IMPROVING ACCESS TO ORGANIZED INFORMATION

A Dissertation Presented

by

BRENT HEERINGA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2006

Computer Science

IMPROVING ACCESS TO ORGANIZED INFORMATION

A Dissertation Presented

by

BRENT HEERINGA

Approved as to style and content by:

Micah Adler, Chair

David A. Mix Barrington, Member

Robert Moll, Member

Dennis L. Goeckel, Member

W. Bruce Croft, Department Chair
Computer Science

For my parents, without whom I would have never finished this paper.

ACKNOWLEDGMENTS

Many people contributed positively to my graduate experience at UMass. Foremost, I want to thank my advisor, Micah Adler. Micah accepted my plea to be his research assistant without ever having me in class. He consistently gave me good advice about research and the academic community; he molded me into a scholar. His patience, support, guidance, clarity, collaboration and friendship made UMass a fulfilling and fun place to study.

I would also like to think my committee members, Dennis Goeckel, David A. Mix Barrington, and Robbie Moll, for their many comments and ideas. Dave, in particular, reviewed this thesis with a fine tooth comb; his suggestions gave clarity to the text. Robbie Moll offered substantial technical, moral, and avuncular support throughout the dissertation process. Fellow theory student Bill Hesse helped me with a convexity argument.

I started at UMass in the Experimental Knowledge Systems Laboratory under the tutelage of Paul Cohen. Paul laid the early groundwork for any success I achieve during my academic career. He kindly pushed me to publish early and taught me to write with simple, clear prose – much of the instruction coming via his own lucid writing. Tim Oates took me under his able wing at the start of graduate school. I thank him for including me in his research and trusting me to write code for his robot experiments. I hope and trust we will work together again soon. Clayton Morrison always offered great philosophical, technical, and musical discussions. He also provided sturdy emotional support during difficult times. Matt Schmill taught me how to give a good talk. David Westbrook was an impressive and effective lab therapist. I am also grateful to Marc Atkin, Brendan Burns, Gary King, Andrew

Hannon, Michael O'Neill, and Charles Sutton for our discussions of research, life, and Lisp.

In addition, I am thankful for the support and encouragement of several people at UMass. Jack Wileden often served as a surrogate advisor. Jen Neville and I dissected our graduate school experiences over many cups of tea and coffee. Dan Bernstein and I did the same except on the squash court.

Lastly, I want to thank Courtney Wade. She offered a careful eye to my research papers and an attentive ear to my talks. More importantly, she lent her full support to my thesis and academic pursuits. Undoubtedly, there are others whom I have missed thanking — my sincerest appreciation to everyone who helped along the way.

ABSTRACT

IMPROVING ACCESS TO ORGANIZED INFORMATION

SEPTEMBER 2006

BRENT HEERINGA

B.A., UNIVERSITY OF MINNESOTA, MORRIS

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Micah Adler

We introduce several new models and methods for improving access to organized information. The first model, Constrained Subtree Selection (CSS), has applications in web site design and the reorganization of directory structures. Given a hierarchy represented as a rooted DAG G with n weighted leaves, one selects a subtree of the transitive closure of G that minimizes the expected path cost. Path cost is the sum of the degree costs along a path from the root to a leaf. Degree cost, γ, is a function of the out degree of a node. We give a sufficient condition for γ that makes CSS **NP**-complete. This result holds even when the leaves have equal weight. Turning to algorithms, we give a polynomial time solution for instances of CSS where G does not constrain the choice of subtrees and γ favors nodes with at most k links. Even though CSS remains **NP**-hard for constant degree DAGs, we give an $O(\log(k)\gamma(d+1))$ approximation for any G with maximum degree d, provided that γ favors nodes with at most k links. Finally, we give a complete characterization of the optimal trees

for two special cases: (1) linear degree cost in unconstrained graphs and uniform probability distributions, and (2) logarithmic degree cost in arbitrary DAGs and uniform probability distributions.

The second problem, Category Tree (CT), seeks a decision tree for categorical data where internal nodes are categories, edges are appropriate values for the categories, and leaves are data items. CT generalizes the well-studied Decision Tree (DT) problem. Our results resolve two open problems: We give a $\ln n + 1$-approximation for DT and show DT does not have a polynomial time approximation scheme unless P=NP. Our work, while providing the first non-trivial upper and lower bounds on approximating DT, also demonstrates that DT and a subtly different problem which also bears the name decision tree have fundamentally different approximation complexity.

