the problem, discuss its applications, and relate our results to other work.

5.5.1 Dial-a-Ride

Given a collection of objects in a metric space, a specified destination point for each object, and a vehicle with a capacity of at most k objects, the *finite capacity dial-a-ride problem*—dial-a-ride for short—is to compute a shortest tour for the vehicle in which all objects can be delivered to their destinations while ensuring that the vehicle carries at most k objects at any point in time. In the preemptive version of the problem an object may be dropped at intermediate locations and then picked up later by the vehicle and delivered. In the non-preemptive version of the problem once an object is loaded on the vehicle it stays on until it is delivered to its final destination.

The dial-a-ride problem is a generalization of the traveling salesman problem (TSP) even for $k = 1$ and is thus NP-hard. Placing an object and its destination at each vertex in the TSP instance yields an instance of the dial-a-ride problem. In this instance the vehicle has to simply find a shortest path tour that visits all the vertices, since any object that is picked up can be delivered immediately to its destination point in the same location.

Approximating Arbitrary Metrics by Tree Metrics

In order to discuss the approximation algorithms for the dial-a-ride problem we first review the results on approximating arbitrary metrics by tree metrics due to Fakcharoenphol *et al.* [48].

Let V be a set of points, and consider a metric space M over V. For $u, v \in V$, the distance between u and v in the metric M is denoted $d_M(u, v)$.

Probabilistic Approximation of Metric Spaces A metric space N over V *dominates* the metric space M over V if for all $u, v \in V$, we have $d_N(u, v) \geq d_M(u, v)$.

Let $\mathcal{S}$ be a family of metrics over V, and let $\mathcal{D}$ be a distribution over $\mathcal{S}$. We say that $(\mathcal{S}, \mathcal{D})$ α-probabilistically approximates a metric M over V if every metric in $\mathcal{S}$ dominates M and for every pair of vertices $u, v \in V$, $E_{N \in (\mathcal{S}, \mathcal{D})}[d_N(u, v)] \leq \alpha \cdot d_M(u, v)$.

Hierarchically Well-Separated Trees A k-hierarchically well-separated tree (k-HST) is a rooted weighted tree such that the weight from any node to each of its children is the same, and the edge weights along any root to leaf path decrease by a factor of 2 in each step.

Height-Balanced Trees Charikar and Raghavachari [28] defines *height-balanced trees*. Let the level i of a tree be all the edges which are the ith edge on the shortest path from the root to some leaf. A *height-balanced tree* is a rooted weighted tree such that,

1. For all leaves, the distance to the root is the same.
2. For all leaves, the path from the root to the leaf has the same number of edges.
3. All edges in the same level have the same length.

Charikar and Raghavachari showed that a 2-HST can be 4-approximated by a height-balanced tree.

Approximating Arbitrary Metrics Fakcharoenphol *et al.* [48] showed that any metric space M over V can be $O(\log n)$-probabilistic approximated by a set of 2-HSTs. The points in V occur as the leaves of the 2-HSTs.

This result can be used to obtain approximation algorithms of various problems including buy-at-bulk network design (to be defined later) and dial-a-ride. Take an instance I of the dial-a-ride problem on a general metric and turn it into an instance I' on a height-balanced tree. Let OPT be the cost of the optimal solution to I and let OPT' be the cost of the optimal solution to I'. Then $E(\mathsf{OPT}') \leq O(\log n \cdot \mathsf{OPT})$. A solution S' to I' can be turned into a solution S of I, such that the cost of S' is at most the cost of S. Thus, if we have a α-approximation for instances of the form of I', we get an $O(\alpha \cdot \log n)$-approximation algorithm for general cases.

Previous Results

The unit-capacity case is also known as the Stacker Crane Problem. Guan [62] showed NP-hardness of preemptive case on trees when $k = 2$. Frederickson and Guan [55] showed that the unit-capacity non-preemptive case is NP-hard on trees. For the unit-capacity non-preemptive case Frederickson *et al.* [56] gave an algorithm with approximation factor 1.8 on general graphs.

Let N denote the number of nodes in the input graph, i.e., the number of points that are either sources or destinations. Charikar and Raghavachari [28] studied the problem for general k. They gave an 2-approximation algorithm for the preemptive case on trees. For the non-preemptive case they gave an $O(\sqrt{k})$-approximation on *height-balanced* trees. Using the results by Fakcharoenphol *et al.* [48] this gives an $O(\log N)$-approximation algorithm for the preemptive case and an $O(\sqrt{k}\log N)$-approximation algorithm for the non-preemptive case.

Algorithm for the Preemptive Case on Trees The algorithm for the preemptive case on trees can be used on any trees, not only height-balanced trees. The *turning point* of an object d is the nearest common ancestor of its source and its destination. The algorithm performs two depth-first traversals of the tree. In the first depth-first traversal all objects are moved up in the tree from their source to their turning point. In the second depth-first traversal all objects are moved down from their turning point to their destination.

Charikar and Raghavachari used two lower bounds, which they denote as the *Steiner lower bound* and the *flow lower bound*, to show that this is 2-approximation algorithm. For object d let p_d be the unique shortest path from its source to its destination. The Steiner lower bounds states that the vehicle must visit all points that lie on some path p_d. The flow lower bound say that for any edge e the vehicle has to traverse e at least $2 \cdot \lceil f_e/k \rceil$ times, where f_e is the maximum number of paths p_d that passes e in one direction.

Algorithm for the Non-Preemptive Case on Trees In the non-preemptive case the algorithm works on height-balanced trees, where all sources and destinations are in the leaves. The algorithm visits all turning points in a depth-first order. When a turning point v is visited all objects having v as a turning point are delivered to their destination.

Charikar and Raghavachari used, in addition to the flow and Steiner lower bounds, a bound they call the *wait lower bound*. The idea of the wait lower bound is as follows. Consider a vertex v that has several objects that need to be delivered to different destinations. The optimal tour visits v a certain number of times. If the tour visits v a large number of times, the contribution of v to the length of the tour is large. If the tour visits v a small number of times, many objects must be picked up on each visit. When a large number of objects are picked up in a particular visit, then since none of the objects can be dropped off at intermediate locations and the destination of these objects are distinct the average time spent in the vehicle (wait time) for these items must be large. Therefore the contribution

to the length of the tour is again large.

Approximation Factor $O(k)$ As noted by Charikar and Raghavachari [28] using the $O(1)$-approximation algorithm in the general case ($k > 1$) gives an $O(k)$-approximation algorithm. To see this first note that any solution using capacity one is a valid solution. That is, $\mathsf{OPT}_k \leq \mathsf{OPT}_1$, where OPT_i is the value of the optimal solution to the problem using a vehicle with capacity i. We will also use OPT_i to denote the actual solution. Given a solution SOL_k to the problem using a vehicle with capacity k we can construct a solution SOL_1 to the problem using a vehicle of capacity 1 of at most k times the cost. Follow the same route as SOL_k when the first object is picked up in SOL_k pick up this object and deliver it to the vertex where it is dropped off in SOL_k. Then go back to the point were the object was picked up, and keep following the route of SOL_k until the next object is picked up. Do the same for this object and so on. If an edge is traversed c times in SOL_k then it is traversed at most $2k \cdot c$ times in SOL_1. Therefore $\mathsf{SOL}_1 \leq 2k \cdot \mathsf{SOL}_k$, and thus $\mathsf{OPT}_1 \leq 2k \cdot \mathsf{OPT}_k$.

We have not been able to find any references to a $O(1)$-approximation algorithm for the preemptive Dial-a-Ride problem with unit capacity in the literature. However, it is simple to construct a 3-approximation algorithm for the preemptive case with unit capacity: Construct a TSP tour on the destination nodes. Start at some source node s_i, use the shortest path to deliver item d_i to t_i. From t_i go to the next destination node $t_{i'}$ in the TSP tour and follow the shortest path from $t_{i'}$ to $s_{i'}$, and then carry on to deliver $d_{i'}$, and so on. The sum of shortest paths is a lower bound on the optimal solution and so is the TSP tour on the destination nodes. Each shortest path is used twice, so the total cost is at most 3 times the value of the optimal solution. Note that this is also a 3-approximation for the non-preemptive dial-a-ride.

Approximation Factor $O(N/k)$ We note that there is a trivial $\frac{3N}{k}$-approximation algorithm. Let us first consider the case when $k = N$. Then all objects can be in the vehicle at the same time. We can construct a tour by first taking a TSP tour on the sources, picking up all objects, and then a TSP tour on the destinations, delivering all objects. Both these TSP tours are a lower bound on value of the optimal solution as the vehicle will have to visit all of them. Using the $3/2$-approximation [32] to construct the TSP tours we get a tour of length $2 \cdot \frac{3}{2} \cdot \mathsf{OPT}_{TSP} \leq 3 \cdot \mathsf{OPT}$. For $k = N/c$ we do this c times each time picking up and delivering N/c objects. This gives an algorithm with approximation factor $3c = 3N/k$.

5.5.2 Special Cases and Related Problems

Several papers have discussed fast implementations of exact algorithms for special cases of the unit-capacity dial-a-ride problem ($k = 1$), e.g., when the underlying graph is a cycle or a tree, [17, 53, 54, 55].

Guan [62] gave a linear time algorithm for the preemptive dial-a-ride problem on a path, and showed that the non-preemptive dial-a-ride problem on a path is NP-hard on the line for $k \geq 2$.

In the k-delivery TSP all objects are identical and can be delivered to any of the destination point. Charikar *et al.* [27] gave a 5-approximation algorithm for both the preemptive and the non-preemptive problem.

Haimovich and Rinnooy Kan [64] gave an 3-approximation for the problem when all objects initially are located at one central depot.

5.5.3 Our Results

We study the hardness of approximation of the preemptive finite capacity dial-a-ride problem. We show that the preemptive Finite Capacity Dial-a-Ride problem has no $\min\{O(\log^{1/4-\varepsilon} N), k^{1-\varepsilon}\}$-approximation algorithm for any $\varepsilon > 0$ unless all problems in NP can be solved by randomized algorithms with expected running time $O(n^{\mathrm{polylog} n})$.

To our knowledge, the TSP lower bound was the best known so far (recall that the dial-a-ride problem generalizes TSP).

Our Techniques

Our results rely on the hardness results for the two network design problems Buy-at-Bulk and SumFiber-ChooseRoute shown by Andrews [9] and Andrews and Zhang [10].

Buy-at-Bulk In the *Buy-at-Bulk* problem we are given an undirected network $\mathcal{N}$, with lengths l_e on the edges, and a set $\{(s_i, t_i)\}$ of source-destination pairs. Each source-destination pair (s_i, t_i) has an associated demand δ_i. Each edge e has a cost function f_e, which is subadditive[1], and $f_e(0) = 0$. Since the cost function is subadditive it exhibits *economies of scale*.

The goal is to route all demands δ_i from their source s_i to their destination t_i minimizing the total cost. The demands are unsplittable, i.e., demand δ_i must follow a single path from s_i to t_i. The cost of an edge e is $f_e(x_e)$ where x_e is the

[1] $f_e(x + y) \leq f_e(x) + f_e(y)$.

amount of demand routed through e in the solution. The total cost of the solution is then

$$\sum_e f_e(x_e)l_e.$$

In this paper the cost function f_e is the same for all edges. This is also known as the *uniform* Buy-at-Bulk problem.

SumFiber-ChooseRoute In the *SumFiber-ChooseRoute* problem we are given an undirected network $\mathcal{N}$, with lengths l_e on the edges, and a set $\{(s_i, t_i)\}$ of source-destination pairs. Each source-destination pair (s_i, t_i) corresponds to a demand δ_i. Each demand requires bandwidth equivalent to one wavelength. Each fiber can carry k wavelengths, and the cost of deploying x fibers on edge e is $x \cdot l_e$. The problem is to specify a path from s_i to t_i for all demands δ_i, and a wavelength for the demand λ_i, minimizing the total cost. Let $f_e(\lambda)$ be the number of demands assigned to wavelength λ that are routed through edge e. Then $\max_\lambda f_e(\lambda)$ is the number of fibers needed on edge e. Thus the total cost of the solution is

$$\sum_e l_e \max_\lambda f_e(\lambda).$$

Hardness Both problems are NP-hard. Let N be the number of nodes in the network. Andrews [9] shows that there is no $O(\log^{\frac{1}{4}-\varepsilon} N)$-approximation algorithm for the uniform Buy-at-Bulk problem for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\mathrm{polylog} n})$.

Andrews and Zhang [10] show that there is no $O(\log^{\frac{1}{4}-\varepsilon} N)$-approximation algorithm for the problem for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\mathrm{polylog} n})$.

The hardness results by Andrews [9] and Andrews and Zhang [10] are obtained by a reduction using the Raz verifier for MAX3SAT(5) [102]. The Raz verifier for a MAX3SAT(5) instance ϕ is used to construct a random high-girth network. For each demand they define a set of *canonical paths* on which the demand can be carried. These canonical paths correspond to accepting interactions and are short paths directly connecting the source and the destination. They show that if ϕ is satisfiable then the optimal solution to the instance of SumFiber-ChooseRoute has small cost, and if ϕ is unsatisfiable then the optimal solution has high cost. More precisely, the cost if ϕ is unsatisfiable is a factor of γ more than if ϕ is satisfiable. Hence if there were an α-approximation for SumFiber-ChooseRoute with $\alpha < \gamma$, then we would be able to determine if ϕ was satisfiable.

Hardness of Dial-a-Ride To get our hardness result for Dial-a-Ride we show a relationship between the Buy-at-Bulk problem with cost function $h(x) = \lceil \frac{x}{k} \rceil$ and

Dial-a-Ride problem in the high-girth network $\mathcal{N}$ constructed from the MAX3SAT instance. More precisely, let B be a Buy-at-Bulk instance in the network $\mathcal{N}$ with source destination pairs S and cost function $h(x) = \lceil \frac{x}{k} \rceil$, and let D be an instance of the preemptive Dial-a-Ride instance in $\mathcal{N}$ with the same source-destination pairs S. We show that

$$\mathsf{OPT}_B^{\mathcal{N}} \leq \mathsf{OPT}_D^{\mathcal{N}} \leq 7 \cdot \mathsf{OPT}_B^{\mathcal{N}} , \tag{5.1}$$

where $\mathsf{OPT}_B^{\mathcal{N}}$ is the value of the optimal solution to B, and $\mathsf{OPT}_D^{\mathcal{N}}$ is the value of the optimal solution to D.

We show hardness results for Buy-at-Bulk problem with cost function $h(x) = \lceil \frac{x}{k} \rceil$. This is obtained by changing the parameters in the network used by Andrews and Zhang [10]. Combining this result with Equation 5.1 we get our hardness results for Dial-a-Ride.

Chapter 6

Asymmetry in k-Center Variants

In this chapter we explore three concepts: the k-center problem, some of its variants, and asymmetry. The k-center problem is fundamental in location theory. Variants of k-center may more accurately model real-life problems than the original formulation. Asymmetry is a significant impediment to approximation in many graph problems, such as k-center, facility location, k-median and the TSP.

We provide an $O(\log^* n)$-approximation algorithm for the asymmetric *weighted* k-center problem. Here, the vertices have weights and we are given a total budget for opening centers. In the *p-neighbor* variant each vertex must have p (unweighted) centers nearby: we give an $O(\log^* k)$-bicriteria algorithm using $2k$ centers, for small p. In k-center with minimum coverage, each center is required to serve a minimum of clients. We give an $O(\log^* n)$-approximation algorithm for this problem.

Finally, the following three versions of the asymmetric k-center problem we show to be inapproximable: *priority k-center*, *k-supplier*, and *outliers with forbidden centers*.

6.1 Introduction

Imagine you have a delivery service. You want to place your k delivery hubs at locations that minimize the maximum distance between customers and their nearest hubs. This is the *k-center* problem—a type of clustering problem that is similar to the facility location [89] and k-median [16] problems. The motivation for the *asymmetric k-center* problem, in our example, is that traffic patterns or one-way streets might cause the travel time from one point to another to differ depending on the direction of travel. Traditionally, the k-center problem was solved in the context of a metric; in this chapter we retain the triangle inequality,

but abandon the symmetry.

Symmetry is a vital concept in graph approximation algorithms. Recently, the k-center problem was shown to be $\Omega(\log^* n)$ hard to approximate [35, 65, 34], even though the symmetric version has a factor 2 approximation. Facility location and k-median both have constant factor algorithms in the symmetric case, but are provably $\Omega(\log n)$ hard to approximate without symmetry [12]. The traveling salesman problem is a little better, in that no super-constant hardness is known, but without symmetry no algorithm better than $\frac{4}{3}\log n)$ [73] has been found either.

Definition 6.1.1 (k-Center)**.** Given $G = (V, E)$, a complete graph with nonnegative (but possibly infinite) edge costs, and a positive integer k, find a set S of k vertices, called *centers*, with minimum covering radius. The covering radius of a set S is the minimum distance R such that every vertex in V is within distance R of some vertex in S.

Kariv and Hakimi [76] showed that the k-center problem is NP-hard. Without the triangle inequality the problem is NP-hard to approximate within any factor (there is a straightforward reduction from the dominating set problem). We henceforth assume that the edge costs satisfy the triangle inequality. Hsu and Nemhauser [70], using the same reduction, showed that the metric k-center problem cannot be approximated within a factor of $(2 - \epsilon)$ unless P = NP. In 1985 Hochbaum and Shmoys [67] provided a (best possible) factor 2 algorithm for the metric k-center problem. In 1996 Panigrahy and Vishwanathan [123, 96] gave the first approximation algorithm for the asymmetric problem, with factor $O(\log^* n)$. Archer [13] proposed two $O(\log^* k)$ algorithms based on many of the ideas of Panigrahy and Vishwanathan. The complementary $\Omega(\log^* n)$ hardness result [35, 65, 34] shows that these approximation algorithms are asymptotically optimal.

6.1.1 Variants of the k-Center Problem

A number of variants of the k-center problem have been explored in the context of symmetric graphs. Perhaps some delivery hubs are more expensive to establish than others. Instead of a restriction on the number of centers we can use, each vertex has a weight and we have a budget W, that limits the total weight of centers. Hochbaum and Shmoys [68] produced a factor 3 algorithm for this *weighted k-center* problem, which has recently been shown to be tight [35, 34].

Hochbaum and Shmoys [68] also studied the *k-supplier* problem, where the vertex set is segregated into suppliers and customers. Only supplier vertices can

be centers and only the customer vertices need to be covered. Hochbaum and Shmoys gave a 3-approximation algorithm and showed that it is the best possible.

Khuller et al. [77] investigated the *p-neighbor k-center* problem where each vertex must have p centers nearby. This problem is motivated by the need to account for facility failures: even if up to $p-1$ facilities fail, every demand point has a functioning facility nearby. They gave a 3-approximation algorithm for all p, and a best possible 2-approximation algorithm when $p < 4$, noting that the case where p is small is "perhaps the practically interesting case".