We complement the above models with a new pruning method for k nearest neighbor queries on R-trees. We show that an extension to a popular depth-first 1-nearest neighbor query results in a theoretically better search. We call this extension Promise-Pruning and construct a class of R-trees where its application reduces the search space exponentially.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

IMPROVING ACCESS TO ORGANIZED INFORMATION

1.1 Introduction

A central goal when organizing information is to create a system where the information is quick and easy to find. This frequently means arranging data so that the organizational structure is intuitive and the search procedure is efficient. For example, when saving email, it is common to store messages in a series of hierarchically arranged folders where each child folder is more specific than its parents. In this way one avoids searching through email about credit card payments when actually looking for directions to an old friend's house. With other types of data the search process plays a more significant role. For example, consider an organization of topological map data indexed by overlapping geometric features. Because the data share features, it is often necessary to inspect large portions of the organization when searching for a map. Consequently, it is advantageous to design algorithms that effectively remove large portions of the search space when performing a search. Unfortunately, search procedures sometimes ignore salient information. In addition, many organizations often become muddled: directory structures lose consistency, files clutter folders, and popular web pages become buried deep in a web site. This thesis explores models and methods for *automatically improving access to organized information.*

We concentrate on three closely related information organization problems. The first centers on the optimal arrangement of hierarchical data: Given some existing organization of topics and their respective popularity, create a hierarchy where navigation is natural and popular information is quickly and easily accessible. For example,

suppose *A Different Drummer's Kitchen* is developing a website for their catalog of kitchenware. They want a website where customers can quickly find information on specific products by descending a hierarchy of general to specific categories. They also want to minimize the number of intermediate pages it takes to find pepper mills, but not at the expense of filling a page with links to marginally related products like cookie cutters. To model this task, we develop a new problem called Constrained Subtree Selection (CSS). Intuitively, the goal of CSS is to build a hierarchy that minimizes the time searching for a topic but still reflects the semantics of the original organization. Automatic hierarchy design provides a principled choice for information organization, facilitates individualized and user-centric organizational structures, and decreases the average time spent searching for relevant information.

The second problem seeks an optimal arrangement of categorical data rather than hierarchical data. Many natural problems fall under this paradigm including the well-studied Decision Tree (DT) problem: Given a set of m binary tests $T = (T_1, \ldots, T_m)$ and a set of n items $X = (X_1, \ldots, X_n)$, construct a binary tree where each leaf is labeled with an item from X and each internal node is labeled with a test from T. If an item passes a test it follows the right branch; if it fails a test it follows the left branch. A path from the root to a leaf uniquely identifies the item labeled by that leaf. The depth of a leaf is the length of its path from the root. The total external path length of the tree is the sum of the depths of all the leaves in the tree. The goal of DT is to find a tree which minimizes the total external path length. Decision trees have applications ranging from experiment design to medical diagnosis to political surveying. We introduce a generalization of the decision tree problem, similar in spirit to CSS, called Category Tree (CT). Several of our results in CT apply to DT. In fact, two of our results resolve open problems concerning the polynomial time approximability of DT.

In addition to good arrangements of data, a principle component of improving accessibility in organized information is searching efficiently through the data. Our last problem focuses on improving the search for objects in high-dimensional spatial data. For example, consider the problem of organizing and retrieving video and music for the purpose of making recommendations, much like the systems provided by most online book and music retailers or home DVD services. If we characterize popular songs by some high-dimensional feature vector (where features have semantic meaning) then similar songs occupy locations near each other in feature space. Given a song, the goal is to search a collection of music and identify similar songs, or rather, to find the closest songs in feature space. This type of search is called a nearest-neighbor query. We introduce a new method for k nearest neighbor queries on a popular data structure called the R-tree that potentially reduces the search-space exponentially.

1.2 Contributions

The primary contribution of this thesis is a rigorous framework for studying common information organization problems. Two central, related questions often arise in these problems:

Question 1: How might one organize information so that navigation is natural and data is easily accessible?

Question 2: How might one efficiently search through organized information?

In response to Question 1 we develop a set of problems which naturally model information organization tasks like building websites (so popular pages are easy to find) and creating good decision structures (so finding information is easy and efficient). The problems include a novel cost function that measures search time in terms of deliberation. In response to Question 2, we contribute a new pruning method that

improves the state-of-the-art depth-first search nearest-neighbor query in the popular R-tree data structure. We provide a new tree construction to demonstrate this algorithmic improvement.

In some cases, our results answer problems open over the last thirty years or resolve historically confusing claims. In other cases, we contribute algorithms and complexity results for the new models. Collectively, these models, algorithms and analyses lay a rigorous foundation for future research on information organization problems. In the following sections we describe our contributions in more detail.

1.3 Models for Organizing Information

An integral component of improving access to organized information is finding a good initial organization. In this section we develop a new organizational model for hierarchal data called Constrained Subtree Selection (CSS). We also extend an existing model for finding good decision structures. We call this model Category Tree (CT).

1.3.1 Organizational Structures

Constrained Subtree Selection (CSS) models organizational tasks where the data are hierarchical. Category Tree (CT) models tasks where the organizational data are categorical. Both CSS and CT seek a search tree where the leaves are data (topic pages in the case of CSS and items or labels in the case of CT), internal nodes are deliberation points, and edges are meaningful choices. The technical difference between CSS and CT is in how they constrain the data. In the case of CSS a graph defines the constraints; in the case of CT category membership defines the constraints. Below we explain and motivate each type of constraint.