Maybe some demand points are more important than others. Plesnik [98] studied the *priority k-center* problem, in which the effective distance to a demand point is enlarged in proportion to its specified priority. Plesnik approximates the symmetric version within a factor of 2.

Charikar et al. [26] note that a disadvantage of the standard k-center formulation is that a few distant clients, *outliers*, can force centers to be located in isolated places. They suggest a variant of the problem, the k-center problem with *outliers and forbidden centers*, where a small subset of clients may be denied service, and some points are forbidden from being centers. Charikar et al. gave a (best possible) 3-approximation algorithm for the symmetric version of this problem.

Lim *et al.* [87] studied k-center with minimum coverage problems, where each center is required to serve a minimum of clients. This problem is motivated by trying to balance the workload and allow for economies of scale. Lim *et al.* defined two problems: the *q-all-coverage k-center problem*, where each center must cover at least q vertices (including itself), and the *q-coverage k-center problem*, where each center must cover at least q non-center nodes. For the q-all-coverage k-center problem Lim *et al.* gave a 2-approximation algorithm, and a 3-approximation algorithm for the weighted and priority versions of the problem. For the q-coverage k-center problem they gave a 2-approximation algorithm, and a 4-approximation algorithm for the weighted and priority versions of the problem.

Bhatia et al. [20] considered a network model, such as a city street network, in which the traversal times change as the day progresses. This is known as the *k-center problem with dynamic distances*: we wish to assign the centers such that the objective criteria are met at all times.

6.1.2 Results and Organization

Table 6.1 gives an overview of the best known results for the various k-center problems. In this chapter we explore asymmetric variants that are not yet in the literature.

Problem	Symmetric		Asymmetric	
k-center	2	[67]	$O(\log^* k)$	[13]
k-center with dynamic distances	$1 + \beta$ †	[20]	$O(\log^* n + \nu)$ ‡	[20]
weighted k-center	3	[68]	$\mathbf{O(log^*n)}$	[60]
q-all-coverage k-center (weighted)	2 (3)	[87]	$\mathbf{O(log^*n)}$	
q-coverage k-center (weighted)	2 (4)	[87]	$\mathbf{O(log^*n)}$	
p-neighbor k-center	3 (2 §)	[29]	$\mathbf{O(log^*k)}$ ¶	[60]
priority k-center	2	[98]	**Inapproximable**	[60]
k-center with outliers and forbidden centers	3	[26]	**Inapproximable**	[60]
k-suppliers	3	[68]	**Inapproximable**	[60]

Table 6.1: An overview of the approximation results for k-center variants. The results in this chapter are in boldface. †β is the maximum ratio of an edge's greatest length to its smallest length. ‡This is a bicriteria algorithm using $k(1 + 3/(\nu + 1))$ centers. §For $p < 4$. ¶This is a bicriteria algorithm using $2k$ centers, for $p \leq n/k$

Section 6.2 contains the definitions and notation required to develop the results. In Section 6.3 we briefly review the algorithms of Panigrahy and Vishwanathan [96], and Archer [13]. The techniques used in the standard k-center problem are often applicable to the variants.

Our first result, in Section 6.4, is an $O(\log^* n)$-approximation for the asymmetric weighted k-center problem. In Section 6.5 we develop an $O(\log^* k)$ approximation for the asymmetric p-neighbor k-center problem, for $p \leq n/k$. As noted by Khuller et al. [77], the case where p is small is the most interesting case in practice. This a bicriteria algorithm, allowing $2k$ centers to be used rather than just k. It can, however, be converted to an $O(\log k)$-approximation algorithm using only k centers. Turning to hardness, we show that the asymmetric versions of the k-center problem with outliers (and forbidden centers), the priority k-center problem, and the k-supplier problem are NP-hard to approximate within any factor (Section 6.6). Finally, in Section 6.7 we provide $O(\log^* n)$-approximation algorithms for the asymmetric q-all-coverage and q-coverage k-center problems and their weighted versions.

6.2 Definitions

To avoid any uncertainty, we note that $\log$ stands for $\log_2$ by default, while $\ln$ stands for $\log_e$.

Definition 6.2.1. For every integer $i > 1$, $\log^i x = \log(\log^{i-1} x)$, and $\log^1 x = \log x$. We let $\log^* x$ represent the smallest integer i such that $\log^i x \leq 2$.

The input to the asymmetric k-center problem is a distance function d on every *ordered* pair of vertices—distances are allowed to be infinite—and a bound k on the number of centers. Note that we assume that the edges are *directed*.

Definition 6.2.2. Vertex c *covers* vertex v *within* r, or c r*-covers* v, if $d_{cv} \leq r$. We extend this definition to a sets so that a set C r-covers a set A if for every $a \in A$ there is some $c \in C$ such that c covers a within r. Often we abbreviate "1-covers" to "covers".

Many of the algorithms for k-center and its variants do not, in fact, operate on graphs with edge costs. Rather, they consider bottleneck graphs [68], in which only those edges with distance lower than some threshold are included, and they appear in the bottleneck graph with unit cost. Since the optimal value of the covering radius must be one of the $n(n-1)$ distance values, many algorithms essentially run through a sequence of bottleneck graphs of every possible threshold radius in ascending order. This can be thought of as *guessing* the optimal radius R_{OPT}. The approach works because the algorithm either returns a solution, within the specified factor of the current threshold radius, or it fails, in which case R_{OPT} must be greater than the current radius.

Definition 6.2.3 (Bottleneck Graph G_r)**.** For $r > 0$, define the bottleneck graph G_r of the graph $G = (V, E)$ to be the graph $G_r = (V, E_r)$, where $E_r = \{(i, j) : d_{ij} \leq r\}$ and all edges have unit cost.

Most of the following definitions apply to *bottleneck* graphs.

Definition 6.2.4 (Power of Graphs)**.** The tth power of a graph $G = (V, E)$ is the graph $G^t = (V, E^{(t)})$, $t > 1$, where $E^{(t)}$ is the set of ordered pairs of distinct vertices that have a path of at most t edges between them in G.

Definition 6.2.5. For $i \in \mathbb{N}$ define

$$\Gamma_i^+(v) = \{u \in V \mid (v, u) \in E^i\} \cup \{v\}, \qquad \Gamma_i^-(v) = \{u \in V \mid (u, v) \in E^i\} \cup \{v\} ,$$

i.e., in the bottleneck graph there is a path of length at most i from v to u, respectively u to v.

Notice that in a symmetric graph $\Gamma_i^+(v) = \Gamma_i^-(v)$. We extend this notation to sets so that $\Gamma_i^+(S) = \{u \in V \mid u \in \Gamma_i^+(v) \text{ for some } v \in S\}$, with $\Gamma_i^-(S)$ defined similarly. We use $\Gamma^+(v)$ and $\Gamma^-(v)$ instead of $\Gamma_1^+(v)$ and $\Gamma_1^-(v)$.

Definition 6.2.6. For $i \in \mathbb{N}$ define

$$\Upsilon_i^+(v) = \Gamma_i^+(v) \setminus \Gamma_{i-1}^+(v), \qquad \Upsilon_i^-(v) = \Gamma_i^-(v) \setminus \Gamma_{i-1}^-(v) \ ,$$

i.e., the nodes for which the path distance from v is exactly i, and the nodes for which the path distance to v is exactly i, respectively.

For a set S, the extension follows the pattern $\Upsilon_i^+(S) = \Gamma_i^+(S) \setminus \Gamma_{i-1}^+(S)$. We use $\Upsilon^+(v)$ and $\Upsilon^-(v)$ instead of $\Upsilon_1^+(v)$ and $\Upsilon_1^-(v)$.

We call x a *parent* of y, and y a *child* of x, if $x \in \Upsilon^-(y)$. If $\Upsilon^-(y)$ is empty we call y an *orphan*.

Definition 6.2.7 (Center Capturing Vertex (CCV))**.** A vertex v is a *center capturing vertex* (CCV) if $\Gamma^-(v) \subseteq \Gamma^+(v)$, i.e., v covers every vertex that covers v.

In the graph $G_{R_{\mathsf{OPT}}}$ the optimum center that covers v must lie in $\Gamma^-(v)$; for a CCV v, it lies in $\Gamma^+(v)$, hence the name. In symmetric graphs all vertices are CCVs and this property leads to the standard 2-approximation.

The following three problems, related to k-center, are all NP-complete [59].

Definition 6.2.8 (Dominating Set)**.** Given a graph $G = (V, E)$, and a weight function $w : V \to \mathbb{Q}^+$ on the vertices, find a minimum weight subset $D \subseteq V$ such that every vertex $v \in V$ is covered by D, i.e., $\Gamma^+(D) = V$.

Definition 6.2.9 (Set Cover)**.** Given a universe $\mathcal{U}$ consisting of n elements, a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of subsets of $\mathcal{U}$, and a weight function $w : \mathcal{S} \to \mathbb{Q}^+$, find a minimum weight sub-collection of $\mathcal{S}$ that includes all elements of $\mathcal{U}$.

Definition 6.2.10 (Max Coverage)**.** Given $\langle \mathcal{U}, \mathcal{S}, k \rangle$, with $\mathcal{U}$ and $\mathcal{S}$ as above, find a sub-collection of k sets that includes the maximum number of elements of $\mathcal{U}$.

6.3 Asymmetric k-Center Review

The $O(\log^* n)$ algorithm of Panigrahy and Vishwanathan [96] has two phases, the *halve* phase, sometimes called the *reduce* phase, and the *augment* phase. As described above, the algorithm guesses R_{OPT}, and works in the bottleneck graph $G_{R_{\mathsf{OPT}}}$. In the halve phase we find a CCV v, include it in the set of centers, mark every vertex in $\Gamma_2^+(v)$ as covered, and repeat until no CCVs remain unmarked.

The CCV property ensures that, as each CCV is found and vertices are marked, the unmarked portion of the graph can be covered with one fewer center. Hence if k'' CCVs are obtained, the unmarked portion of the graph can be covered with $k' = k - k''$ centers. The authors then prove that this unmarked portion, CCV-free, can be covered with only $k'/2$ centers if we use radius 5 instead of 1. That is to say, $k'/2$ centers suffice in the graph $G^5_{R_{\mathsf{OPT}}}$.

The k-center problem in the bottleneck graph is identical to the dominating set problem. This is a special case of set cover in which the sets are the Γ^+ terms. In the augment phase, the algorithm recursively uses the greedy set cover procedure. Since the optimal cover uses at most $k'/2$ centers, the first cover has size at most $\frac{k'}{2} \log \frac{2n}{k'}$.

The centers in this first cover are themselves covered, using the greedy set cover procedure, then the centers in the second cover, and so forth. After $O(\log^* n)$ iterations the algorithm finds a set of at most k' vertices that, together with the CCVs, $O(\log^* n)$-covers the unmarked portion, since the optimal solution has $k'/2$ centers. Combining these with the k'' CCVs, we have k centers covering the whole graph within $O(\log^* n)$.

Archer [13] presents two $O(\log^* k)$ algorithms, both building on the work by Panigrahy and Vishwanathan [96]. The algorithm more directly connected with the earlier work nevertheless has two fundamental differences. Firstly, in the reduce phase Archer shows that the CCV-free portion of the graph can be covered with $2k'/3$ centers within radius 3. Secondly, he constructs a set cover-like integer program and solves the relaxation to get a total of k' *fractional* centers that cover the unmarked vertices. From these fractional centers, he obtains a 2-cover of the unmarked vertices with $k' \log k'$ (integral) centers. These are the seed for the augment phase, which thus produces a solution with an $O(\log^* k')$ approximation to the optimum radius. We now know that all of these approximation algorithms are asymptotically optimal [35, 65, 34].

6.4 Asymmetric Weighted k-Center

Recall the application in which the costs of delivery hubs vary. In this situation, rather than having a restriction on the *number* of centers used, each vertex has a *weight* and we have a budget W that restricts the total weight of centers used.

Definition 6.4.1 (Weighted k-Center)**.** Given a weight function on the vertices, $w : V \to \mathbb{Q}^+$, and a bound $W \in \mathbb{Q}^+$, find a set $S \subseteq V$ of total weight at most W, so that S covers V with minimum radius.

Hochbaum and Shmoys [68] gave a 3-approximation algorithm for the symmetric weighted version, applying their approach for bottleneck problems. We propose an $O(\log^* n)$-approximation for the asymmetric version, based on Panigrahy and Vishwanathan's technique for the unweighted problem. Note that in light of the complementary hardness result just announced [35, 65, 34], this algorithm is asymptotically the best possible. There is another variant that has both the k and the W restrictions, but we will not expand on that problem here.

First, a brief sketch of the algorithm, which works with bottleneck graphs. In the reduce phase, having found a CCV, v, we pick the lightest vertex u in $\Gamma^-(v)$ (which might be v itself) as a center in our solution. We then mark everything in $\Gamma_3^+(u)$ as covered, and continue looking for CCVs. We can show that there exists a 49-cover of the unmarked vertices with total weight less than a quarter of the optimum. Finally, we recursively apply a greedy procedure for weighted sets and elements $O(\log^* n)$ times, similar to the one used for set cover. The total weight of centers in our solution set is at most W.

The following lemma concerning vertex-weighted digraphs is the key to our reduce phase and is analogous to Lemma 4 in Panigrahy and Vishwanathan's paper [96] and Lemma 16 in Archer's [13].

Lemma 6.4.2. *Let $G = (V, E)$ be a digraph with weighted vertices, but unit edge costs. Then there is a subset $S \subseteq V$, $w(S) \leq w(V)/2$, such that every vertex with positive indegree is reachable in at most 3 steps from some vertex in S.*

Proof. To construct the set S repeat the following, to the extent possible: *Select* a vertex v with positive outdegree and if possible *select* one with indegree zero (that is, $\Upsilon^-(v)$ is empty). Compare sets $\{v\}$ and $\Upsilon^+(v)$: add the lighter set to S and remove $\Gamma^+(v)$ from G.

It is clear that the weight of S is no more than half the weight of V. We must now show that S 3-covers all non-orphan vertices.

The children of a selected vertex v, $\Upsilon^+(v)$, are clearly 1-covered. Assume v is not in S (trivial otherwise): if v was an orphan initially then ignore it. If v is an orphan when selected, but not initially, then at some previous stage in the procedure some parent of v must have been removed by the selection of a grandparent (a vertex in $\Upsilon_2^-(v)$), so v is 2-covered. Note that if one of v's parents had been selected then v would already have been removed from G.

Now assume v has at least one parent when it is selected. Consequently, at that state in the procedure, there are no vertices that have children, but are orphans, otherwise on of them would have been selected instead of v. We conclude that the sets of parents of v, $S_1 = \Upsilon^-(v)$, parents of S_1, $S_2 = \Upsilon^-(S_1)$, and parents of S_2,

$S_3 = \Upsilon^-(S_2)$, are not empty. Although these sets might not be pairwise disjoint, if they contained any of v's children, then v would be 3-covered.

After v is removed, there are three possibilities for S_2: (i) Some vertex in S_3 is selected, removing part of S_2; (ii) Some vertex in S_2 is selected and removed; (iii) Some vertex in S_1 is selected, possibly making some S_2 vertices childless. One of these events *must* happen, since S_1 and S_2 are non-empty. As a consequence, v is 3-covered. ☐

Henceforth call the vertices that have not yet been covered/marked *active*. Using Lemma 6.4.2 we can show that after removing the CCVs from the graph, we can cover the active set with half the weight of an optimum cover if we are allowed to use distance 7 instead of 1.

Lemma 6.4.3. *Consider a subset $A \subseteq V$ that has a cover consisting of vertices of total weight W, but no CCVs. Assume there exists a set C_1 that 3-covers exactly $V \setminus A$. Then there exists a set of vertices S of total weight $W/2$ that, together with C_1, 7-covers A.*

Proof. Let U be a subset of the optimal centers that covers A. We call $u \in U$ a *near* center if it can be reached in 4 steps from C_1, and a *far* center otherwise. Since C_1 5-covers all of the nodes covered by near centers, it suffices to choose S to 6-cover the far centers, so that S will 7-cover all the nodes they cover.

Define an auxiliary graph H on the (optimal) centers U as follows. There is an edge from x to y in H if and only if x 2-covers y in G (and $x \neq y$). The idea is to show that any far center has positive indegree in H. As a result, Lemma 6.4.2 shows there exists a set $S \in U$ with $|S| \leq W/2$ such that S 3-covers the far centers in H, and thus 6-covers them in G.

Let x be any far center: note that $x \in A$. Since A contains no CCVs, there exists y such that y covers x, but x does not cover y. Since $x \notin \Gamma_4^+(C_1)$, $y \notin \Gamma_3^+(C_1)$, and thus $y \in A$ (since everything not 3-covered by C_1 is in A). Thus there exists a center $z \in U$, which is not x, but might be y, that covers y and therefore 2-covers x. Hence x has positive indegree in the graph H. ☐

As we foreshadowed, we will use the greedy heuristic to complete the algorithm. We now analyze the performance of this heuristic in the context of the dominating set problem in node-weighted graphs. All vertices V are available as potential members of the dominating set (i.e. centers), but we need only dominate the active vertices A. The heuristic is to select the most *efficient* vertex: the one that maximizes $w(A(v))/w(v)$, where $A(v) \equiv A \cap \Gamma^+(v)$.

Lemma 6.4.4. *Let $\langle G = (V, E), w : V \to \mathbb{Q}^+, A \subseteq V\rangle$ be an instance of the dominating set problem in which a set A is to be dominated. Also, let w^* be the weight of an optimum solution for this instance. The greedy algorithm gives an approximation guarantee of*

$$2 + \ln(w(A)/w^*) \ .$$

Proof. In every application of the greedy selection there must be some vertex $v \in V$ for which

$$\frac{w(A(v))}{w(A)} \geq \frac{w(v)}{w^*} \tag{6.1}$$

otherwise no optimum solution of weight w^* would exist. This is certainly true of the most efficient vertex v, so make v a center and make all the vertices it covers inactive, leaving A' active. Now,

$$w(A') = w(A) - w(A(v)) \leq w(A)\left(1 - \frac{w(v)}{w^*}\right) < w(A)\exp\left(-\frac{w(v)}{w^*}\right) \ .$$

After j steps, the remaining active vertices, A^j, satisfy

$$w(A^j) < w(A^0)\prod_{i=1}^{j}\exp\left(-\frac{w(v_i)}{w^*}\right) \ , \tag{6.2}$$

where v_i is the ith center picked (greedily) and A^0 is the original active set.