1.3.1.1 Organizations Constrained by Graphs

Recall that Constrained Subtree Selection (CSS) models tasks where hierarchical data are reorganized so that popular items are easily accessible but not at the expense of inferring new structure which violates the semantics of the original hierarchy. An example of this type of task is website design: Suppose that prior to site development, topics are hierarchically arranged by a designer to represent their natural organization. We represent this initial hierarchy as a rooted, directed acyclic graph, called the *constraint graph* where the nodes are categories, the leaves are topics, and the edges are topical constraints. A path through the constraint graph follows a general to specific trajectory through the categories. For example, in the kitchenware hierarchy cutlery leads to knives leads to paring knives. Note that a particular paring knife may belong to other categories (like the knife manufacturer), and thus the constraint graph may be a DAG that is not a directed tree. Solutions to CSS are subtrees of the transitive closure of the constraint graph that include the root and all the leaves. In this way, intermediate pages may be skipped but no new pages are created. We stipulate that the subtree include the root and leaves of the constraint graph since they represent the entry and endpoints of any natural descent in the hierarchy.

Websites are not the only realization of this model. For example, consider creating and maintaining user-specific directory structures on a file system. One can imagine that the location of `/etc/httpd` may be promoted to the root directory for a system administrator whereas a developer might find `~/projects/source` directly linked in their home directory. Similarly, users may have individualized views of network filesystems targeted to their own computing habits. In this scenario a canonical version of the network structure is maintained, but the CSS problem is tailored to the individual. In general, any hierarchical environment where individuals actively use the hierarchy to find information invites modeling with CSS.

1.3.1.2 Organizations Constrained by Categories

Category Tree resembles CSS, but uses categories instead of a graph to constrain the organization. That is, instead of viewing each item as a leaf in a constraint graph, we view each item as a string of length m over a finite alphabet of size k where each character position represents a distinct category and the character at that position maps to appropriate categorical value . For example, consider the problem of organizing a collection of digital films suitable for human search. Here the categories might be *title*, *year*, *director*, *film length*, and *genre* and an appropriate item might be *(Rushmore, 1998, Wes Anderson, 93 min., Comedy)*. Given a set of n distinct items over m categories a *valid* category tree is any decision tree with n leaves where each node is a category, each edge is a valid value for its parent category, and each item is a leaf. In this way, a path from the root to a leaf yields a set of categories and values that completely disambiguates the leaf item from the other items. Put differently, a valid category tree distinguishes each item from one another consistent with the categorical constraints.

1.3.2 Deliberation Cost and Weights

One novel contribution of Constrained Subtree Selection and Category Tree is the cost function; we view the cost of a path in terms of the degrees of the nodes along the path (instead of just the length of the path) because it naturally models the time it takes to find an item of interest or make a correct choice. This seems reasonable since each internal node represents a decision, and deliberation time is typically a function of how many options are available. Certainly this is natural, for example, when determining the correct link to take on a web page or choosing the right genre when searching for a movie.

More specifically, we say the path cost is the sum of the degree cost of the nodes along the search path. The cost of the tree is the sum of all the path costs. We

represent node cost, γ, as a function of the number of outgoing links of the node, so it is called the *degree cost*. Adding more links decreases the height of the tree, but increases the time spent searching a node for the correct link; minimizing the number of links on a page makes finding the right link easy, but adds height to the tree. For this reason, the degree cost may be thought of as capturing the inherent tension between breadth and depth. Different scenarios demand different tradeoffs between these competing factors. For example, if network latency is a problem when loading web pages then favoring flatter trees with many links per page decreases idle waiting. By contrast, web browsers on handheld devices have little screen area, so to reduce unnecessary scrolling it is better to decrease the number of links in favor of a deeper tree. In the spirit of generality, we attempt to keep the results degree-cost independent. At times however, we examine particular degree costs such as logarithmic, linear, and constant. We often refer to the expected path cost of a tree as the *deliberation cost* because each node is a deliberation point.

Naturally, some items are more popular than others. We capture this aspect with a probability distribution over the items, or equivalently by a set of item weights. Given a path, the weighted path cost is the sum of the node costs along the path (*i.e.* the unweighted path cost) multiplied by the item weight. Since we want a structure that minimizes the average search time for an item, we take the cost of a tree as the expected path cost for an item chosen from the probability distribution over the items. In the case of CSS, an optimal tree is any minimal cost subtree of the transitive closure of the constraint graph that includes the leaves and root. In the case of CT, an optimal tree is a valid tree with minimal cost.

1.3.3 Problem Complexity

We give a sufficient condition on the degree cost which makes Constrained Subtree Selection **NP**-complete in the strong sense. Many natural degree costs (e.g., linear,

exponential, ceiling of the logarithm) meet this condition. This result holds even for the case of uniform leaf weights. When the leaf weights are allowed to vary even more instances become hard. For example, the depth-one tree is always optimal for equal leaf weights, arbitrary graphs, and the logarithmic degree cause. Naturally it does not meet the sufficient condition outlined above for **NP**-completeness. However, when the leaf weights are allowed to change, the problem for the logarithmic degree cost is **NP**-complete. This means that the probability distribution adds some complexity to the problem.