Assume that after some number of steps, say j, there are still some active elements, but the upper bound in (6.2) has just dropped below w^*. That is to say,

$$\sum_{i=1}^{j} w(v_i) > w^* \ln(w(A^0)/w^*) \ .$$

Before we picked the vertex v_j we had

$$\sum_{i=1}^{j-1} w(v_i) \leq w^* \ln(w(A^0)/w^*) \ , \quad \text{and so,} \quad \sum_{i=1}^{j} w(v_i) \leq w^* + w^* \ln(w(A^0)/w^*) \ ,$$

for (6.1) tells us that $w(v_j)$ is no greater than w^*. To cover the remainder, A^j, we just use A^j itself, at a cost less than w^*. Hence the total weight of the solution is at most $w^*(2 + \ln(w(A^0)/w^*))$.

On the other hand, if the upper bound on $w(A^j)$ never drops below w^* before A^j becomes empty, then we have a solution of weight at most $w^* \ln(w(A^0)/w^*)$. □

We now show that this tradeoff between covering radius and optimal cover size leads to an $O(\log^* n)$ approximation.

Lemma 6.4.5. *Given $A \subseteq V$, such that A has a cover of weight W, and a set $C_1 \subseteq V$ that covers $V \setminus A$, we can find in polynomial time a set of vertices of total weight at most $4W$ that, together with C_1, covers A (and hence V) within a radius of $O(\log^* n)$.*

Proof. We will be applying the greedy algorithm of Lemma 6.4.4. The approximation guarantee is $2 + \ln(w(A)/W)$, which is less than $\log_{1.5}(w(A)/W)$ when $w(A) \geq 4W$.

Our first attempt at a solution, S_0, is all vertices of weight no more than W. Only these vertices could be in the optimum center set and their total weight is at most nW. Since C_1 covers $S_0 \setminus A$, consider $A_0 = S_0 \cap A$, which has a cover of size W. Lemma 6.4.4 shows that the greedy algorithm results in a set S_1 that covers A_0 and has weight

$$w(S_1) \leq W \log_{1.5}(\frac{Wn}{W}) = W \log_{1.5} n \ ,$$

assuming $n \geq 4$. The set C_1 covers $S_1 \setminus A$, so we need only consider $A_1 = S_1 \cap A$. We continue this procedure and note that at the ith iteration we have $w(S_i) \leq W \log_{1.5}(w(S_{i-1})/W)$. By induction, after $O(\log^* n)$ iterations the weight of our solution set, S_i, is at most $4W$. □

All the algorithmic tools can now be assembled to form an approximation algorithm.

Theorem 6.4.6. *The weighted k-center problem can be approximated within a factor of $O(\log^* n)$ in polynomial time.*

Proof. Guess the optimum radius, R_{OPT}, and work in the bottleneck graph $G_{R_{\mathsf{OPT}}}$. Initially, the active set A is V. Repeat the following as many times as possible: Pick a CCV v in A, add the lightest vertex u in $\Gamma^-(v)$ to our solution set of centers, and remove the set $\Gamma_3^+(u)$ from A. Since v is covered by an optimum center in $\Gamma^-(v)$, u is no heavier than this optimum center. Moreover, since the optimum center lies in $\Gamma^+(v)$, $\Gamma_3^+(u)$ includes everything covered by it.

Let C_1 be the centers chosen in this first phase. We know the remainder of the graph, A, has a cover of total weight $W' = W - w(C_1)$, because of our choices based on CCV and weight.

Lemma 6.4.3 shows that we can cover the remaining uncovered vertices with weight no more than $W'/2$ if we use covering radius 7. Applying the lemma again, we can cover the remaining vertices with weight $W'/4$ centers if we allow radius 49. So let the active set A be $V \setminus \Gamma_{49}^+(C_1)$, and recursively apply the greedy algorithm as described in the proof of Lemma 6.4.5 on the graph $G_{R_{\mathsf{OPT}}}^{49}$. As a result, we have a set of size W' that covers A within radius $O(\log^* n)$. □

6.5 Asymmetric p-Neighbor k-Center

Imagine that we wish to place k facilities so that the maximum distance of a demand point from its pth-closest facility is minimized. As a consequence, failures in $p-1$ facilities do not cause severe network performance loss.

Definition 6.5.1 (Asymmetric p-Neighbor k-Center Problem)**.** For every subset S and vertex v in V, let $d_p(S, v)$ denote the distance from the pth closest vertex in S to v. The problem is to find a subset S of at most k vertices that minimizes $\max_{v \in V \setminus S} d_p(S, v)$.

We show how to approximate the asymmetric p-neighbor k-center problem within a factor of $O(\log^* k)$ if we allow ourselves to use $2k$ centers. Our algorithm is restricted to the case $p \leq n/k$, but this is reasonable as p should not be too large [77].

We use the same techniques as before, including bottleneck graphs, but in the augment phase we use the greedy algorithm for the *constrained set multicover* problem [122]. That is, each element, e, needs to be covered r_e times, but each set can be picked at most once. The p-neighbor k-center problem has $r_e = p$ for all e. We say that an element e is *active* if it occurs in fewer than p sets chosen so far. The greedy heuristic is to pick the set that covers the most active elements. It can be shown that this algorithm achieves an approximation factor of $H_n = O(\log n)$ [122, Section 13.2]. However the following result is more appropriate to our application.

Lemma 6.5.2. *Let k be the value of the optimum solution to the Constrained Set Multicover problem. The greedy algorithm gives approximation guarantee of* $\log_{1.5}(np/k)$.

Proof. The same kind of averaging argument used for standard set cover shows that the greedy choice of a set reduces the total number of unmarked element copies by a factor $1 - 1/k$. So after i steps, the number of copies of elements yet to be covered is $np(1 - 1/k)^i < np(e^{-1/k})^i$. Hence after $k \ln(np/k)$ steps the number of uncovered copies of elements is less than k. A naive cover of these last k element copies leads the total number of sets in the solution to be at most $k + k\ln(np/k)$. Since $p \geq 2$, this greedy algorithm has an approximation factor less than $\log_{1.5}(np/k)$. □

If $p \leq n/k$ the approximation guarantee above is less than $\log_{1.2}(n/k)$. We can now apply the standard recursive approach [96]. Recall that Panigrahy and Vishwanathan use $O(\log^* n)$ iterations to get down to $2k$ centers, which gives them

a $O(\log^* n)$ approximation because of the halve phase. They also state that using $O(\log n)$ iterations instead they would get down to k centers without the halve phase. Since we do not have anything similar to the halve phase, for the p-neighbor k-center problem we need $O(\log n)$ iterations to get down to k centers. There is no analogy to Lemma 4 [96], in which Panigrahy and Vishwanathan show that all vertices with positive indegree can be 2-covered by half the number of centers.

The approximation guarantee can be lowered to $O(\log^* k)$, with $2k$ centers, using Archer's LP-based priming, which we describe now in detail.

We first solve the LP for the constrained set multicover problem. Let y_v be the (fractional) extent to which a vertex is a center:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} y_v \\ \text{subject to} \quad & \sum_{u \in \Gamma^-(v)} y_u \geq p, && v \in A \\ & -y_v \geq -1, && v \in V \\ & y_v \geq 0, && v \in V\,. \end{aligned}$$

In the solution each vertex is covered by an amount p of fractional centers, out of a total of k. We can now use the greedy method to obtain an initial set of $k^2 \ln k$ centers that 2-covers every vertex in the active set with at least p centers.

Let A be the active vertices (the vertices that are covered fewer than p times) and let $A(v) = \Gamma^+(v) \cap A$. Let $y'(v) = y_v \cdot a_v$, where a_v is the number of times v still needs to be covered, and let $y'(S) = \sum_{v \in S} y'(v)$ for all $S \subseteq V$. Note that $v \in A \Leftrightarrow a_v > 0$ and thus $y'(A) = y'(V)$. The function y' indicates the extent to which an optimal fractional center is not yet covered. We will see that when the value of $y'(V)$ is low, we can be sure that we have found a reasonable cover of all the vertices.

Start with an empty set S and repeat the following until $y'(V) < p$: Choose the vertex v from $T = V - S$ maximizing $y'(\Gamma^+(v))$, add it to S, and set $a_u = a_u - 1$ for all vertices $u \in A(v)$.

Lemma 6.5.3. *Once $y'(V) < p$, the set S 2-covers every vertex with at least p centers.*

Proof. For every v, let $\alpha(v)$ be its active parents, $\alpha(v) = \{u : u \in \Gamma^-(v), a_u \geq 1\}$, and let $\beta(v)$ be its inactive parents, $\beta(v) = \{u : u \in \Gamma^-(v), a_u = 0\}$.

Since $y'(V) < p$ we have

$$\sum_{u \in \alpha(v)} y_u \leq \sum_{u \in \alpha(v)} y'_u < p\,.$$

By the first LP constraint we have

$$\sum_{u\in\alpha(v)} y_u + \sum_{u\in\beta(v)} y_u = \sum_{u\in\Gamma^-(v)} y_u \geq p \ ,$$

and thus $\sum_{u\in\beta(v)} y_u > 0$. We conclude that there must be at least one vertex in $\beta(v)$. The p vertices covering this vertex 2-cover v. □

The following lemma corresponds to Archer's Lemma 4 [13].

Lemma 6.5.4. *There exists $v \in T$ such that*

$$y'(A(v)) \geq \frac{y'(A)}{y(T)} \ .$$

Proof. We take a weighted average of $y'(A(v))$ over $v \in T$.

$$\begin{aligned}
\frac{1}{y(T)}\sum_{v\in T} y_v \cdot y'(A(v)) &= \frac{1}{y(T)}\sum_{v\in T}\sum_{u\in A(v)} y_v \cdot y'(u) \\
&= \frac{1}{y(T)}\sum_{u\in A} y'(u) \sum_{v\in\Gamma^-(u)\cap T} y_v \\
&\geq \frac{1}{y(T)}\sum_{u\in A} y'(u)
\end{aligned}$$

The inequality follows from the fact that for all $u \in A, y'(u) \geq 0$ and $y(\Gamma^-(u)\cap T) \geq 1$ (otherwise there would be more than $p-1$ members of $\Gamma^-(u)$ in S). Since some term is at least as large as the weighted average, the lemma follows. □

Lemma 6.5.5.

$$|S| \leq k^2 \ln k \ .$$

Proof. Due to Lemma 6.5.4, the vertex v chosen in every application of the greedy method has $y'(\Gamma^+(v)) = y'(A(v)) \geq y'(A)/y(T)$. In this proof we focus on one iteration at a time and let A' stand for the active vertices *after* the iteration and A for those before. Now,

$$\begin{aligned}
y'(A') &= y'(A) - y(A(v)) \\
&\leq y'(A) - y'(A(v))/p \\
&\leq y'(A) - \frac{y'(A)}{y(T)\cdot p} \\
&\leq y'(A) - \frac{y'(A)}{kp} \\
&= y'(V)(1 - \frac{1}{kp})
\end{aligned}$$

since $y(B) \geq y'(B)/p$ for any set B and $y(T) \leq k$. Initially, $y'(V) = kp$, so $y'(V) < p$ after at most $kp \ln k$ iterations. Since $p \leq k$—otherwise no solution exists—we have $|S| \leq k^2 \ln k$. ☐

Repeatedly applying the greedy procedure for constrained set multicover, this time for $O(\log^* k)$ iterations, we get $2k$ centers that cover all active vertices within $O(\log^* k)$. Alternatively, we could carry out $O(\log k)$ iterations and stick to just k centers.

6.6 Inapproximability Results

In this section we give inapproximability results for the asymmetric versions of the k-center problem with outliers, the priority k-center problem, and the k-supplier problem. These problems all admit constant factor approximation algorithms in the symmetric case.

6.6.1 Asymmetric k-Center with Outliers

A disadvantage of the standard k-center problem is that a few distant clients can force centers to be located in isolated places. This situation is averted in the following variant problem, in which a small subset of clients may be denied service, and some points are forbidden from being centers.

Definition 6.6.1 (k-Center with Outliers and Forbidden Centers)**.** Find a set $S \subseteq C$, where C is the set of vertices allowed to be centers, such that $|S| \leq k$ and S covers at least p nodes, with minimum radius.

Theorem 6.6.2. *For any function $\alpha(n)$, the asymmetric k-center problem with outliers (and forbidden centers) cannot be approximated within a factor of $\alpha(n)$ in polynomial time, unless P $=$ NP.*

Proof. We reduce instance $\langle U, \mathcal{S}, k \rangle$ of max coverage to our problem. Construct vertex sets A and B so that for each set $S \in \mathcal{S}$ there is $v_S \in A$, and for each element $e \in U$ there is $v_e \in B$. From each vertex $v_S \in A$, create an edge of unit length to vertex $v_e \in B$ if $e \in S$. Let $p = |B| + k$.

If the optimum value of the max coverage instance is $|\mathcal{U}|$, then the k nodes in A that correspond to some optimal sub-collection will cover p nodes within radius 1. Our $\alpha(n)$-approximation algorithm will thus return k centers that cover p nodes in some *finite* distance. If the maximum coverage with k sets is less than $|\mathcal{U}|$, then the optimum covering radius for p nodes, using k centers, is infinite. Since

our approximation can distinguish between these two cases, the approximation problem must be NP-complete. □

Note that the proof never relied on the fact that the B vertices were forbidden from being centers—setting p to $|B| + k$ ensured this.

6.6.2 Asymmetric Priority k-Center

Perhaps some demand points have a greater need for centers to be closer to them than others. This situation is captured by the priority k-center problem, in which the distance to a demand vertex is effectively enlarged by its priority. Note that the triangle inequality still holds for the original distances.

Definition 6.6.3 (Priority k-Center)**.** Given a priority function $p : V \rightarrow \mathbb{Q}^+$ on the vertices, find $S \subseteq V$, $|S| \leq k$, that minimizes R so that for every $v \in V$ there exists a center $c \in S$ for which $p_v d_{cv} \leq R$.

Theorem 6.6.4. *For any polynomial time computable function $\alpha(n)$, the asymmetric k-center problem with priorities cannot be approximated within a factor of $\alpha(n)$ in polynomial time, unless P $=$ NP.*

Proof. The construction of the sets A and B is similar to the proof of Theorem 6.6.2. Again, we have the unit length set-element edges from A to B, but this time we make the set A a complete digraph, with edges of length ℓ, as in Figure 6.1. Give the nodes in set A priority 1 and the nodes in set B priority ℓ.

If there exists a collection of k sets that cover all elements, then there exist k elements of A that cover every vertex in A and B within radius ℓ. If there do not exist k such sets, then the optimal covering radius using k centers is $\ell^2 + \ell$: some vertex in B is at distance $\ell + 1$ from its nearest center and has priority ℓ. Since we can set ℓ equal $\alpha(n)$, our algorithm can distinguish between the two types of max coverage instance. Therefore the approximation problem is NP-complete. □

6.6.3 Asymmetric k-Supplier

In the k-supplier problem the vertex set is segregated into suppliers and customers. Only supplier vertices can be centers and only customer vertices need to be covered.

Definition 6.6.5 (k-Supplier)**.** Given a set of suppliers Σ and a set of customers C, find a subset $S \subseteq \Sigma$ that minimizes R such that S covers C within R.

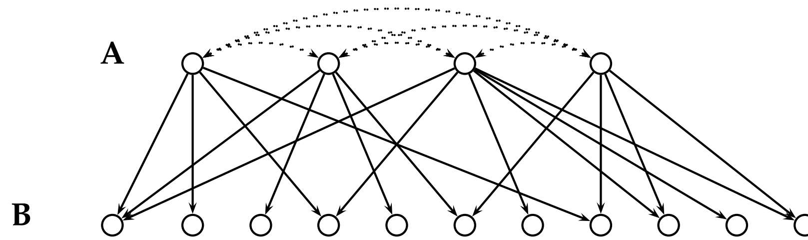


Figure 6.1: k-center with priorities. Solid lines have length 1, dotted lines length ℓ.

Theorem 6.6.6. *For any function $\alpha(n)$, the asymmetric k-supplier problem cannot be approximated within a factor of $\alpha(n)$ in polynomial time, unless P = NP.*

Proof. By a reduction from the max coverage problem similar to the proof of Theorem 6.6.2. □

6.7 Asymmetric k-Center with Minimum Coverage

In this section we give approximation algorithms and inapproximability results for various asymmetric k-center with minimum coverage problems. These problems have been studied by Lim *et al.* [87] in the symmetric setting. In k-center with minimum coverage, each center is required to serve a minimum of clients.

Lim *et al.* studies the following problems:

- The *q-all-coverage k-center* problem, where each center must cover at least q vertices (including itself).
- The *q-coverage k-center* problem, where each center must cover at least q non-center nodes.
- The *q-coverage k-supplier* problem, where each supplier must cover at least q demands.

Furthermore, Lim *et al.* studied both the weighted and the priority versions of these problems.

For the q-all-coverage k-center problem Lim *et al.* gave an 2-approximation algorithm, and an 3-approximation algorithm for the weighted and priority versions of the problem. For the q-coverage k-center problem they provided an 2-approximation algorithm, and an 4-approximation algorithm for the weighted and priority versions of the problem. For the q-coverage k-supplier problem they gave an 3-approximation algorithm for both the basic, the weighted, and the priority version.

6.7.1 Inapproximability Results

Since the q-all-coverage k-center problem and the q-cover k-center problem are generalizations of the k-center problem (set $q = 1$ and $q = 0$, respectively), the priority version of these problems cannot be approximated within any factor in the asymmetric case unless P = NP. Since the q-coverage k-supplier problem is a generalization of the k-supplier problem ($q = 0$), it cannot be approximated within any factor in the asymmetric version unless P = NP.

6.7.2 q-All-Coverage k-Center

In this section we give a $O(\log^* n)$-approximation algorithm for the asymmetric q-all-coverage k-center problem.

Definition 6.7.1 (q-All-Coverage k-Center)**.** Given $G = (V, E)$, a complete graph with nonnegative (but possibly infinite) edge costs, and a positive integer k, find a set S of k vertices, called *centers*, with minimum covering radius, such that each center covers at least q vertices.

Our algorithm is based on Panigrahy and Vishwanathan's technique for the asymmetric k-center problem[96]. As before, the algorithm guesses R_{OPT}, and works in the bottleneck graph $G_{R_{\mathsf{OPT}}}$.

First we note that if we are in the right bottleneck graph any node either has out-degree at least $q-1$ or is covered by a node with out-degree at least $q-1$. We will modify the definition of a CCV to reflect this.

Definition 6.7.2. Let $\Gamma^{q-}(v) = \{u \mid u \in \Gamma^-(v) \text{ and } \deg(u) \geq q-1\}$. Node v is a CCV$_q$ if $\Gamma^{q-}(v) \in \Gamma^+(v)$.

It follows immediately, that if v is a CCV$_q$ then v covers a center in the optimal solution.

In the halve phase we find a CCV$_q$ v, include it in the set of centers, mark every vertex in $\Gamma_2^+(v)$ as covered, and repeat until no CCV$_q$s remain unmarked. The CCV$_q$ property ensures that, as each CCV$_q$ is found and vertices are marked, the unmarked portion of the graph can be covered with one fewer center. Hence if k'' CCV$_q$s are obtained, the unmarked portion of the graph can be covered with $k' = k - k''$ centers.

We will prove that this unmarked portion, CCV$_q$-free, can be covered with only $k'/2$ centers if we use radius 5 instead of 1. That is to say, $k'/2$ centers suffice in the graph $G^5_{R_{\mathsf{OPT}}}$.

Panigrahy and Vishwanathan [96] show the following lemma.

Lemma 6.7.3 (Panigrahy and Vishwanathan [96])**.** *Let $G = (V, E)$ be a digraph with unit edge costs. Then there is a subset $S \subseteq V$, $|S| \leq |V|/2$, such that every vertex with positive indegree is reachable in at most 2 steps from some vertex in S.*

We can use this to show a lemma analog to that for the standard asymmetric k-center problem.

Lemma 6.7.4. *Consider a subset $A \subseteq V$ that has a cover consisting of vertices of size k, but no CCV$_q$s. Assume there exists a set C_1 that 3-covers exactly $V \setminus A$, and every vertex in C_1 3-covers at least q vertices. Then there exists a set of vertices S of size $k/2$ that, together with C_1, 5-covers A, and every vertex in S covers at least q vertices.*

Proof. Let U be a subset of the optimal centers that covers A. We call $u \in U$ a *near* center if it can be reached in 4 steps from C_1, and a *far* center otherwise. Since C_1

5-covers all of the nodes covered by near centers, it suffices to choose S to 4-cover the far centers, so that S will 5-cover all the nodes they cover. We also need to ensure that any vertex in S 5-covers at least q vertices.

Define an auxiliary graph H on the (optimal) centers U as follows. There is an edge from x to y in H if and only if x 2-covers y in G (and $x \neq y$). The idea is to show that any far center has positive indegree in H. As a result, Lemma 6.7.3 shows there exists a set $S \in U$ with $|S| \leq k/2$ such that S 2-covers the far centers in H, and thus 4-covers them in G. Since $S \in U$ and U is the set of optimal centers, all vertices in S covers at least q vertices.

Let u be any far center: note that $u \in A$. Since A contains no CCV$_q$s, there exists $v \in \Gamma^{q-}(u)$ that is not covered by u. Since $u \notin \Gamma_4^+(C_1)$, $v \notin \Gamma_3^+(C_1)$, and thus $v \in A$ (since everything not 3-covered by C_1 is in A). Thus there exists a vertex $w \in U$, which is not u, but might be v, that covers v and therefore 2-covers u. Hence u has positive indegree in H. □

In the augment phase we use the greedy set cover algorithm, which has approximation guarantee $1 + \ln(n/OPT)$, where n is the number of elements. Only nodes that have degree at least $q-1$ in the bottleneck graph G_i before the removal of CCVs are possible centers. It is easy to check wether it is possible to cover the graph with only these nodes. If not then we are not in the right bottleneck graph.

Applying the greedy algorithm for set cover $O(\log^* n)$-times we get down to $2k'$ centers.

Lemma 6.7.5. *Given $A \subseteq V$, such that A has a cover of size k, where all centers in the cover covers at least q vertices, and a set $C_1 \subseteq V$ that covers $V \setminus A$, where all centers in C_1 covers at least q vertices. We can then find in polynomial time a set of centers of size most $2k$ that, together with C_1, covers A (and hence V) within a radius of $O(\log^* n)$, such that all centers cover at least q vertices.*

Proof. We will be apply the greedy set cover algorithm recursively. The initial set of centers S_0 is constructed as follows. For any vertex v for which $\Gamma^+(v) \cap A$ is non-empty, and which has out-degree at least $q - 1$ construct a set containing $\Gamma^+(v)$.

The greedy algorithm for set cover has approximation guarantee $O(\log(n/k))$, which is less than $\log_{1.5}(n/k)$ when $n \geq 2k$. Applying this algorithm thus results in a set S_1 that covers A and has size at most $O(k \log(n/k))$.

The set C_1 covers $S_1 \setminus A$, so we need only consider $A_1 = S_1 \cap A$. Remove all sets with $\Gamma^+(v) \cap A = \emptyset$. We apply the greedy set cover algorithm again to obtain a set S_2 of size at most

$$k(\log_{1.5}(|A_1|/k)) = k(\log_{1.5}(k \log_{1.5}(n/k)/k)) = k(\log_{1.5}(\log_{1.5}(n/k))) \ .$$

We continue this procedure and note that at the ith iteration we have

$$|S_i| \leq k \log_{1.5}(|S_{i-1}|/k) \; .$$

By induction, after $O(\log^* n)$ iterations the size of our solution set, S_i, is at most $2k$. □

We can now combine the results to get

Theorem 6.7.6. *The q-all-coverage k-center problem can be approximated within a factor of $O(\log^* n)$ in polynomial time.*

Proof. Guess the optimum radius, R_{OPT}, and work in the bottleneck graph $G_{R_{\mathsf{OPT}}}$. Initially, the active set A is V. Repeat the following as many times as possible: Pick a CCV$_q$ v in A, add v to our solution set of centers, and remove the set $\Gamma_2^+(u)$ from A. Since v is covered by an optimum center in $\Gamma^-(v)$, and this optimum center lies in $\Gamma^+(v)$, $\Gamma_2^+(v)$ includes everything covered by it.

Let C_1 be the centers chosen in this first phase. We know the remainder of the graph, A, has a cover of total size $k' = k - |C_1|$.

Lemma 6.7.4 shows that we can cover the remaining uncovered vertices with at most $k'/2$ centers if we use covering radius 5. Let the active set A be $V \setminus \Gamma_5^+(C_1)$, and recursively apply the greedy algorithm as described in the proof of Lemma 6.4.5 on the graph $G^5_{R_{\mathsf{OPT}}}$. As a result, we have a set of size k that covers A within radius $O(\log^* n)$. □

6.7.3 Approximation of q-Coverage k-Center

Definition 6.7.7 (q-Coverage k-Center)**.** Given $G = (V, E)$, a complete graph with nonnegative (but possibly infinite) edge costs, and a positive integer k, find a set S of k vertices, called *centers*, with minimum covering radius, such that each center covers at least q vertices in $V \setminus S$.

We use the algorithm from the previous section to find a set S of centers for the $(q + 1)$-all-coverage k-center problem. First we note that the centers found in the halve phase all cover at least q non-centers, since when we pick a CCV$_{q+1}$ as v a center we mark $\Gamma_2^+(v)$ as covered and thus none of these at least q vertices will later be picked as centers. The potentially problematic centers are the centers found in the augment phase. These centers all cover q vertices, but they might not cover q non-centers.

Lemma 6.7.8. *Given a set of centers S that covers all vertices, and for all $v \in S$ v covers at least q vertices. We can find a set $S' \subseteq S$ of centers that 2 covers all vertices, such that each center $v \in S'$ 2-covers at least q vertices from $V \setminus S$.*

Proof. Let P be the set of problematic centers, i.e., centers that do not cover q non-centers. To construct the set S' repeat the following as long as P is non-empty: Pick a center v from P. Remove all vertices $\Gamma^+(v) \cap S$ except v from S, and remove all vertices in $\Gamma^-(v) \cap P$ from P. When P is empty set $S' = S' \cup S$.

Let v be a center in S'. We need to show that v 2-covers at least q non-center vertices. If v was never in P then clearly v covers at least q non-center vertices, as $S' \subseteq S$. Assume v was initially in P. Then either v was picked or some center in $\Gamma^+(v)$ was picked. If v was picked, then since v covers at least q vertices and all vertices covered by v now are non-centers, v covers at least q non-centers. If some center $u \in \Gamma^+(v)$ was picked then as u covers at least q non-center v 2-covers at least q vertices.

We must now show that S' 2-covers all vertices. Assume $v \in S$ was picked. Since all vertices in $\Gamma^-(v)$ are removed from P, and thus v remains a center and thus $v \in S'$. Assume $v \in S$ was not picked by the procedure. If $v \notin S'$ then it must be the case that some vertex $u \in Gamma^-(v)$ was picked. As just argued $u \in S'$. All vertices in $\Gamma^+(v)$ are 2-covered by u. Therefore, S' 2-covers all vertices covered by S. □

Using Lemma 6.7.8 we can now show

Theorem 6.7.9. *The q-coverage k-center problem can be approximated within factor $O(\log^* n)$ in polynomial time.*

Proof. Apply the algorithm from the previous section to find a set S of centers for the $(q+1)$-all-coverage k-center problem. Let α be the actual approximation ratio obtained by the $(q+1)$-all-coverage k-center algorithm on this instance.

Now apply the procedure from Lemma 6.7.8 on S in the graph $G^{\alpha}_{R_{\mathsf{OPT}}}$. This gives us a set of centers that 2α-covers all the vertices, and all the centers 2α-covers at least q non-center vertices. Since $\alpha = O(\log^* n)$ this gives an $O(\log^* n)$-approximation. □

6.7.4 Weighted Versions

We can approximate the weighted version of the q-all-coverage k-center problem and the q-coverage k-center problem with a factor of $O(\log^* n)$ by adapting our algorithm for the weighted set cover to the approaches above.

Chapter 7

Finite Capacity Dial-a-Ride

We study hardness of approximation of the *preemptive Finite Capacity Dial-a-Ride problem*. Let k be the capacity of the vehicle and N the number of nodes in the input graph. We show that the problem has no $\min\{O(\log^{1/4-\varepsilon} N), k^{1-\varepsilon}\}$-approximation algorithm for any $\varepsilon > 0$ unless all problems in NP can be solved by randomized algorithms with expected running time $O(n^{\mathrm{polylog} n})$.

7.1 Finite Capacity Dial-a-Ride

In the *Finite Capacity Dial-a-Ride problem*—or *Dial-a-Ride* for short—the input is a metric space, a set of objects, where each object d_i specifies a source s_i and a destination t_i, and an integer k—the capacity of the vehicle used for making the deliveries. The goal is to compute a shortest tour for the vehicle in which all objects can be delivered to their destinations (from their sources) while ensuring that the vehicle carries at most k objects at any point in time. There are two variants of the problem: the *non-preemptive* case, in which an object once loaded on the vehicle it stays on until it is delivered to its destination, and the *preemptive* case in which an object may be dropped at intermediate locations and then picked up later by the vehicle and delivered.

The Dial-a-Ride problem generalizes the traveling salesman problem (TSP) even for $k = 1$ and is therefore NP-hard. By placing an object and its destination in each vertex in the TSP instance yields an instance of the Dial-a-Ride problem. In this instance the vehicle has to simply find a shortest path tour that visits all the vertices, since any object that is picked up can be delivered immediately to its destination point in the same location.

7.1.1 Applications

The Dial-a-Ride problem has several practical applications [88] such as transportation of elderly and/or disabled persons, certain courier services, and shared taxi services [110]. Although single-vehicle Dial-a-Ride systems are not very common, single-vehicle Dial-a-Ride algorithms are used as subroutines in large scale multi-vehicle Dial-a-Ride environments and is therefore important.

7.1.2 Previous Results

Guan [62] proved that the unit-capacity preemptive case is NP-hard for trees when $k \geq 2$. Frederickson and Guan [55] showed that the unit-capacity non-preemptive case is NP-hard even on trees. For this case Frederickson *et al.* [56] gave an algorithm with approximation factor 1.8 on general graphs. We have not been able to find any references to a $O(1)$-approximation algorithm for the preemptive Dial-a-Ride problem with unit capacity in the literature. However, it is simple to construct a 3-approximation algorithm for the preemptive case with unit capacity (see Section 5.5.1).

Let N denote the number of nodes in the input graph, i.e., the number of points that are either sources or destinations. The first non-trivial approximation algorithms for the Finite Capacity Dial-a-Ride problem for general k were given by Charikar and Raghavachari [28]. As noted by Charikar and Raghavachari an $O(k)$-approximation algorithm can be obtained by taking the $O(1)$-approximation algorithm for the unit-capacity case ($k = 1$).

As noted by Charikar and Raghavachari [28] using the $O(1)$-approximation algorithm in the general case when $k > 1$ gives a $O(k)$-approximation algorithm. To see this first note that any solution using capacity one is a valid solution. That is, $\mathsf{OPT}_k \leq \mathsf{OPT}_1$, where OPT_i is the value of the optimal solution to the problem using a vehicle with capacity i. We will also use OPT_i to denote the actual solution. Given a solution SOL_k to the problem using a vehicle with capacity k we can construct a solution SOL_1 to the problem using a vehicle of capacity 1 of at most k times the cost. Follow the same route as SOL_k when the first object is picked up in SOL_k pick up this object and deliver it to the vertex where it is dropped off in SOL_k. Then go back to the point were the object was picked up, and keep following the route of SOL_k until the next object is picked up. Do the same for this object and so on. If an edge is traversed c times in SOL_k then it is traversed at most $2k \cdot c$ times in SOL_1. Therefore $\mathsf{SOL}_1 \leq 2k \cdot \mathsf{SOL}_k$, and thus $\mathsf{OPT}_1 \leq 2k \cdot \mathsf{OPT}_k$.

We note that there also is a trivial $\frac{3N}{k}$-approximation algorithm. Let us first consider the case when $k = N$. Then all objects can be in the vehicle at the same

time. We can construct a tour by first taking a TSP tour on the sources, picking up all objects, and then a TSP tour on the destinations, delivering all objects. Both these TSP tours are a lower bound on value of the optimal solution as the vehicle will have to visit all of them. Using the $3/2$-approximation [32] to construct the TSP tours we get a tour of length $2 \cdot \frac{3}{2} \cdot \mathsf{OPT}_{TSP} \leq 3 \cdot \mathsf{OPT}$. For $k = N/c$ we do this c times each time picking up and delivering N/c objects. This gives an algorithm with approximation factor $3c = 3N/k$.

Preemptive Charikar and Raghavachari gave a 2-approximation algorithm for trees for the preemptive Dial-a-Ride problem. Using the results on probabilistic approximation of metric spaces by tree metrics [48] this gives an $O(\log N)$-approximation for arbitrary metrics.

Non-Preemptive For the non-preemptive Dial-a-Ride problem, Charikar and Raghavachari gave an $O(\sqrt{k})$-approximation algorithm for special instances on height-balanced trees. Using the results on probabilistic approximation of metric spaces by tree metrics [48] this gives an $O(\sqrt{k} \log N)$-approximation algorithm for arbitrary metrics. When the points lie on a line Charikar and Raghavachari note that they have a 2-approximation algorithm.

Relation between Preemptive and Non-Preemptive Charikar and Raghavachari showed that the ratio of the cost of the optimal non-preemptive solution to the cost of the optimal preemptive solution can be as large as $\Omega(k^{2/3})$.

7.1.3 Our Results and Techniques

We show that there is no $\min\{O(\log^{1/4-\varepsilon} N), k^{1-\varepsilon}\}$-approximation algorithm for the preemptive Finite Capacity Dial-a-Ride problem for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\mathrm{polylog} n})$[1].

To our knowledge, the TSP lower bound was the best known so far.

7.1.4 Buy-at-Bulk and SumFiber-ChooseRoute

Our results rely on the hardness results for the two network design problems Buy-at-Bulk and SumFiber-ChooseRoute.

[1] $\mathsf{ZPTIME}(n^{\mathrm{polylog} n})$ is the class of problems solvable by a randomized algorithm that always returns the right answer and has expected running time $O(n^{\mathrm{polylog} n})$, where n is the size of the input.

Buy-at-Bulk In the *Buy-at-Bulk* problem we are given an undirected network $\mathcal{N}$, with lengths l_e on the edges, and a set $\{(s_i, t_i)\}$ of source-destination pairs. Each source-destination pair (s_i, t_i) has an associated demand δ_i. Each edge e has a cost function $f_e(x)$, which is a function of the amount of demand using edge e. The function f_e is subadditive[2], and $f_e(0) = 0$. Since the cost function is subadditive it exhibits *economies of scale*.

The goal is to route all demands δ_i from their source s_i to their destination t_i minimizing the total cost. The demands are unsplittable, i.e., demand δ_i must follow a single path from s_i to t_i. The cost of an edge e is $f_e(x_e)$ where x_e is the amount of demand routed through e in the solution. The total cost of the solution is then

$$\sum_e f_e(x_e) l_e.$$

In this chapter the cost function f_e is the same for all edges. This is also known as the *uniform* Buy-at-Bulk problem.

The Buy-at-Bulk problem includes as a special case the Steiner tree problem and is therefore NP-hard. Let N be the number of nodes in the network. For the uniform Buy-at-Bulk problem the best known approximation algorithm achieves an approximation factor of $O(\log N)$ due to the work of Awerbuch and Azar [18], and Fakcharoenphol *et al.* [48]. The algorithm uses the result from [48] to turn the metric given by the shortest paths in the network into a tree. In a tree there is only one way to route the flows.

Andrews [9] shows that there is no $O(\log^{\frac{1}{4}-\varepsilon} N)$-approximation algorithm for the uniform Buy-at-Bulk problem for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\text{polylog} n})$.

SumFiber-ChooseRoute In the *SumFiber-ChooseRoute* problem we are given an undirected network $\mathcal{N}$, with lengths l_e on the edges, and a set $\{(s_i, t_i)\}$ of source-destination pairs. Each source-destination pair (s_i, t_i) corresponds to a demand δ_i. Each demand requires bandwidth equivalent to one wavelength. Each fiber can carry k wavelengths, and the cost of deploying x fibers on edge e is $x \cdot l_e$. The problem is to specify a path from s_i to t_i for all demands δ_i, and a wavelength for the demand λ_i, minimizing the total cost. Let $f_e(\lambda)$ be the number of demands assigned to wavelength λ that are routed through edge e. Then $\max_\lambda f_e(\lambda)$ is the number of fibers needed on edge e. Thus the total cost of the solution is

$$\sum_e l_e \max_\lambda f_e(\lambda).$$

[2] $f_e(x + y) \leq f_e(x) + f_e(y)$.

The SumFiber-ChooseRoute problem is NP-hard. Andrews and Zhang [10] give an $O(\log N)$-approximation algorithm for SumFiber-ChooseRoute, and show that there is no $O(\log^{\frac{1}{4}-\varepsilon} N)$-approximation algorithm for the problem for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\text{polylog} n})$.