Not surprisingly, the Category Tree problem is also **NP**-complete. We show that DT instances do not admit a polynomial time approximation scheme unless P=**NP**. The reduction is from MAX-3SAT5 and relies on constructing a set system where each set has exactly 6 items (*i.e.*, the set system is 6-uniform).

1.3.4 Algorithms and Approximations

Because of the negative results above, we turn our attention to restricted scenarios and approximation algorithms. For CSS, we first consider the case of inputs where the topological constraints of the graph are removed (*i.e.*, where the constraint graph allows any website tree to be constructed). Within this scenario, we give an $O(n^{k+\gamma(k)})$ time algorithm for finding an optimal tree when the topological constraints of the graph are removed and when γ is integer valued and always favors nodes with at most k links. This result holds for arbitrary leaf weights and demonstrates that the constraint graph also imposes computational hardness of the CSS problem. We also provide an exact characterization of the optimal solution for the linear cost function in the case of a uniform probability distribution and no topological constraints.

We next consider the case of bounded out-degree constraint graphs. When γ favors complete k-ary trees, CSS remains **NP**-hard for graphs with degree at most $k + 5$ and uniform leaf weights. However, there is a polynomial time constant factor

approximation algorithm for constraint graphs with degree no greater than d and arbitrary leaf weights, provided that γ favors nodes with at most k links for some k. The approximation ratio depends on both d and γ. Additionally, we show the linear degree cost favors complete 3-ary trees. We also demonstrate that the depth-one tree approximates (within an additive constant of 1) instances of CSS where $\gamma(x) = \lceil \log_2(x) \rceil$, the leaf weights are equal and the constraint-graph is arbitrary, even though this case is **NP**-compete.

For CT, we focus on a class of instances where the item weights are equal, the degree cost is positive and non-decreasing, and the alphabet is binary. We give a $\ln n +$ 1-approximation for these instances of CT. Instances of this type include the decision tree problem (DT). Hence, our result provides the first non-trivial approximation for DT and, as a consequence, also demonstrates that DT and a subtly different problem which also bears the name decision tree have fundamentally different approximation complexity.

1.4 Efficient Search Procedures

Integral to improving access to organized information is an efficient search procedure. In some cases, the combination of query type and organizational structure alone produce a proficient procedure. For example, when using a decision tree to diagnose a disease, one always starts at the root and proceeds down the tree following a single path. Each step downward yields a new test, the result of which uniquely determines the next step. The process ends at a leaf with a (hopefully) correct diagnoses. However, in many other structures natural searches require investigating multiple paths or disparate areas of the organization. This is the case, for example, in range queries on a B-Tree or nearest neighbor queries in an R-tree.

Our final result is an extension to a popular depth-first 1-nearest neighbor query in R-trees called Promise-Pruning. Our new pruning strategy potentially reduces the

search space exponentially when compared with the same algorithm alone. It relies on a new tree construction where a query with PROMISE-PRUNING makes $O(\log n)$ queries but the query without PROMISE-PRUNING makes $O(n)$ queries.

We also provide an additional, related construction that clarifies recent results from the literature dismissing certain pruning strategies as ineffective.

CHAPTER 2

RELATED WORK

Constrained Subtree Selection is related technically to the Hotlink Assignment problem and Optimal Prefix-free Code problem and conceptually to work in the AI community on adaptive websites. Category Tree generalizes a classic decision tree problem formalized rigorously in the late 1960s. It also bears resemblance to the Set Cover problem and a recent variant called the Min Sum Set Cover problem. Finally, PROMISE-PRUNING builds on existing work from the mid-to-late 1990s in nearest-neighbor algorithms for the R-tree.

2.1 Adaptive Websites

Perkowitz and Etzioni [40] introduced the idea of an adaptive website to the AI community in hopes of challenging them with a large-scale research problem. While the authors are concerned with many issues related to building intelligent websites, they concentrate on the *index page synthesis problem* which seeks to "automatically generate index pages to facilitate efficient navigation of a site or to offer a novel view of the site." They propose new clustering and concept learning algorithms which harness the access logs of the website to accomplish this goal. Here "efficient" means guaranteeing visitors find their topic of interest (recall) and minimizing the amount of time spent finding that topic (effort). The time spent finding a topic is measured by the time it takes to scan successive pages for the right link and the overall number of links taken. Notice that their definition of effort strongly resembles our notion of

cost. In this light, our work supplies a formal model for the index page synthesis problem as it relates to minimizing the average effort in finding the topic of interest.