7.1.5 Our Techniques

Andrews and Zhang show that their hardness result for SumFiber-ChooseRoute using a network constructed from an interactive 2-prover system for MAX3SAT. Andrews [9] states that this network can be used to show the same results for the uniform Buy-at-Bulk network design problem with cost function $f(x) = L + x$. The function $f(x)$ is asymptotically the same as the function $h(x) = \lceil \frac{x}{k} \rceil$ when $L = k - 1$. Using the almost the same construction we show that Buy-at-Bulk with cost function $h(x)$ has no $O(\log^{\frac{1}{4}-\varepsilon} n)$-approximation algorithm for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\text{polylog} n})$, when $k = \Omega(\log^{\frac{1}{4}+\frac{7\varepsilon}{11}} N)$. By changing some of the parameters in the network construction we are also able to show that the problem is not approximable within a factor of $k^{1-\varepsilon}$ for any $\varepsilon > 0$ when $k < \log^{\frac{1}{4}} N$.

We then show the same hardness results for Dial-a-Ride by showing a relation between this problem and the Buy-at-Bulk problem with cost function $h(x)$.

7.2 Relation between Buy-at-Bulk and Dial-a-Ride

The following lemma shows a connection between the Buy-at-Bulk problem with cost function $h(x) = \lceil \frac{x}{k} \rceil$ and the Dial-a-Ride problem.

Lemma 7.2.1. *Let OPT_B be the value an optimal solution to the Buy-at-Bulk instance B with source destination pairs S in graph G and cost function $h(x) = \lceil \frac{x}{k} \rceil$, and let OPT_D be the value an optimal solution to the preemptive Dial-a-Ride instance D with source-destination pairs S in the graph G. Then*

$$\mathsf{OPT}_B \leq \mathsf{OPT}_D.$$

Proof. We will abuse notation and let OPT_i stand for both the value of the optimal solution and the solution itself. We can turn the optimal solution to the Dial-a-Ride instance OPT_D into a solution to Buy-at-Bulk instance B the following way: We route a demand δ_i from its source s_i to its destination t_i by the same edges as item δ_i passes in the Dial-a-Ride solution. Clearly, this is a valid solution. The cost of this solution is no larger than the cost of the Dial-a-Ride solution. To see this consider an edge in our solution to the Buy-at-Bulk instance. This edge is

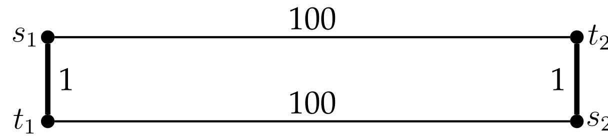


Figure 7.1: The value of the optimal solution to Buy-at-Bulk is 2 (thick edges), whereas the value of the optimal solution to Dial-a-Ride is 202.

only used by items that are in the vehicle when it is crossing this edge in the Dial-a-Ride solution. Thus the Dial-a-Ride solution must have used this edge at least $\lceil \frac{x_e}{k} \rceil$ times where x_e is the number of items using the edge. This is at least the same as the Buy-at-Bulk solution will have to pay for this edge. $\square$

Since the optimal solution to the Buy-at-Bulk instance might be disconnected, there is in general no way to turn the solution to the Buy-at-Bulk instance B into a solution to the Dial-a-Ride instance D at a cost bounded in terms of OPT_B (see Figure 7.1. But on the network used to construct the hardness result for Buy-at-Bulk we can show that in the case were the MAX3SAT instance ϕ is satisfiable it is possible to turn the solution to the Buy-at-Bulk instance into a solution to the Dial-a-Ride instance at cost at most $7 \cdot \mathsf{OPT}_B$.

7.3 The Network

In this section we describe the network by Andrews and Zhang [10] used to show the hardness. The network is constructed randomly from an interactive proof system for MAX3SAT.

7.3.1 Interactive Proof Systems

To show the hardness of Buy-at-Bulk Andrews and Zhang construct a reduction using the Raz verifier for MAX3SAT(5) [102].

A MAX3SAT(5) formula has n variables and $5n/3$ clauses. Each variable appears in exactly 5 distinct clauses and each clause contains exactly 3 literals. The MAX3SAT(5) problem is to find an assignment that maximizes the number of satisfied clauses. A MAX3SAT(5) formula is called a *yes-instance* if it is satisfiable, and a *no-instance* if no assignment satisfies more than a $1 - \varepsilon$ fraction of the clauses for some fixed constant $\varepsilon > 0$. It follows from the PCP-theorem [15] that it is NP-hard to distinguish between yes-instances and no-instances.

A Raz-verifier is an *interactive two-prover system*. An interactive two-prover system for MAX3SAT(5) consists of a polynomial time *verifier* with access to a

source of randomness and two computationally unbounded *provers*. The verifier sends a polynomial size query to each prover and receives a polynomial size answer. The provers try to convince the verifier that the formula is satisfiable (yes-instance). The provers cannot communicate with each other and are restricted to see only the queries addressed to them.

A Raz-verifier for MAX3SAT(5) with ℓ repetitions is defined as follows. A verifier interacts with two provers, a clause prover (prover 0) and a variable prover (prover 1). Given a MAX3SAT(5) formula ϕ, the verifier sends prover 0 a clause query that consists of ℓ clauses $c_1, \ldots, c_\ell$ chosen uniformly at random. To prover 1 it sends a variable query that consists of one variable v_i chosen uniformly at random from each of the ℓ clauses. Prover 0 sends back the assignment of every variable in the ℓ clauses $c_1, \ldots, c_\ell$, and prover 1 sends back the assignment of the ℓ variables $v_1, \ldots, v_\ell$. The verifier *accepts* ϕ if all the ℓ clauses are satisfied and the two provers give consistent assignments to the ℓ variables $v_1, \ldots, v_\ell$. The verifier *rejects* ϕ otherwise.

If ϕ is satisfiable the verifier accepts with probability 1. If ϕ is unsatisfiable then regardless of how the provers answer the verifier accepts with very low probability. We call this probability the *error probability* and denote it by η.

Proof System Parameters Let R be the random bits, Q_i the random query sent to prover i, and A_i the answer returned by prover i. We will use lowercase letters to denote specific values of these strings. Each random string r uniquely identifies a pair of queries q_0 and q_1. Each query may have many different answers. We say $a \in q$ if a is an answer to query q. We assume that the verifier appends the name of the prover to the query and the provers append the query name to its answer string. This way, an interaction is uniquely identified by the triple (r, a_0, a_1). If the verifier accepts the answers a_0 and a_1 from the provers we say that (r, a_0, a_1) is an *accepting interaction*. Note that two different random string might result in the same prover-0 query (or prover-1 query), but in that case they will result in different prover-1 (prover-0) queries.

Let $m(Q_i)$ denote the number of distinct possible values of Q_i. By padding random bits, we can assume,

$$m(Q_0) \leq m(Q_1) < 2m(Q_0).$$

For a suitable choice of ℓ, the Raz verifier has the following properties. Here $|x|$ denotes the number of bits in the string x.

1.

$$\begin{aligned} |R| &= O(\log^2 n) \\ |Q_i| &= O(\log^2 n) \\ |A_i| &= O(\log^2 n) \\ \eta &= 2^{-\Omega(\log n)}. \end{aligned}$$

2. All possible queries occur with equal probability. Formally, for each i and for any $q \in \{0,1\}^{|Q_i|}$: $Pr[Q_i = q] \in \{0, 1/m(Q_i)\}$.

7.3.2 Description of the Network

Andrews and Zhang [10] take the two prover system for MAX3SAT and turn it into an instance of the SumFiber-ChooseRoute problem. The construction is very similar to the one used by Andrews [9] to show hardness of Buy-at-Bulk. For each demand they define a set of *canonical paths* on which the demand can be carried. These canonical paths correspond to accepting interactions and are short paths directly connecting the source and the destination. They show that if ϕ is satisfiable then the optimal solution to the instance of SumFiber-ChooseRoute has small cost, and if ϕ is unsatisfiable then the optimal solution has high cost. More precisely, the cost if ϕ is unsatisfiable is a factor of γ more than if ϕ is satisfiable for $\gamma = O(\log^{\frac{1}{4}-\varepsilon} n)$. Hence if there were an α-approximation for SumFiber-ChooseRoute with $\alpha < \gamma$, then we would be able to determine if ϕ was satisfiable.

The Basic Network $\mathcal{N}_0$

Andrews and Zhang first construct a basic network $\mathcal{N}_0$, which is used as the base case in the random construction. Given an instance ϕ, first construct the two-prover interactive proof system. The proof system is turned into an instance of SumFiber-ChooseRoute as follows. For each possible answer a there is an *answer edge* (also denoted by a). For each random string r there is a source node s_r, a destination node t_r, and a demand d_r of one to be routed from s_r to t_r. For each accepting interaction (r, a_0, a_1) there is a canonical path p. This path starts at node s_r passes through a_0 and a_1 and ends at t_r. To make this possible we place edges between s_r and a_0, between a_0 and a_1, and between a_1 and t_r. The edge between a_0 and a_1 is refered to as a *center edge*, and the edge between s_r and a_0, and between a_1 and t_r as a *demand edge*. For each query q the answer edges $a \in q$ are grouped together (see Figure 7.2). The answer edges have length $h > 1$ and the other edges have length 1.

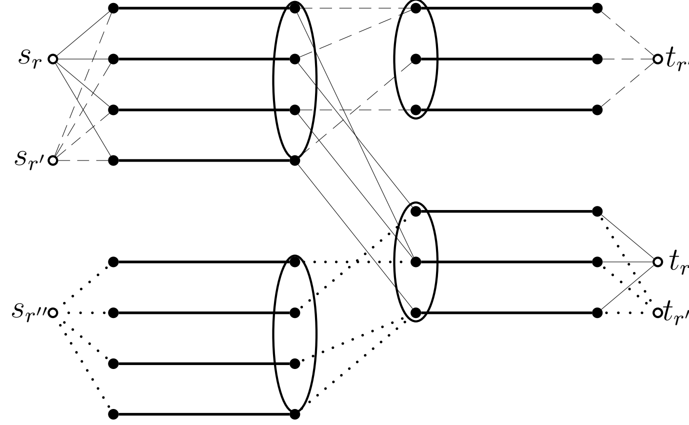


Figure 7.2: The basic network $\mathcal{N}_0$. For each of the three random strings r, r' and r'', four canonical paths corresponding to four accepting interactions, are shown (r solid, r' dashed, and r'' dotted). The long thick edges are the answer edges.

The Expanded Network $\mathcal{N}_2$

Before defining the final network $\mathcal{N}_2$ Andrews and Zhang first define a random network $\mathcal{N}_1$ in terms of $\mathcal{N}_0$ and two parameters X and Z. The network essentially replicates $\mathcal{N}_0$ in the vertical direction XZ times. Each answer edge a_0 (resp. a_1) of $\mathcal{N}_0$ has XZ copies, denoted by $a_{0,x,z}$ ($a_{1,x,z}$) where $0 \leq x < X$ and $0 \leq z < Z$. The center edges and the demands are created as follows. For each random string r, there are created X demands $d_{r,x}$ and X source and destination nodes $s_{r,x}$ and $t_{r,x}$, where $0 \leq x < X$. Each of the X demands $d_{r,x}$ routes one unit of flow from $s_{r,x}$ to $t_{r,x}$. For each accepting interaction (r, a_0, a_1), the demand $d_{r,x}$ has a canonical path that starts at $s_{r,x}$ passes through $a_{0,x',z'}$ and $a_{1,x'',z''}$ and ends at $t_{r,x}$. The answer edges $a_{0,x',z'}$ and $a_{1,x'',z''}$ are chosen randomly. More precisely, x' and x'' are chosen uniformly at random from the range $\{0, 1, \ldots, X-1\}$ and z' and z'' are chosen uniformly at random from the range $\{0, 1, \ldots, Z-1\}$. To make the canonical paths feasible, $\mathcal{N}_1$ has center edges connecting $a_{0,x',z'}$ and $a_{1,x'',z''}$, and edges connecting $s_{r,x}$ to $a_{0,x',z'}$, and $a_{1,x'',z''}$ to $t_{r,x}$.

The network $\mathcal{N}_1$ is used to construct the final network $\mathcal{N}_2$. The network $\mathcal{N}_2$ is essentially a concatenation of $\mathcal{N}_1$ in the horizontal direction Y times for some parameter Y, where each level is constructed randomly and independently. Each answer edge is indexed by $a_{0,x,z,y}$ (resp. $a_{1,x,z,y}$) where $y \in \{0, 1, \ldots, Y-1\}$. As in $\mathcal{N}_1$, there are created X demands $d_{r,x}$, $0 \leq x < X$, for each random string r. For each accepting interaction (r, a_0, a_1), the demand $d_{r,x}$ has a canonical path starting at $s_{r,x}$ followed by answer edges $a_{0,x,z,0}$ and $a_{1,x,z,0}$ chosen uniformly at

random at level $y = 0$. At each subsequent level y, the answer edges are chosen uniformly at random until the path ends at $t_{r,x}$. The center edges and demand edges are defined by the canonical paths. Each canonical path also requires an edge between each consecutive pair of levels. See Figure 7.3.
In Section 7.4 we will define the parameters used in the network.

7.4 Hardness of Buy-at-Bulk with Cost Function $\lceil \frac{x}{k} \rceil$

We will use the network by Andrews and Zhang to show hardness of Buy-at-Bulk with cost function $\lceil \frac{x}{k} \rceil$. The only change is in the parameters. We show two hardness results: one depending on N (the number of nodes in the graph) and one depending only on k.

7.4.1 Hardness with Dependence on N

To show that the Buy-at-Bulk problem with cost function $\lceil \frac{x}{k} \rceil$ is hard to approximate within a factor of $O(\log^{1/4-\varepsilon} N)$ for any $\varepsilon > 0$ we change one of the parameters used by Andrews and Zhang.

Parameters We now define the parameters. To define X, Y, and Z, we use the following auxiliary parameters, which are useful in the proofs.

- $\ell = \log^{\alpha} n$ for some constant α.
- $\sigma = \log^{\frac{\alpha}{4}} n$.

The parameters of the network $\mathcal{N}_2$ can now be defined:

- $Z = \frac{2^{|r|}}{k \min\{m(Q_0), m(Q_1)\}}$.
- $Y = \sqrt{\ell} = \log^{\frac{\alpha}{2}} n$.
- $X = (2^{6+|r|+|a_0|+|a_1|} YZ)^{2l+1} = 2^{O(\log^{\alpha+2} n)}$.
- $h = \frac{2^{|r|}}{(m(Q_0)+m(Q_1))Z}$.
- $k = \log^{\frac{\alpha}{4}+4} n$.
- $\eta = \frac{1}{\sigma^2 \log n}$.

Recall, that η is the error parameter of the proof system. The only parameter we have changed compared to Andrews and Zhang [10] is h. Andrews and Zhang has $h = \frac{2^{|r|}}{\log k(m(Q_0)+m(Q_1))Z}$.

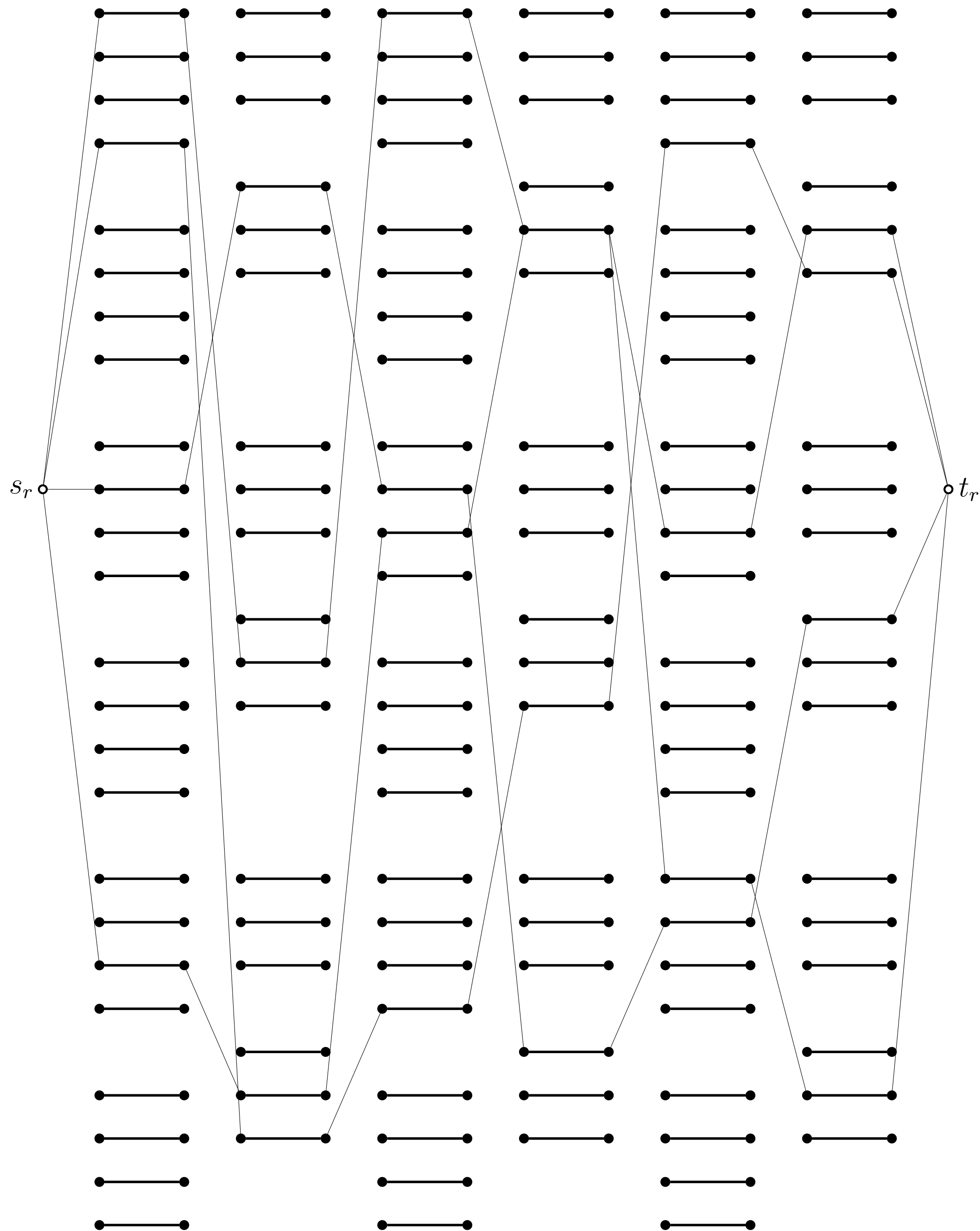


Figure 7.3: The network $\mathcal{N}_2$ with parameters $X = 1$, $Z = 3$, and $Y = 3$. For the random strings r four canonical paths corresponding to four accepting interactions, are shown.

Satisfiable Instances

We will now bound the value of the optimal solution to the Buy-at-Bulk instance when ϕ is satisfiable. An answer edge is said to be *bought* if any demand is routed

through it. We show,

Lemma 7.4.1. *If ϕ is satisfiable, then the Buy-at-Bulk instance has a solution of total cost at most* $2^{|r|}(2Y+1)X + (m(Q_0)+m(Q_1))hXYZ$.