2.2 Hotlink Assignment

The Hotlink Assignment (HA) problem introduced by Bose et al. [5] attempts to improve access to information on a website by adding shortcuts to popular pages. Here, a website is represented by a directed graph. Popularity is given by a probability distribution over the leaves. The problem seeks to assign at most one arc, called a hotlink, to each node to minimize the expected distance from the root to the leaves. Since multiple paths from the root to a leaf may exist, the expected distance is computed using the shortest path. The problem is **NP**-hard for directed graphs, but nothing is known about the problem when the graph is a tree and the weights are arbitrary. When the graph is a complete binary tree, the authors give upper and lower bounds for cases where the probability distribution is uniform, arbitrary, Zipfian, and geometric. They show matching upper and lower bounds in the uniform case and upper and lower bounds differing by only a constant in the geometric case.

In later work Kranakis *et al.* [34, 35] give a quadratic-time approximation algorithm for trees with maximum degree d. A lower bound on the optimal cost is essentially entropy. This follows closely from Shannon's theorem. Their algorithm achieves a cost within an additive constant of optimal for any fixed d. The idea is to propagate the leaf weights up to the parent's parent: an internal node's weight is the sum of its children's weights. There is always a node in the tree with weight one-third to two-thirds the total weight of the tree. The algorithm creates a shortcut from the root to this node and then applies the same procedure to its subtrees until it has reached the leaves. We incorporate a similar procedure into an approximation algorithm for the Constrained Subtree Selection problem.

Advancement on the initial HA work is limited. Fuhrmann et. al. ([21]) introduce a variant called the (h, k)-hotlink assignment problem where the input is a k-regular complete tree and one can assign at most h hotlinks to each node. Bose et. al. [6] show that hotlink assignment relates strongly to asymmetric communication protocols. There are also some empirical studies of heuristic algorithms for the problem [14, 33].

Hotlink Assignment is different from CSS for a number of reasons. The first is how we model page cost. In HA, page cost does not change with the addition of hotlinks. In CSS, the cost of a page is a function of the number of links it contains. This means we can think of CSS as minimizing the expected amount of choice a user faces when traversing a website as opposed to HA, which minimizes the expected number of hops. Recently, the problem was revised so that nodes have a fixed page cost proportional to the size of the web page they represent [15]. In this formulation, the cost of a path is not its length, but instead the sum of the page costs on the path. Adding shortcuts in this model minimizes the expected latency, but shortcut additions don't change page costs. Contrast this with the direct interplay between degree and page cost in the CSS model. Also note that the generality of our degree function means we can also include a network latency term into the degree cost.

Another difference is how we view the initial topologies. With HA, the graph represents a website that needs improvement. In CSS, we take the DAG as a set of constraints for building a website. This difference is both conceptual and technical. While the *shortest path* tree can be extracted from the Hotlink graph after the links are assigned, a tree with longer paths cannot be considered. We consider all paths in the subtree selection since longer paths are viewed in terms of constraints and not cost. Finally, HA assigns a constant number of hotlinks to each node where CSS has no restriction. The constant number is important to HA because without this restriction, the optimal website would always have hotlinks from the root to all the leaves. In CSS this corresponds to a constant degree function where the optimal tree

is always the depth-one tree. Ultimately, hotlink assignment and constrained subtree selection are fundamentally different problems.

2.3 Prefix-free Codes

Certain relaxed versions of the Constrained Subtree Selection problem bear resemblance to the Optimal Prefix-free Coding (OPC) problem: The general version asks for a minimal prefix code for n weighted words using at most r symbols where symbol i has cost c_i ([31], [24]). This problem is equivalent to finding a tree with n leaves where all internal nodes have degree at most r, the length of the i^{th} edge of a node is c_i, and the external weighted path length is minimized. There is no known polynomial time solution for the general problem, but it is not known to be **NP**-hard. When the costs are restricted to fixed integers, there is an $O(n^{C+2})$ time dynamic programming algorithm where C is the maximum integer cost [25] and when $r = 2$ (*i.e.*, a binary alphabet) the problem has a $O(n^C)$ solution. When the symbol costs are are equal (*i.e.*, $c_1 = c_2 = \cdots = c_r$) the problem becomes the Huffman coding problem which has a $O(n \log n)$ time greedy solution. When the leaves weights are equal, but the symbol costs unequal, the problem is called Varn coding. There is an $O(n \log r)$ time solution to this problem [11] which improves earlier work by Golin and Young ([26]) by a factor of $\log r$.

On the surface, the problems appear similar because they both ask to minimize external weighted path cost—the sum of weighted path costs from the root to each of the leaves. However the cost in OPC is edge-based, where the cost of CSS is node-based, and perhaps more importantly, a function of the out-degree of the node. If one views the node costs as edge costs, than adding an edge potentially changes the edge costs of all its siblings. This difference, along with the lack of prior constraints on the tree structure in prefix-free codes, distinguish the problems enough that it seems difficult to transform one to the other. The only case where this isn't true is cases

of CSS where the graph is constraint-free and the degree cost always favors binary trees. Since every optimal tree is a binary tree, we push the node costs down to the edges and the problem becomes standard Huffman codes with equal letter costs.