Proof. Since ϕ is satisfiable there are two provers that always cause the verifier to accept. We route the demand on answer edge a if and only if for these two provers a is the answer to query q. For each string r there must be some accepting interaction (r, a_0, a_1) for which both a_0 and a_1 have been bought. Each of the demands $d_{r,x}$, for $0 \le x < X$, has one canonical path that corresponds to (r, a_0, a_1). The demand $d_{r,x}$ is routed along this path.

There are $2Y + 1$ length one edges on this path and thus the total number of edges of length one needed is at most $2^{|r|}(2Y+1)X$.

Now look at the cost of the answer edges. Consider an answer edge a that is bought in the solution. Assume without loss of generality that a is a prover-0 answer edge. There are $2^{|r|}X$ demands, and $\frac{2^{|r|}X}{m(Q_0)}$ of these can be routed through a since each prover-i query is equally likely. Each of these demands has XZ choices of answer edges, namely the XZ answer edges corresponding to the answer a. Hence, each demand has probability $\frac{1}{XZ}$ to pass through a, and the expected number of demands passing through a is thus

$$\frac{2^{|r|}X}{m(Q_0)} \cdot \frac{1}{XZ} \le k \ .$$

Since the cost function is $\lceil \frac{x}{k} \rceil$ the the expected cost of an answer edges is one. The expected total cost of the answer edges is therefore $XZY(m(Q_0)+m(Q_1))h$.

This solution has expected cost

$$2^{|r|}(2Y+1)X + (m(Q_0)+m(Q_1))hXYZ \ ,$$

and the cost of the optimal solution must therefore have cost no higher than that. □

Unsatisfiable Instances

In this section we lower bound the cost of the optimal solution to the Buy-at-Bulk problem when ϕ is unsatisfiable. The only parameter we have changed is h. Except Lemma 7, all of the lemmas of Andrews and Zhang [11] still hold without modification. In this section we give the new proof of Lemma 7. We first state some of the concepts and lemmas by Andrews and Zhang that we need in the proof.

Let p be a canonical path passing through answer edges $a_{0,x',z',y}$ and $a_{1,x'',z'',y}$. Then p is *routable* at level y if both $a_{0,x',z',y}$ and $a_{1,x'',z'',y}$ are bought. A demand is routable at level y if one of its canonical paths is routable at level y. The idea is to show that with high probability: 1) if many demands are routable on $3Y/4$ levels, then then the number of bought edges is high, and 2) if a demand routable at at most $3Y/4$ levels the length of its route is at least $Y^2/4$. That is, either we buy many of the expensive answer edges or many demands have a long route. In both cases the total cost is large.

Let S_y be the answer edges bought at level y. Let $E_y(a_i)$ be the set of XZ edges corresponding to answer a_i on level y, and let

$$w_y(a_i) = |E_y(a_i) \cap S_y|/(XZ) \ ,$$

i.e., $w_y(a_i)$ is the fraction of edges that are bought. A prover-i query is called *heavy* if

$$\sum_{a_i \in q_i} w_y(a_i) > 100\sigma.$$

A demand $d_{r,x}$ is *heavy* if the string r causes the verifier to send a heavy query to either prover 0 or prover 1. Let $B(y, S_y)$ be the bad event that $|S_y| \leq \sigma(m(Q_0) + m(Q_1))XZ$ and the number of non-heavy demands that are routable at level y is more than $2X\eta 2^{|r|}10^4\sigma^2$.

The following two lemmas are the same as in Andrew and Zhang's paper [11], only SumFiber-ChooseRoute is replaced by Buy-at-Bulk.

Lemma 7.4.2.

$$Pr[B(y, S_y) \textit{ for some } y, S_y] = o(1).$$

This probability is with respect to the random construction of the network.

Lemma 7.4.3. *Suppose that $B(y, S_y)$ does not occur for any y, S_y. Then for any solution to our Buy-at-Bulk instance that buys $\sigma(m(Q_0) + m(Q_1))XYZ/10$ answer edges, there are at most $(X2^{|r|})(77/375 + 24\eta 10^3\sigma^2)$ demands that are routable at more than $3Y/4$ levels.*

The Incidence Graph Andrews and Zhang defines a graph called the *incidence graph* $\mathcal{G}_2$, which has the following properties:

- For any route in $\mathcal{N}_2$ of length l the corresponding route in $\mathcal{G}_2$ is at most $4l$.
- Disjoint components in $\mathcal{N}_2$ map to disjoint components in $\mathcal{G}_2$.

This makes it possible to bound the length of the path used for routing a demand $d_{r,x}$ in $\mathcal{N}_2$ by considering the route in both $\mathcal{N}_2$ and in $\mathcal{G}_2$. The main idea is to show that $\mathcal{G}_2$ is a random graph and hence is unlikely to have many short cycles. This implies that most demands cannot be routed on canonical paths. Andrews and Zhang show the following lemma:

Lemma 7.4.4. *Demand $d_{r,x}$ must satisfy one of the following conditions:*

1. *The node $d_{r,x}$ in $\mathcal{G}_2$ is at distance at most ℓ away from a cycle of length ℓ.*
2. *The route for $d_{r,x}$ in $\mathcal{G}_2$ has length at least ℓ.*
3. *The route for $d_{r,x}$ in $\mathcal{N}_2$ has length at least $Y^2/4$.*
4. *Demand $d_{r,x}$ is routable at more than $3Y/4$ levels.*

Let $B(\mathcal{G}_2)$ be the bad event that more than X nodes in $\mathcal{G}_2$ are at distance at most ℓ from any cycle of length ℓ. Andrews and Zhang show the following lemma using the fact that $\mathcal{G}_2$ is constructed in a random fashion and a result similar to the Erdös-Sachs theorem that states that *high-girth*[3] graphs exist.

Lemma 7.4.5.

$$Pr[B(\mathcal{G}_2)] \leq \frac{1}{3}.$$

Bounding the Cost of the Solution Andrews and Zhang bound the cost of the solution in the case where ϕ is unsatisfiable using Lemma 7.4.2-7.4.5. The following lemma is identical to the one in used by Andrews and Zhang [10] except SumFiber-ChooseRoute is substituted by Buy-at-Bulk.

Lemma 7.4.6. *With probability $\frac{2}{3} - o(1)$, if the instance ϕ of 3SAT is unsatisfiable then the cost of any solution to our instance of Buy-at-Bulk is at least the minimum of V_1 and V_2, where*

$$\begin{aligned} V_1 &= \frac{\sigma h}{10}(m(Q_0) + m(Q_1))XYZ, \\ V_2 &= \frac{Y^2}{4k}\big((X2^{|r|})(1 - \frac{77}{375} - o(1)) - X\big). \end{aligned}$$

We are now ready to prove the main lemma of this section.

Lemma 7.4.7. *Let $\gamma = \log^{\frac{\alpha}{4}-5} n$. If there exists a γ-approximation algorithm, A, for Buy-at-Bulk with cost function $f(x) = \lceil \frac{x}{k} \rceil$, then there exists a randomized $O(n^{\text{polylog } n})$ time algorithm for 3SAT.*

[3]The girth of a graph is the length of the minimum cycle in the graph.

Proof. For any 3SAT instance ϕ we construct the network $\mathcal{N}_2$ from the two-prover system and then apply a γ-approximation algorithm A for Buy-at-Bulk. We declare ϕ to be satisfiable if and only if A returns a solution of cost at most $\gamma 2^{|r|}(3Y+1)X$.

If the 3SAT instance ϕ is satisfiable then by Lemma 7.4.1 there is a solution to our instance of Buy-at-Bulk of cost at most

$$\begin{aligned} 2^{|r|}(2Y{+}1)X+ & \\ (m(Q_0)+m(Q_1))hXYZ &= 2^{|r|}(2Y+1)X+\frac{(m(Q_0)+m(Q_1))2^{|r|}}{(m(Q_0)+m(Q_1))Z}XYZ \\ &= 2^{|r|}(2Y+1)X+2^{|r|}XY \\ &= 2^{|r|}(3Y+1)X. \end{aligned}$$

Hence, the γ-approximation algorithm returns a solution of cost at most $\gamma 2^{|r|}(3Y+1)X$, and we declare ϕ satisfiable.

If ϕ is unsatisfiable then by Lemma 7.4.6 and our choice of h, with probability $2/3 - o(1)$, any solution have cost at least the minimum of $\Omega(\sigma 2^{|r|}XY)$ and $\Omega(\frac{\ell}{k}X2^{|r|})$. Both these expressions are strictly larger than $\gamma 2^{|r|}(3Y+1)X$:

1. $\Omega(\sigma 2^{|r|}XY) > \gamma 2^{|r|}(3Y+1)X$. Follows immediately for large n from $\sigma > \gamma$.

2. $\Omega(\frac{\ell}{k}X2^{|r|}) > \gamma 2^{|r|}(3Y+1)X$. Follows from

$$\frac{\ell}{k\cdot\gamma} = \frac{\log^{\alpha} n}{\log^{(\alpha/4)+4} n\cdot \log^{(\alpha/4)-5} n} = \frac{\log^{\alpha} n}{\log^{(\alpha/2)-1} n} = \log^{\frac{\alpha}{2}+1} n,$$

   and $\Omega(\log^{\frac{\alpha}{2}+1} n) > 3\log^{\frac{\alpha}{2}} n + 1 = 3Y+1$ for large n.

The construction of the network takes time $O(n^{\text{polylog } n})$ since the network $\mathcal{N}_2$ has size $O(n^{\text{polylog } n})$. Hence we have described a randomized $O(n^{\text{polylog } n})$ time algorithm for 3SAT that has one-sided error probability at most $1/3 + o(1)$. It is possible to convert this into a randomized algorithm that never makes an error and has expected running time $O(n^{\text{polylog } n})$.

$\square$

The size of the Buy-at-Bulk instance is $N = O(2^{2+|r|+|a_0|+|a_1|}XYZ) = 2^{O(\log^{\alpha+2} n)}$. For any constant $\varepsilon > 0$, if we set $\alpha = \frac{11}{2\varepsilon} - 2$ then $\gamma = \Omega(\log^{\frac{1}{4}-\varepsilon} N)$:

$$N = 2^{O(\log^{\frac{11}{2\varepsilon}} n)} \Leftrightarrow \log N = O(\log^{\frac{11}{2\varepsilon}} n),$$

and

$$\gamma = \log^{\frac{\alpha}{4}-5} n = \log^{\frac{11}{2\varepsilon}(\frac{1}{4}-\varepsilon)} n = \Omega(\log^{\frac{1}{4}-\varepsilon} N).$$

In the above construction we had $k = \log^{\frac{\alpha}{4}+4} n = \Omega(\log^{\frac{1}{4}+\frac{7\varepsilon}{11}} N)$. A closer look at the proofs in the paper by Andrews and Zhang [11] reveals that the value of the hidden constant in $\Omega(\log^{\frac{1}{4}+\frac{7\varepsilon}{11}} N)$ does not matter. To summarize, we have shown

Theorem 7.4.8. *For any $\varepsilon > 0$ there is no $O(\log^{\frac{1}{4}-\varepsilon} N)$-approximation algorithm for the Buy-at-Bulk problem with cost function $f(x) = \lceil \frac{x}{k} \rceil$, where $k = \Omega(\log^{\frac{1}{4}+\frac{7\varepsilon}{11}} N)$, unless all problems in* NP *can be solved by a randomized algorithm with expected running time $O(n^{\text{polylog } n})$.*

7.4.2 Hardness with Dependence on k

In this section we show that if $k < \log^{\frac{1}{4}} N$ then Buy-at-Bulk with cost function $f(x) = \lceil \frac{x}{k} \rceil$ cannot be approximated within a factor of $k^{1-\varepsilon}$ for any $\varepsilon > 0$. We use the same network as before and only change a few of the variables.

In the previous section we had $k = \log^{\frac{\alpha}{4}+4} n$. If we allow k to be more than a constant factor smaller than that we must change other variables too, to make the proofs correct.

Change of Variables If k gets smaller then Z increases. This results in a problem in the proof of Lemma 7.4.2, where we want to ensure

$$\log(2e(2^{|a_0|} + 2^{|a_1|})) = o\Big(\frac{10^4\sigma^2\eta 2^{|r|}X - 3\log Y}{3\sigma(m(Q_0) + m(Q_1))XZ}\Big). \tag{7.1}$$

We will therefore change Z, such that the value of Z no longer depends on k, in order to keep Equation 7.1 correct. Let $c > 1$ be a constant such that $k = \log^{\frac{\alpha}{4}+4} n/c$ and set

$$Z = \frac{2^{|r|}}{c \cdot k \cdot \min\{m(Q_0), m(Q_1)\}} = \frac{2^{|r|}}{\log^{\frac{\alpha}{4}+4} n \cdot \min\{m(Q_0), m(Q_1)\}} .$$

Clearly, the value of the right hand side of Equation 7.1 stays the same, as the value of Z is the same as before.

The new definition of Z will change the number of times an answer edge is used in the satisfiable case to $\frac{2^{|r|}X}{m(Q_0)} \cdot \frac{1}{XZ} \le ck$. Thus the cost for each answer edge in this case is $c \cdot h$. To ensure that the cost of the solution when ϕ is satisfiable stays the same we set

$$h = \frac{2^{|r|}}{c \cdot (m(Q_0) + m(Q_1))Z} .$$

The total cost of the answer edges in the satisfiable case is then

$$\begin{aligned} c \cdot h(m(Q_0) + m(Q_1))XYZ &= c(m(Q_0) + m(Q_1))XYZ \cdot \frac{2^{|r|}}{c \cdot (m(Q_0) + m(Q_1))Z} \\ &= 2^{|r|}XY . \end{aligned}$$

No other of the proofs by Andrews and Zhang [11], except the one for the Lemma 7, are affected by the change in Z.

Let Φ be the value of the Buy-at-Bulk solution when ϕ is satisfiable, and let V_1 and V_2 be as in Lemma 7.4.6. Setting h as above we ensure that the values of Φ and V_1 are the same as before. V_2 is not changed by the changes in h and Z, and it increases when k decreases. We can therefore show,

Theorem 7.4.9. *If there exists a $k^{1-\varepsilon}$-approximation algorithm for Buy-at-Bulk with cost function $f(x) = \lceil \frac{x}{k} \rceil$, for any $\varepsilon > 0$ and $k < \log^{\frac{1}{4}} n$, then there exists a randomized $O(n^{\text{polylog } n})$ time algorithm for 3SAT.*

Proof. Let $\gamma = k^{1-\varepsilon}$. For any 3SAT instance ϕ we construct the network $\mathcal{N}_2$ from the two-prover system and then apply a γ-approximation algorithm A for Buy-at-Bulk. We declare ϕ to be satisfiable if and only if A returns a solution of cost at most $\gamma 2^{|r|}(3Y+1)X$.

If the 3SAT instance ϕ is satisfiable then there is a solution to our instance of Buy-at-Bulk of cost at most $\Phi = 2^{|r|}(3Y+1)X$. Hence, the γ-approximation algorithm returns a solution of cost at most $\gamma 2^{|r|}(3Y+1)X$, and we declare ϕ satisfiable.

If ϕ is unsatisfiable then by Lemma 7.4.6 and our choice of h, with probability $2/3 - o(1)$, any solution have cost at least the minimum of $\Omega(\sigma 2^{|r|}XY)$ and $\Omega(\frac{\ell}{k}X2^{|r|})$. We want to show $V_i > k^{1-\varepsilon} \cdot \Phi$ for any $\varepsilon > 0$. To do this we need to express k in terms of n instead of N.

Since $k < \log^{\frac{1}{4}} N$ we have $k \leq \log^{\frac{1}{4}-\epsilon} N$ for some $\epsilon > 0$. Recall, that $\log N = O(\log^{\alpha+2} n)$. By setting $\alpha \geq \frac{5}{2\epsilon} - 2$ we ensure $k < \log^{\frac{\alpha}{4}-1} n$ for large n. Setting $\alpha = \frac{5}{2\epsilon} - 2$ we get

$$k \leq \log^{\frac{1}{4}-\epsilon} N = O(\log^{(\frac{1}{4}-\epsilon)(\alpha+2)} n) = O(\log^{\frac{5}{8\epsilon}-\frac{5}{2}} n) ,$$

and

$$\log^{\frac{\alpha}{4}-2} n = \log^{\frac{5}{8\epsilon}-\frac{5}{2}} n .$$

We are now ready to show $V_i > k^{1-\varepsilon} \cdot \Phi$. There are two cases.

1. $V_1 > k^{1-\varepsilon} \cdot \Phi$: Since $k \leq \log^{\frac{\alpha}{4}-1} n$ we have $\sigma = \log^{\frac{\alpha}{4}} n > k$ and thus $\Omega(\sigma 2^{|r|}XY) > k^{1-\varepsilon} \cdot 2^{|r|}(3Y+1)X$, since $\varepsilon > 0$.

2. $V_2 > \gamma \cdot \Phi$: Note that

$$\ell = \log^{\alpha} n > 3\log^{\alpha-2} n + \log^{\alpha/2-2} n \geq k^2(3Y+1) > k^{2-\varepsilon}(3Y+1),$$

   for $\log n \geq 2$, when $\alpha > 0$. Since we can set α to anything greater than $\frac{5}{2\epsilon} - 2$ we can ensure $\alpha > 0$. Thus $\Omega(\frac{\ell}{k}X2^{|r|}) > k^{1-\varepsilon} \cdot 2^{|r|}(3Y+1)X$. □

From Theorem 7.4.8 and Theorem 7.4.9 we get

Corollary 7.4.10. *For any $\varepsilon > 0$, there is no $\min\{O(\log^{\frac{1}{4}-\varepsilon} N), k^{1-\varepsilon}\}$-approximation algorithm for Buy-at-Bulk with cost function $f(x) = \lceil \frac{x}{k} \rceil$ unless all problems in* NP *can be solved by a randomized algorithm with expected running time $O(n^{\text{polylog } n})$.*

7.5 Routing in the Network

Let B be the instance of Buy-at-Bulk constructed in Section 7.4, and let D be an instance of the preemptive Dial-a-Ride with the same source-destination pairs S also in $\mathcal{N}_2$.

Let SOL_B denote the solution used to give the bound of the cost of the optimal solution in Lemma 7.4.1, and let OPT_D be the optimal solution to the Dial-a-Ride instance. In this section we show how construct a tour for the vehicle of cost at most $7 \cdot \mathsf{SOL}_B$, in the case when ϕ is satisfiable.