Most prefix-free coding algorithms use lopsided trees. Lopsided trees are a representation of trees with fixed edge costs such that a node appears at the depth of its total cost from the root. For example, if we consider binary lopsided trees where the left branch has edge cost 1 and the right branch has edge cost 2, then the leftmost grandchild of the root appears at depth 2 while the rightmost grandchild appears at depth 4. In general, there are at most r edges per node, each with a fixed value depending only on its order. That is, the first child has cost c_1, the second child cost c_2 and so on with $c_i \leq c_{i+1}$ for all $1 \leq i \leq r$. Perhaps the most important property of these lopsided trees is that adding an edge does not change the prior structure of the tree. This means that one can view the optimal tree as a cut of the infinite lopsided tree. In other words, the infinite lopsided tree may be viewed as a static structure from which the optimal tree is a rooted subtree. This approach does not work in CSS because cost is a function of node degree; remove an edge and the lopsided tree changes structure. While we cannot take advantage of this cut property, there are other benefits which we exploit in Section 3.4 to help design algorithms for certain cases of CSS.

2.4 Decision Trees

Category Tree is a type of decision tree problem where the optimization criterion is deliberation time and the the categories represent tests with at most k outcomes. The decision tree literature is extensive and diverse [39]. Work in the artificial intelligence community focuses on learning trees to classify attribute/value data especially with an eye toward disambiguation [43]. In other words, one begins with a set of labeled training data and attempts to build a tree which correctly labels the data and

successfully generalizes to other data. This is the goal, for example, of the well-known ID3 algorithm [41]. Recent work in this area concentrates on finding lower bounds on the approximation ratio of a problem we call ConDT, for consistent decision tree.

The input to ConDT is a set of n positive / negative labeled binary strings, each of length m, called examples (note: many papers take m to be the number of examples and take n to be the number of bits). The output is a binary tree where each internal node tests some bit i of the examples, and maps the example to its left child if i is a 0 and its right child if i is a 1. Each leaf is labeled either TRUE or FALSE. A consistent decision tree maps each positive example to a leaf labeled TRUE and each negative example to a leaf labeled FALSE. The size of a tree is the number of leaves. ConDT seeks the minimum size tree which is consistent with the examples.

Alekhnovich et. al. [1] show it is not possible to approximate size s decision trees by size s^k decision trees for any constant $k \geq 0$ unless **NP** is contained in DTIME$[2^{m^\epsilon}]$ for some $\epsilon < 1$. This improves a result from Hancock et. al. [28] which shows that no $2^{\log^\delta s}$-approximation exists for size s decision trees for any $\delta < 1$ unless **NP** is contained in quasi-polynomial time. These results hold for $s = \Omega(n)$.

Our results demonstrate that DT and ConDT – although closely related – are quite different in terms of approximability: The results above imply ConDT has no $c \ln n$-approximation for any constant c (unless **P** =**NP**) whereas our results yield such an approximation for DT for $c > 1$. Also, we show that the lower bounds on learning decision trees of the ConDT type hold when minimizing total external path length instead of minimum size. Note that tree size is not an insightful measure for DT since all feasible solutions have n leaves. Thus, it is the difference in input and output, and not the difference in measure, that accounts for the difference in approximation complexity.

Not surprisingly, the difference in approximation complexity between DT and ConDT, combined with the ambiguity of the name decision tree has caused confusion

in the literature. For example, [2] and its online incarnation [13], define the decision tree problem according to the DT input and output but cite the negative results for ConDT in Hancock et. al [28]. Therefore, we consider the separation of DT and ConDT in terms of approximation complexity one contribution of our work.

Moret [36] views DT and ConDT as separate instances of a general decision tree problem where each item is tagged with k possible labels. With DT there are always $k = n$ labels, but only one item per label. With ConDT, there are only two labels, but multiple items carry the same label. These problems are unified via decision trees over so-called *information systems*. A finite information system $U = (A, B, F)$ is a finite set A, a finite set B with at least 2 members, and a collection of functions F from A to B. Each function f_i in F is called an attribute, but can be thought of as a test or category. A problem over u is an $(n+1)$-tuple $z = (v, f_1, \ldots, f_n)$ such that $v : B^n \to \omega$ where $\omega = \{1, 2, \ldots\}$. The problem z partitions A into equivalence classes such that $a_j = a_k$ if and only if $z(a_j) = z(a_k) = v(f_1(a_j), \ldots, f_n(a_j)) = v(f_1(a_k), \ldots, f_n(a_k))$. A decision tree for U is a finite, rooted tree where each leaf is labeled with a member from ω, each internal node is labeled with one of the f_i, and each edge is appropriately labeled with some value from B. Trees for information systems are analyzed for time and space complexity measures like worst-case time (depth of the tree), average-case time (average depth) [9, 10] and minimum number of nodes [8]. We are unaware of any non-trivial results bounding the approximation ratio of decision trees for information systems.