Let $\mathcal{N}_2^f$ be the graph consisting of the edges bought in Buy-at-Bulk solution SOL_B. Recall that in SOL_B all demands are routed on canonical paths. For each demand d, let p_d be the canonical path that d is routed on in the Buy-at-Bulk solution SOL_B. We will say that an edge $e \in \mathcal{N}_2^f$ is *used* by a item d if e is on the path p_d.

7.5.1 The Tour when $\mathcal{N}_2^f$ is Connected

We will first explain how to construct the tour when $\mathcal{N}_2^f$ is connected. We will say that the tour is using an edge in the forward direction if it goes from left to right when the graph is drawn as in Figure 7.3, and backwards otherwise.

Assume without that any edge in $\mathcal{N}_2^f$ is used by at most k items (we will show in the end of the section how to get rid of this assumption). We will ensure that the tour has the following properties:

(i) The tour only uses edges from $\mathcal{N}_2^f$.

(ii) An item d will only be in the vehicle when the vehicle is on an edge $e \in p_d$.

(iii) When the vehicle goes forward on an edge it is either empty or carries all items using that edge.

Algorithm Start at vertex $s = s_{r,x}$ for some r, x and pick up $d = d_{r,x}$. Item d is now the *active item*. We will deliver d ensuring the above properties.

Follow path p_d. Whenever there is an edge e on p_d used by items other than the active item d do:

1. If all such items are present at the left endpoint of e, pick up all these items, and traverse e. At the right endpoint of e drop off all items not going in the same direction as d and proceed along p_d as before.

2. If some items using the edge are not at the left endpoint, the vehicle drops of d and goes to pick up these items as follows. Let d' be such an item. The vehicle follows $p_{d'}$ backwards from e until it encounters d'. It then picks up d' and goes forward on path $p_{d'}$. On the way to e the same problem may happen again—we need to go forward on an edge that is used by other flows, and the corresponding items are not there. This is taken care of the same way as before.

When d reaches its destination t_d it is dropped off and d is no longer the active item. The vehicle then traverses the route followed until now backwards until it gets to an undelivered item d' left at some node on the route. It picks up d' (d' is now the active item) and deliver it in the same way as d. When d' and all items that were left "hanging " in the graph due to the delivery of d' are delivered, the vehicle goes back to the point where d' was picked up when it became the active item. It then keeps following the route constructed when delivering the previous active item backwards to find other hanging items.

When all items are delivered the vehicle takes the shortest route back to the starting point.

Analysis It is easy to verify that the tour made by the algorithm satisfies property (i), (ii), and (iii). We will say that the part of the tour followed by the vehicle while an item d was active *belongs to* d, and we denote this route by r_d.

Lemma 7.5.1. *For any item d, the route r_d, has the following properties:*

(iv) *The route r_d only goes backwards on an edge e to fetch "missing" items. If d' is such an item then $e \in p_{d'}$.*

(v) *If r_d goes backwards on edge e it returns to the right endpoint of e through e.*

(vi) *When route r_d traverses an edge e in the forward direction the vehicle contains all items using e.*

Proof. It follows immediately from the description of the algorithm that the route only goes backwards to fetch missing items, and that if d' is such an item then

$e \in p_{d'}$ (property (iv)). Property (vi) also follows directly from the description. We need to show property (v) and thereby also that it is possible to fetch all missing items before traversing an edge.

All canonical paths go through all levels of the graph in increasing order. Therefore an item missing at the left endpoint of some edge at level i can be fetched at a level smaller than i or at i if the edge is not the first edge on level i. It is therefore possible to fetch all items missing at a certain node, since there are no cyclic dependencies.

When the vehicle follows the canonical paths of the missing items using edge e at level i, it only traverses the edges of these canonical paths on levels smaller than or equal to i. Therefore the only way the vehicle can return to the right endpoint of e using canonical paths for items using e is through e. ☐

The properties of the lemma gives us the following two corollaries.

Corollary 7.5.2. *For any item d, the route r_d traverses each edge in $\mathcal{N}_2^f$ at most once in each direction.*

Proof. Follows from the properties in Lemma 7.5.1. While d is active the vehicle is—due to property (vi)—only going forward on an edge when it is carrying all items going on that edge. This can clearly happen only once per edge. When the vehicle is going backward on e it returns to the right endpoint of e using e due to property (v). Due to property (v) it carries all items using e when returning to the right endpoint of e, and due to property (iv) it can thus only go backwards on e once, since no more items using edge e are missing. ☐

Corollary 7.5.3. *For any two items d_1 and d_2 the routes r_{d_1} and r_{d_2} are disjoint.*

Proof. Follows from the properties in Lemma 7.5.1. Consider an edge e. By property (vi) e is only traversed in the forward direction in r_{d_i} when the vehicle carries all the items using that edge. Thus, only one of r_{d_1} and r_{d_2} can traverse e in the forward direction. If r_{d_i} goes backwards on edge e it also goes forward due to property (v).By the previous argument only one of the routes can go forward on e, and thus also only one of them can go backwards. ☐

Lemma 7.5.4. *All items are delivered to their destination.*

Proof. By contradiction. Recall, we assume that $\mathcal{N}_2^f$ is connected. Assume some subset of items S are not delivered. Consider an item $d \in S$. Item d cannot have been left hanging at some node in network, since then it would have been picked up and delivered when the vehicle goes back on the tour $r_{d'}$, where d' is

the element that was active when d was left at the node. Thus d must still be at its source s_d.

Since d is still at s_d the path p_d does not share any edges with any path $p_{d'}$ where d' is a delivered item. Assume d shared an edge e with a delivered item d'. Due to property (ii) the vehicle crossed e containing d', since d' is delivered. Due to property (vi) of Lemma 7.5.1 d must have been in the vehicle when it crossed e, and thus d would no longer be at its source node s_d.

Since SOL_B are using canonical paths for each item, the graph $\mathcal{N}_2^f$ has the property that if two canonical paths p_d and $p_{d''}$ meet at some vertex then they must share an edge adjacent to that vertex. Therefore p_d cannot share any vertices with any path $p_{d'}$ where d' is a delivered item. This is true for all items $d \in S$, contradicting that $\mathcal{N}_2^f$ is connected. □

Lemma 7.5.5. *When $\mathcal{N}_2^f$ is connected the tour has length at most $4 \cdot \mathsf{SOL}_B$.*

Proof. Let $l(r_d)$ denote the length of the route r_d. The sum of the parts of the tour where there is an active item is of length $\sum_{d\in D, d \text{ active}} l(r_d)$.

Now consider the parts of the tour when there is no active item. This happens when we are going backwards on the tour belonging to some item d to find the next item to be the active item. The total length of these parts of the tour is at most $\sum_{d\in D, d \text{ active}} l(r_d)$, since each route r_d is traversed at most once backwards to find non-delivered items. When we traverse a route r_d backwards to find hanging items, we stop each time we meet such an item d' and deliver it. That item is then the active item, and by Corollary 7.5.3 $r_{d'}$ and r_d are disjoint. When this item is delivered, we go backwards on $r_{d'}$—which was disjoint from r_d—and then return to r_d where d' was picked up. The route r_d is thus traversed at most once to find non-delivered items.

Adding together the length of the tour when there is an active item and the length of tour when there are no active items, we get $2 \cdot \sum_{d\in D,\, d \text{ active}} l(r_d)$.

Using Corollary 7.5.2 and Corollary 7.5.3 we get that the tour uses each edge in $\mathcal{N}_2^f$ at most 4 times, and thus the cost of the tour is at most $4 \cdot \mathsf{SOL}_B$. □

Edges used by than k items We assumed that any edge in $\mathcal{N}_2^f$ is used by at most k items. We can get rid of this assumption by a minor modification of the algorithm. Let S_e be the set of items using edge e. Then the solution SOL_B paid $\lceil \frac{S_e}{k} \rceil \cdot l_e$ for this edge. As before, when we want to traverse e we go backwards and pick up all items in S_e. We then go forward and back on e carrying as many items from S_e as possible each time until all items from S_e are on the right endpoint of e. The number of times we traverse e is $\lceil \frac{S_e}{k} \rceil$, and thus Lemma 7.5.5 still holds.

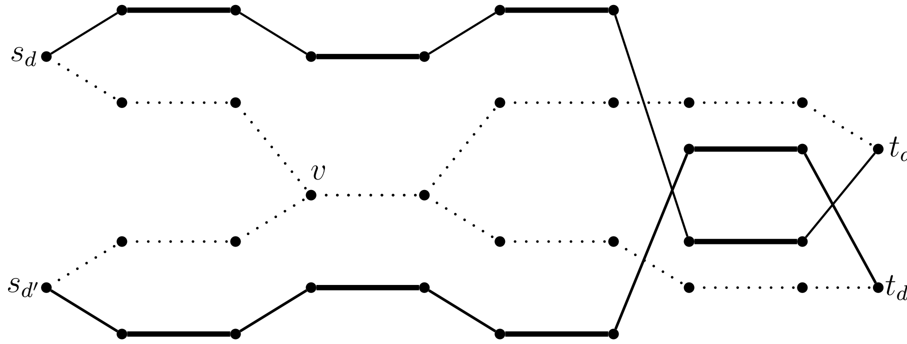


Figure 7.4: An example network with two demands, that each has two canonical paths. The network $\mathcal{N}_2^f$ (solid edges) is disconnected, but $\mathcal{N}_2^c$ (all edges) is connected. We can connect $\mathcal{N}_2^f$ by adding the edges on the path from s_d to v and from v to $t_{d'}$.

7.5.2 $\mathcal{N}_2^c$ Connected and $\mathcal{N}_2^f$ Disconnected

Let $\mathcal{N}_2^c$ be the graph induced by the canonical paths. If $\mathcal{N}_2^c$ is connected but $\mathcal{N}_2^f$ is disconnected we can add edges from $\mathcal{N}_2^c$ to $\mathcal{N}_2^f$ to connect it. We can do this by adding edges of total length equal to the number of connected components minus one times the length of a canonical path in $\mathcal{N}_2^c$.

First we note that since $\mathcal{N}_2^c$ consists of the union of canonical paths, then for any component C in $\mathcal{N}_2^f$ there must be another component C' in $\mathcal{N}_2^f$ such that some item d routed in C has a canonical path p that intersect with a canonical path p' for an item d' routed in C'. We connect C and C' by adding the following edges: All edges on p from s_d to the intersecting edge e (including e), and all edges on p' from e to $t_{d'}$ (see Figure 7.4). We call these added edges a *connecting path from C' to C*. Since $\mathcal{N}_2^c$ is connected we can make $\mathcal{N}_2^f$ connected by adding $c-1$ connecting paths, where c is the number of connected components in $\mathcal{N}_2^f$. We add these connecting paths in such a way that all components can be reached from one component—called the *start component*—using a path that when going from component C to a component C' uses a connecting path from C to C' (not from C' to C). Since the length of a connecting path is the same as the length of a canonical path the total length is $c-1$ times the length of a canonical path. Since each connected component consists of at least one canonical path the total length of the connecting paths is at most the same as the sum of all edges in $\mathcal{N}_2^f$, i.e., SOL_B.

We now describe the tour. Start in the start component C_s in $\mathcal{N}_2^f$ and deliver the items in this component as described in the previous section. Every time that the vehicle has delivered an item d that has an adjacent connecting path from C_s to

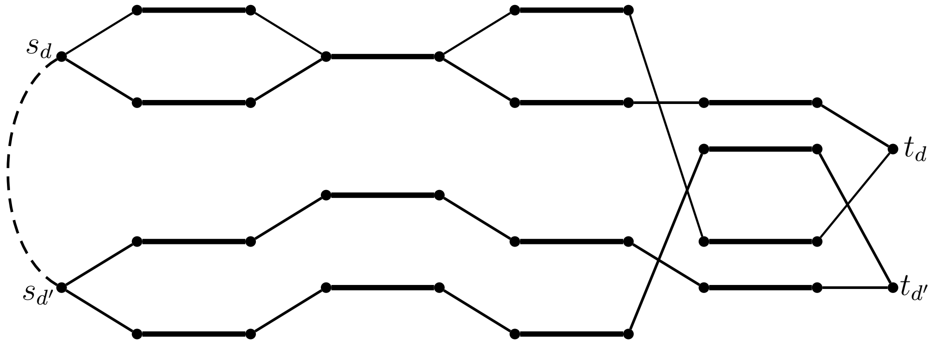


Figure 7.5: An example network with two demands, that each has two canonical paths. The the two components are connected with a component edge (dashed).

another component C, it follows this connecting path to C and delivers the items in C the same way. When all items in a component C are delivered the vehicle returns to the starting point in C and from there to the previous component C' if such a component exists. It then carries on delivering the items in C'.

Lemma 7.5.6. *When $\mathcal{N}_2^c$ is connected the optimal solution to the Dial-a-Ride instance has cost at most $6 \cdot \mathsf{SOL}_B$.*

Proof. If $\mathcal{N}_2^f$ is connected it follows from Lemma 7.5.5. If $\mathcal{N}_2^f$ is not connected we use the approach described above. To deliver the items in a single component we use no more time than in the previous section. By Lemma 7.5.5 the contribution from these parts of the tour is at most $4 \cdot \mathsf{SOL}_B$ in total. To get to the next component and back again we use a connecting path and the sum of the edges used to get to and from connected components is thus at most $2 \cdot \mathsf{SOL}_B$. □

7.5.3 $\mathcal{N}_2^c$ Disconnected

If $\mathcal{N}_2^c$ is disconnected we connect it by adding edges of length one between a source node in one component and a source node in another component (see Figure 7.5). We call these edges *component* edges. We add the minimum number of component edges, i.e., $l - 1$ where l is the number of connected components. This can be seen as constructing a tree on the components.

Since we add the component edges between disjoint components in $\mathcal{N}_2^c$, which are also disjoint components in $\mathcal{N}_2$, we do not introduce any new cycles in $\mathcal{N}_2$. Therefore the component edges cannot decrease the cost of the optimal solution to the Buy-at-Bulk instance or to the Dial-a-Ride instance: Let C_1 and C_2 be two components connected by a component edge e. If some item d with source s_d in

C_1 is using e, then it has to use it again to get back to C_1, since $s_d \in C_1$ and the only connection between C_1 and C_2 goes through e.

We also need to add edges in the incidence graph $\mathcal{G}_2$ corresponding to the component edges in $\mathcal{N}_2$. For each component edge e in $\mathcal{N}_2$, between s_d and s'_d, we will add an edge of length one between d and d' in $\mathcal{G}_2$. Since $\mathcal{G}_2$ has the property that disjoint components in $\mathcal{N}_2$ map to disjoint components in $\mathcal{G}_2$, we do not introduce any new cycles in $\mathcal{G}_2$. Therefore all the lemmas and theorems in Section 7.4 still hold.

Constructing the Tour The vehicle first delivers the items in a component C in $\mathcal{N}_2^c$ as described in the previous section. When it gets to the source node in the component that has a component edge to a source node in another component C', it goes to C' and delivers the items in C' the same way. When all items in a component are delivered it returns to the starting point of this component and follows the component edge back to the previous component C if such a component exists. It then carries on delivering the items in component C.

Lemma 7.5.7. *The optimal solution to the Dial-a-Ride instance has cost at most* $7 \cdot \mathsf{OPT}_B$.

Proof. The cost of delivering the items in the original components of $\mathcal{N}_2$ is at most $6 \cdot \mathsf{SOL}_B$ due to Lemma 7.5.6. The total length of the new edges is $l-1$ which is less than $1/2 \cdot \mathsf{SOL}_B$, since each connected component has a canonical path of length greater than two. The new edges are used twice: once in each direction. ☐

7.6 Hardness of Preemptive Dial-a-Ride

In the previous section we showed that the value optimal solution to the Dial-a-Ride instance D, OPT_D, is at most 7 times value of the optimal solution to the corresponding Buy-at-Bulk instance B, OPT_B (Lemma 7.5.7). In Section 7.4 we gave an upper bound on OPT_B when ϕ is satisfiable. Putting together the results of Lemma 7.5.7 and Lemma 7.4.1, we get

Lemma 7.6.1. *If ϕ is satisfiable, then the Dial-a-Ride instance has a solution of total cost* $7 \cdot 2^{|r|}(2Y+1)X + (m(Q_0) + m(Q_1))hXYZ$.

We have showed that $\mathsf{OPT}_B \leq \mathsf{OPT}_D$ (Lemma 7.2.1) and given a lower bound on OPT_B when ϕ is unsatisfiable (Lemma 7.4.6). Using these two lemmas together with Lemma 7.6.1, we get

Lemma 7.6.2. *Let $\gamma = \log^{\frac{\alpha}{4}-5} n$. If there exists a γ-approximation algorithm for the Finite Capacity Dial-a-Ride problem, then there exists a randomized $O(n^{\text{polylog } n})$ time algorithm for 3SAT.*

The proof is the same as the proof of Lemma 7.4.7. In the Dial-a-Ride instance N is the number of sources and destinations, which is at most twice the number of demands. Recall, that we have $2^{|r|}X$ demands, and thus $N = 2\cdot 2^{|r|}X = 2^{O(\log^{\alpha+2} n)}$. By the same calculations as in the Buy-at-Bulk case, we get

Corollary 7.6.3. *Let $k = \Omega(\log^{\frac{1}{4}+\frac{7\varepsilon}{11}} N)$. Then there is no $O(\log^{\frac{1}{4}-\varepsilon} N)$-approximation algorithm to the preemptive Finite Capacity Dial-a-Ride problem on general graphs for any constant $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\text{polylog} n})$.*

By changing the variables as in Section 7.4.2 and using Lemma 7.2.1 and Lemma 7.4.6, we get

Corollary 7.6.4. *Let $k < \log^{\frac{1}{4}} N$. Then there is no $k^{1-\varepsilon}$-approximation algorithm to the preemptive Finite Capacity Dial-a-Ride problem on general graphs for any constant $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\text{polylog} n})$.*

The proof is the same as the proof of Theorem 7.4.9. To summerize we have shown,

Theorem 7.6.5. *There is no $\min\{O(\log^{\frac{1}{4}-\varepsilon} N), k^{1-\varepsilon}\}$-approximation algorithm to the preemptive Finite Capacity Dial-a-Ride problem on general graphs for any constant $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(n^{\text{polylog} n})$.*

Chapter 8

Future Work

In this chapter we will discuss possible directions for future work.