Another line of research uses decision trees as a tool for proving computational lower bounds [20, 48]. Here, problems are cast as membership queries on disjoint sets $S_1, \ldots, S_m \subseteq \mathbb{R}^n$. An algebraic decision tree for a problem labels each leaf with one of the sets S_i and each internal node is some polynomial function of the input. The internal nodes act as k-ary predicates and the appropriate outgoing edge is taken depending on the result. The worst-case time-complexity of the tree is the length

of the longest path; that is, the height of the tree. In most of these cases, the tree is an abstraction of a real algorithm since the computation at the internal nodes is considered constant and thus doesn't add to the complexity.

Many authors consider complexity-related questions for various restrictions on the branching factor, restrictions on the type of functions allowed at the internal nodes, and the dimension of input. For example, Moshkov and Chikalov [37] strictly study boolean functions where internal nodes may only examine one bit of the input. This line of work does not directly impact our problem however the general lower-bounds on tree size may help in the development of good approximation algorithms for CT.

With respect to upper bounds, Ehrenfeucht and Haussler [16] show that decision trees on n boolean variables (which have size polynomial in n) are PAC-learnable in $n^{O(\log(n))}$ time. Like CSS with 2-favorable degree costs, when we restrict CT to binary categories, then the optimal trees are always binary, so the cost function is, in most cases, irrelevant.

2.5 Set Cover

DT shares some similarities with set cover. Since each pair of items is separated exactly once in any valid decision tree, one can view a path from the root to a leaf as a kind of covering of the items. In this case, each leaf defines a set cover problem where it must cover the remaining $n-1$ items using an appropriate set of bits or tests. In fact, our analysis is inspired by this observation. However, in the decision tree problem, the n set cover problems defined by the leaves are not independent. For example, the bit at the root of an optimal decision tree appears in each of the n set cover solutions, but it is easy to construct instances of DT for which the optimal (independent) solutions to the n set cover instances have no common bits. More specifically, one can construct instances of DT where the n independent set cover problems have solutions of size 1, yielding a decision tree with cost $\Theta(n^2)$, but where the optimal decision tree has

cost $O(n \log(n))$. Hence, the interplay between the individual set cover problems appears to make the DT problem fundamentally different from set cover. Conversely, set cover instances naturally map to decision tree instances however, the difference in cost between the two problems means that the optimal set cover is not necessarily the optimal decision tree.

The min sum set cover (MSSC) problem is also similar to DT. The input to MSSC is the same as set cover (*i.e.*, a universe of items X and a collection C of subsets of X), but the output is a linear ordering of the sets from 1 to $|C|$. If $f(x)$ gives the index of the first set in the ordering that covers x then the cost of the ordering is $\sum_{x \in X} f(x)$. This is similar, but not identical, to the cost of the corresponding DT problem because the covered items must still be separated from one another, thus adding additional cost. Greedily selecting the set which covers the most remaining uncovered items yields a 4-approximation to MSSC [19, 38]. This approximation is tight unless $\mathbf{P} = \mathbf{NP}$. As with set cover, we can think of DT as n instances of MSSC, but again, these instances are not independent so the problems inherent in viewing DT as n set cover problems remain when considering DT as n instances of MSSC.

2.6 Pruning in R-trees

R-trees were developed in the mid-80s to address the storage and retrieval demands of multi-dimensional geometric data [27, 4]. Ten years later, the first nearest-neighbor algorithms for R-trees appeared in the literature. Two popular nearest neighbor algorithms for the R-tree are the branch and bound depth-first search [42] algorithm (denoted by RKV) and the best-first search [29] algorithm (denoted by HS). The HS algorithm is optimal with respect to the number of page accesses (*i.e.*, nodes examined) but has worst case space complexity that is linear in the total number of tree nodes. On the other hand, the RKV algorithm, while not optimal in terms of page accesses, has a worst case space complexity that is logarithmic in the number of

tree nodes. In addition, [4] note that statically constructed indices map all pages on a branch to contiguous regions on disk, so a depth-first search may "yield fewer disk head movements than the distance-driven search of the HS algorithm." Hence RKV may be preferable to HS not only in terms of space complexity but also for reasons of performance.