8.1 Multiple Dispatching

Our algorithm uses optimal space, so the first obvious thing to do would be to try to improve the running time. Another question is how to make our data structure dynamic. One way would be to use *retroactive* data structures, introduced by Demaine *et al.* [40]. In a retroactive data structure it is possible to insert, delete, or change an update operation performed at a given time in the past. All such retroactive changes potentially affect all existing versions between the time of modification and the present time. A data structure is *partially retroactive* if, in addition to supporting updates and queries on the current version of the data structure, it supports insertion and deletion of updates at past times. A data structure is *fully retroactive* if, in addition to allowing updates in the past, it can answer queries about the past. A fully retroactive data structure can be seen as a partially retroactive version of a partially persistent data structure. One way to construct a dynamic bridge color data structure would be to make a partially retroactive fully persistent data structure for the tree color problem. That seems not to be the easiest way to get around the problem. Instead it is possible to solve the bridge color problem within the same time and space bound as our solution using partial persistence (see [99]). A fully retroactive predecessor data structure and a fully retroactive tree color data structure would thus give a dynamic data structure for our problem. Demaine *et al.* [40] gave a number of fully retroactive data structures, but provided no general way to obtain a fully retroactive version of a data structure. They gave a fully retroactive predecessor data structure using $O(\log^2 n)$ time per operation and $O(n \log n)$ space.

It would also be interesting to see if similar ideas could be used to solve the bridge color problem for $d > 2$. In that case, the bridges would be hyperedges between d trees.

In some object oriented languages classes can inherit from more than one class, and the class hierarchy is thus a DAG. Constructing a data structure for the dispatching problem in that case requires other ideas than those used to solve the bridge color problem.

8.2 Tree Inclusion

As the space is optimal, the obvious direction would be to get the worst-case running time down, i.e., below $O(n_P n_T / \log n_T)$.

We plan to try to adapt our tree inclusion algorithm to the queries made in *core XPATH* (see e.g.[61]), and implement it to see if we can improve the running times.

For some applications considering *unordered* trees is more natural. However, in [91, 80] this problem was proved to be NP-complete. Constructing approximation algorithms for optimization variants of the tree inclusion problem would be interesting, and would make a connection between the two different research areas presented in the dissertation.

8.3 Union-find with Deletions

As the time and space complexities of our algorithm are asymptotically optimal, there is not much to do here. One possibility would be to look into whether our algorithm will work for the entire optimal worst case range, i.e., *union* in $O(k)$ and *find* in $O(\log_k n)$ time.

Another possibility would be to see if our potential function can be used to give local amortized bounds for the two one-pass variants of path compression path halving and path splitting.

8.4 Asymmetric k-Center

The approximation factor of our algorithms for the weighted k-center and k-center with minimum coverage problems are essentially the best possible, and the algorithms run in $\tilde{O}(n^2)$ time. Further effort into these problems does not

seem warranted. Getting down to k centers in the p-neighbor k-center problem is an open problem.

8.5 Dial-a-Ride

Closing the gap between the $O(\log N)$ upper bound and our $\Omega(\log^{\frac{1}{4}-\varepsilon} N)$ lower bound in the preemptive case would be interesting. The $O(\log N)$ algorithms relies on the a results on approximating arbitrary metrics by tree metrics [48]. The $O(\log n)$ is tight for approximating metrics by tree metrics, and thus another approach is needed to get an approximation guarantee below $O(\log N)$.

We are currently working on showing similar hardness results for the non-preemptive Dial-a-Ride problem. The best known upper bound for this problem is $O(\sqrt{k}\log N)$ and it seems to get its hardness for other reasons than the preemptive version. It is not possible to directly adapt the techniques used for the preemptive version to the non-preemptive version. The result showing the relation between the preemptive Dial-a-Ride and the buy-at-bulk problem in the network $\mathcal{N}_2$ heavily relies on the preemptiveness of the problem. A first step in narrowing down the gap would be to get an algorithm with better approximation guarantee than $O(\sqrt{k})$ on trees or to show that this is the best possible. The $O(\sqrt{k})$-approximation algorithm always picks up k objects and deliver all these before picking up any new objects. We will call an algorithm that always pick up a group of items and deliver all of them without any intermixed pickups and deliveries for a fetch-deliver algorithm. It is possible to give examples where a fetch-deliver algorithm will give a tour of length $k/\log k$ more than the optimal tour (this is on a line and not on the special instances of height-balanced trees that the $O(\sqrt{k})$-approximation algorithm works on). This indicates that we should try to find an approximation algorithm that is not a fetch-deliver algorithm. In trying to show hardness for the problem, it will possibly be a good idea to exploit this property too.

We would also like to investigate other similar routing problems like Dial-a-Ride with time-windows, Dial-a-Ride where all objects are initially located at one depot (symmetric and asymmetric), k-delivery TSP, and the famous asymmetric TSP.

Bibliography

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

[2] L. Alonso and R. Schott. On the tree inclusion problem. In *Proceedings of Mathematical Foundations of Computer Science*, pages 211–221, 1993.

[3] S. Alstrup, A. M. Ben-Amram, and T. Rauhe. Worst-case and amortised optimality in union-find. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 499–506, May 1999.

[4] S. Alstrup, I. L. Gørtz, T. Rauhe, and M. Thorup. Worst-case union-find with fast deletions. Technical Report TR-2003-25, IT University Technical Report Series, 2003.

[5] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming*, pages 270–280, 1997.

[6] S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *Scandinavian Workshop on Algorithm Theory*, pages 46–56, 2000.

[7] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 534–543, 1998.

[8] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *SODA '02: Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms*, pages 947–953, 2002.

[9] M. Andrews. Hardness of buy-at-bulk network design. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 115–124, October 2004.

[10] M. Andrews and L. Zhang. Bounds on fiber minimization in optical networks with fixed fiber capacity. In *IEEE INFOCOM*, 2005.

[11] M. Andrews and L. Zhang. Bounds on fiber minimization in optical networks with fixed fiber capacity (full version). Unpublished manuscript, 2005.

[12] A. Archer. Inapproximability of the asymmetric facility location and k-median problems. Unpublished manuscript available at `www.orie.cornell.edu/~aarcher/Research`, 2000.

[13] A. Archer. Two $O(\log^* k)$-approximation algorithms for the asymmetric k-center problem. In K. Aardal and B. Gerads, editors, *Integer Programming and Combinatorial Opimization (IPCO)*, volume 2081 of *Lecture Notes in Computer Science*, pages 1–14, Berlin Heidelberg, 2001. Springer-Verlag.

[14] A. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 14–23, 1992.

[15] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[16] V. Arya, N. Garg, R. Khandekar, K. Munagala, A. Meyerson, and V. Pandit. Local search heuristic for k-median and facility location problems. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 21–29, 2001.

[17] M. J. Atallah and S. R. Kosaraju. Efficient solutions to some transportation problems with applications to minimizing robot arm travel. *SIAM Journal on Computing*, 17(5):849–869, 1988.

[18] B. Awerbuch and Y. Azar. Buy-at-bulk network design. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, pages 542–547, 1997.

[19] A. M. Ben-Amram and Z. Galil. A generalization of a lower bound technique due to fredman and saks. *Algorithmica*, 30, 2001.

[20] R. Bhatia, S. Guha, S. Khuller, and Y. J. Sussmann. Facility location with dynamic distance function. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 23–34, 1998.

[21] P. Bille. A survey on tree edit distance and related problems. *To appear in Theorectical Computer Science (TCS)*, 2005.

[22] P. Bille and I. L. Gørtz. The tree inclusion problem: In optimal space and faster. Technical Report TR-2005-54, IT University of Copenhagen, January 2005.

[23] N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4):1021–1024, 1986.

[24] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common LISP object system specification X3J13 document 88-002R. *ACM SIGPLAN Notices*, 23, 1988. Special Issue, September 1988.

[25] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.

[26] M. Charikar, S. Khuller, D. M. Mount, and G.Narasimhan. Algorithms for facility location problems with outliers. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 642–651, Jan. 2001.

[27] M. Charikar, S. Khuller, and B. Raghavachari. Algorithms for capacitated vehicle routing. *SICOMP: SIAM Journal on Computing*, 31(3):665–682, 2002.

[28] M. Charikar and B. Raghavachari. The finite capacity dial-a-ride problem. In *39th Annual IEEE Symposium on Foundations of Computer Science*, pages 458–467, November 1998.

[29] S. Chaudhuri, N. Garg, and R. Ravi. The p-neighbor k-center problem. *Information Processing Letters*, 65(3):131–134, 13 Feb. 1998.

[30] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[31] W. Chen. More efficient algorithm for ordered tree inclusion. *Journal of Algorithms*, 26:370–385, 1998.

[32] N. Christofedes. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburg, PA, 1976.

[33] M. J. Chung. $O(n^{2.5})$ algorithm for the subgraph homeomorphism problem on trees. *Journal of Algorithms*, 8(1):106–112, 1987.

[34] J. Chuzhoy, S. Guha, E. Halperin, G. Kortsarz, S. Khanna, and S. Naor. Asymmetric k-center is $\log^* n$-hard to approximate. In *Proceedings of the 36th annual ACM symposium on Theory of computing (STOC)*, pages 21–7, 2004.

[35] J. Chuzhoy, S. Guha, S. Khanna, and S. Naor. Asymmetric k-center is $\log^* n$-hard to approximate. Technical Report 03-038, Elec. Coll. Comp. Complexity, 2003.

[36] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000.

[37] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic o(n log3 n)-time. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 245–254. Society for Industrial and Applied Mathematics, 1999.

[38] I. A. Computer. Dylan interim reference manual, 1994.

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

[40] E. D. Demaine, J. Iacono, and S. Langerman. Retroactive data structures. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 281–290, 2004.

[41] P. F. Dietz. Fully persistent arrays. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74, Berlin, Aug. 1989. Springer-Verlag.

[42] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.

[43] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 524–531. IEEE Computer Society Press, 1988.

[44] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and Systems Sciences*, 38(1):86–124, 1989.

[45] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. In *Proceedings of the 31st IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 145–150, 1990.

[46] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-dispatch in the java virtual machine: Design and implementation. In *6th USENIX Conference on Object-Oriented Technologies*, 2001.

[47] D. Eppstein and S. Muthukrishnan. Internet packet filter manegement and rectangle geometry. In *Proceedings of the 12th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

[48] Fakcharoenphol, Rao, and Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *JCSS: Journal of Computer and System Sciences*, 69(3):385–497, 2004.

[49] P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In *European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 107–120, 1996.

[50] P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching problems. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 483–491, May 1999.

[51] R. Fleischer. A simple balanced search tree with O(1) worst-case update time. *International Journal of Foundations of Computer Science*, 7:137–149, 1996.

[52] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *IEEE Symposium on Foundations of Computer Science*, pages 632–641, 1991.

[53] G. N. Frederickson. A note on the complexity of a simple transportation problem. *SIAM Journal on Computing*, 22(1):57–61, 1993.

[54] G. N. Frederickson and D. J. Guan. Preemptive ensemble motion planning on a tree. *SIAM Journal on Computing*, 22(1):1130–1152, 1992.

[55] G. N. Frederickson and D. J. Guan. Non-preemptive ensemble motion planning on a tree. *Journal of Algorithms*, 15(1):29–60, 1993.

[56] G. N. Frederickson, M. S. Hecht, and C. E. Kim. Approximation algorithms for some routing problems. *SIAM Journal on Computing*, 7(2):178–193, 1978, May.

[57] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89)*, pages 345–354, May 1989.

[58] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319, Sept. 1991.

[59] M. R. Garey and D. S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[60] I. L. Gørtz and A. I. Wirth. Asymmetry in k-center variants. In *Proceedings of the 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, volume 2764 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 2003.

[61] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases (VLDP)*, pages 95–106, 2002.

[62] D. J. Guan. Routing a vehicle of capacity greater than one. *Discrete Applied Mathematics*, 81(1-3), 1998.

[63] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69–85, 2001.

[64] M. Haimovich and A. H. G. Rinnooy Kan. Bounds and heuristics for capacitated routing problems. *Mathematics of Operations Research*, 10(4):527–542, 1985.

[65] E. Halperin, G. Kortsarz, and R. Krauthgamer. Tight lower bounds for the asymmetric k-center problem. Technical Report 03-035, Elec. Coll. Comp. Complexity, 2003.

[66] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[67] D. S. Hochbaum and D. B. Shmoys. A best possible approximation algorithm for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.

[68] D. S. Hochbaum and D. B. Shmoys. A unified approach to approximate algorithms for bottleneck problems. *Journal of the ACM*, 33(3):533–550, July 1986.

[69] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the Association for Computing Machinery (JACM)*, 29(1):68–95, 1982.

[70] W. L. Hsu and G. L. Nemhauser. Easy and hard bottelneck location problems. *Discrete Appl. Math.*, 1:209–216, 1979.

[71] G. F. Italiano and R. Raman. Topics in data structures. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*. CRC Press, 1998.

[72] H. Kaplan. Persistent data structures. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*. CRC Press, 2005.

[73] H. Kaplan, M. Lewenstein, N. Shafrir, and M. Sviridenko. Approximation algorithms for asymmetric tsp by decomposing directed regular multigraphs. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 56–67, 2003.

[74] H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. In *Proceedings of the 34rd annual ACM symposium on Theory of computing*, pages 573–582, New York, NY, USA, 2002. ACM Press.

[75] H. Kaplan, N. Shafrir, and R. E. Tarjan. Union-find with deletions. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 19–28, Jan. 2002.

[76] O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems. I. The p-centers. *SIAM Journal on Applied Mathematics*, 37(3):513–538, Dec. 1979.

[77] Khuller, Pless, and Sussmann. Fault tolerant K-center problems. *Theoretical Computer Science (TCS)*, 242, 2000.

[78] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, Department of Computer Science, November 1992.

[79] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the 16th Ann. Int. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222. ACM Press, 1993.

[80] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal of Computing*, 24:340–356, 1995.

[81] P. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th annual European Symposium on Algorithms (ESA) 1998.*, pages 91–102. Springer-Verlag, 1998.

[82] P. Klein, S. Tirthapura, D. Sharvit, and B. Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 696–704, 2000.

[83] D. E. Knuth. *The Art of Computer Programming, Volume 1*. Addison Wesley, 1969.

[84] S. R. Kosaraju. Efficient tree pattern matching. In *Proceedings of the 30th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 178–183, 1989.

[85] D. L. Kozen. *The Design and Analysis of Algorithms*. Springer, Berlin, 1992.

[86] C. Levcopoulos and M. Overmars. A balanced search tree with O(1) worstcase update time. *Acta Informatica*, 26:269–277, 1988.

[87] A. Lim, B. Rodrigues, F. Wang, and Z. Xu. k-center problems with minimum coverage. In *Proceedings of 10th International Computing and Combinatorics Conference (COCOON)*, pages 349–359. Springer-Verlag, 2004.

[88] O. Madsen, H. Ravn, and J. Rygaard. A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives. *Annals of Operations Research*, 60:193–208, 1995.

[89] M. Mahdian, Y. Ye, and J. Zhang. An 1.52-approximation algorithm for the uncapacitated facility location problem, 2002. Manuscript.

[90] H. Mannila and K. J. Räihä. On query languages for the p-string data model. *Information Modelling and Knowledge Bases*, pages 469–482, 1990.

[91] J. Matoušek and R. Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Mathematics*, 108:343–364, 1992.

[92] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log\log n)$ time and $O(n)$ space. *Information Processing Letters*, 35:183–189, 1990.

[93] S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–51, Jan. 1996.

[94] A. Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents, 2002.

[95] M. H. Overmars. The design of dynamic data structures. volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[96] R. Panigrahy and S. Vishwanathan. An $O(\log^* n)$ approximation algorithm for the asymmetric p-center problem. *Journal of Algorithms*, 27(2):259–268, May 1998.

[97] C. H. Papadimitriou and S. Vempala. On the approximability of the traveling salesman problem (extended abstract). In *Proceedings of the thirty-second annual ACM symposium on Theory of computing (STOC)*, pages 126–133, 2000.

[98] Plesnik. A heuristic for the p-center problem in graphs. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 17:263–268, 1987.

[99] C. K. Poon and A. Kwok. Space optimal packet classification for 2d conflict-free filters. In *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 260–265, 2004.

[100] R. Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time*. PhD thesis, University of Rochester, Computer Science Department, October 1992. Technical Report TR439.

[101] R. Ramesh and I. Ramakrishnan. Nonlinear pattern matching in trees. *Journal of the Association for Computing Machinery (JACM)*, 39(2):295–316, 1992.

[102] R. Raz. A parallel repetition theorem. *SIAM Journal on Computing*, 27(3):763–803, June 1998.

[103] T. Richter. A new algorithm for the ordered tree inclusion problem. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM), in Lecture Notes of Computer Science (LNCS), volume 1264*, pages 150–166. Springer, 1997.

[104] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[105] T. Schlieder and H. Meuss. Querying and ranking XML documents. *J. Am. Soc. Inf. Sci. Technol.*, 53(6):489–503, 2002.

[106] T. Schlieder and F. Naumann. Approximate tree embedding for querying XML data. In *ACM SIGIR Workshop On XML and Information Retrieval*, 2000.

[107] R. Seidel and M. Sharir. Top-down analysis of path compression. *SIAM J. Comput.*, 34(3):515–525, 2005.

[108] R. Shamir and D. Tsur. Faster subtree isomorphism. *Journal of Algorithms*, 33:267–280, 1999.

[109] M. Smid. A data structure for the union-find problem having good single-operation complexity. *ALCOM: Algorithms Review, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM)*, 1, 1990.

[110] L. Suen, A. Ebrahim, and M. Oksenhendler. Computerized dispatching for shared-ride taxi operations. *Transportation and Planning Technology*, 33:33–48, 1981.

[111] K.-C. Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery (JACM)*, 26:422–433, 1979.

[112] R. E. Tarjan. Testing flow graph reproducibility. In *Proceddings of the 5th Annual ACM Symposium on Theory of Computing (STOC)*, pages 96–107, 1973.

[113] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal of Computing*, 3:62–89, 1974.

[114] R. E. Tarjan. Efficiency of a good but not linear disjoint set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[115] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[116] R. E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.

[117] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, Apr. 1984.

[118] A. Termier, M. Rousset, and M. Sebag. Treefinder: a first step towards XML data mining. In *IEEE International Conference on Data Mining (ICDM)*, 2002.

[119] M. Thorup. Space efficient dynamic stabbing with fast queries. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of Computing (STOC)*, pages 649–658, New York, NY, USA, 2003. ACM Press.

[120] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1978.

[121] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[122] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin Heidelberg, 2001.

[123] S. Vishwanathan. An $O(\log^* n)$ approximation algorithm for the asymmetric p-center problem. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–5, Jan. 1996.

[124] H. Yang, L. Lee, and W. Hsu. Finding hot query patterns over an xquery stream. *The VLDB Journal*, 13(4):318–332, 2004.

[125] L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *Proceedings of the 29th VLDB Conference*, pages 69–80, 2003.

[126] A. C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.

[127] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree signatures for XML querying and navigation. In *LNCS 2824*, pages 149–163, 2003.

[128] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.

[129] K. Zhang, D. Shasha, and J. T. L. Wang. Approximate tree matching in the presence of variable length don't cares. *Journal of Algorithms*, 16(1):33–66, 1994.