The original RKV algorithm uses three different strategies (here called S1, S2, and S3 respectively) to prune branches. S3 is based purely on a metric called MINDISTANCE which gives the actual distance between a page and a query point. S1 and S2 use a second metric called MINMAXDISTANCE which provides an upper bound on the distance between an actual object in a page and a query point. All the pruning strategies are defined for 1-nearest neighbor. The authors provide suggestions for extending the strategies to k nearest neighbors. For S3 and S1, the extensions are straight-forward, however, as [4] points out, the details surrounding an extension of S2 are less clear. Results from [29] and [7] appear to render the details meaningless: Both papers show that S1 is redundant modulo sorting pages by MINDISTANCE. Furthermore, both papers dismiss S2 as unnecessary since it does not directly prune branches. Still, S2 *does* update the nearest neighbor estimate, which *can* facilitate future pruning. As a result, 1-nearest neighbor searches with S2 may prune more branches than searches without it. In fact, we show that RKV with S2 can potentially reduce the search space *exponentially* over RKV without it. This demonstrates the tremendous benefit of MINMAXDISTANCE on depth-first search 1-nearest-neighbor queries. Our Promise-Pruning algorithm correctly generalizes S2 to k nearest neighbors. This algorithmic extension to RKV is essential to pruning in light of the of the potential search space reductions offered by MINMAXDISTANCE.

[4] suggest another, somewhat similar extension to RKV based on MINMAXDISTANCE and MAXDISTANCE — a metric that gives an upper bound on the distance between all objects in a page and a query point – that may reduce the search space,

although a proof of the claim is omitted. Since PROMISE-PRUNING is a restriction on their extension, our exponential search space reduction result also applies to their method.

CHAPTER 3

CONSTRAINED SUBTREE SELECTION

In this chapter we formally define Constrained Subtree Selection, explore the problem's complexity, develop a dynamic-programming algorithm for certain restricted instances, describe an approximation algorithm for some **NP**-Hard instances and end with some exact characterizations of optimal solutions for two different degree costs.

3.1 Definitions

Let T be a directed tree (a branching) with n leaves where leaf u_i has weight w_i. Let $(u_{i_1}, \ldots, u_{i_m})$ be a path from the root of T to the i^{th} leaf of T. If $\delta(v)$ is the out-degree of node v and γ is a function from the positive integers to the reals, then the cost of u_i is:

$$c(u_i) = \sum_{j=1}^{m-1} \gamma(\delta(u_{i_j}))$$

and the weighted cost is $w_i \cdot c(u_i)$. The cost of T is the sum of the n weighted paths:

$$c(T) = \sum_{i=1}^{n} w_i \cdot c(u_i)$$

An instance of the *Constrained Subtree Selection* problem is a triple $I = (G, \gamma, (w_i))$ where G is a rooted, directed, acyclic *constraint* graph with n leaves, γ is a function from the positive integers to the non-negative reals, and $(w_i) = (w_1 \ldots w_n)$ are non-negative, real-valued leaf weights summing to one. A solution to I is a directed subtree T (hereafter a tree) of the transitive closure of G that includes the leaves and root of G. An optimal solution is one that minimizes the cost function under

γ. Sometimes we consider instances of CSS with fixed components. For example, we might study the problem when the degree cost is always linear, or leaf weights form a uniform probability distribution. These cases are called *CSS with* γ or *CSS with equal leaf weights* so that it is clear that γ and (w_i) are not part of the input.

The following definitions are useful when discussing CSS:

Definition 1 (full tree)**.** *A tree is* full *when every interior node has at least two children.*

Definition 2 (constraint-free graphs)**.** *A constraint graph G with n leaves is called* constraint-free *when every full tree with n leaves is a subtree of the transitive closure of G.*

Definition 3 (monotone tree)**.** *A tree is* monotone *when the leaf weights cannot be permuted (among the leaves) to yield a tree of less cost.*

Definition 1 is self-explanatory. Definition 2 means that G does not constrain the optimal subtree. Definition 3 says that if we listed the leaves in increasing order by path cost, the weights of the leaves would be in decreasing order. From these definitions it's easy to see that every instance of CSS has at least one optimal solution that is full (and that when γ is not the trivial function $\gamma(x) = 0$ that every solution is a full tree) and that all solutions to CSS are monotone when the the graph is constraint-free.

The following definition is also useful because it provides a bound on the out-degree of any node in an optimal solution to the CSS problem where the graph is constraint-free.

Definition 4 (k-favorability)**.** *A degree cost γ is* k-favorable *if and only if there exists $k > 0$ such that any instance of CSS where G is constraint-free has an optimal solution under γ where the out-degree of every node is at most k.*

A natural method for showing k-favorability is to directly apply the definition: suppose that a node exists with degree greater than k where the children of that node are roots of subtrees with arbitrary weights. Split the node into two subtrees and add a new root. If the cost of the tree doesn't increase, then the tree is k-favorable. We use this technique below to show that the linear degree cost, $\gamma(x) = x$, is 3-favorable, although in general, the complexity (or even computability) of determining whether a degree cost is k-favorable remains an open question.

Lemma 1. *The linear degree cost, $\gamma(x) = x$, is 3-favorable.*

Proof. Let $I = (G, \gamma, (w_i))$ be an instance of CSS where G is constraint-free and $\gamma(x) = x$ and T an optimal tree for I. When $\delta(v) \geq 4$ for some node v in T we can split the node's children proportionally among two additional nodes, connect them to the original root and not increase the cost since

$$\gamma(x) = x \geq p(\lceil x/2 \rceil + 2) + (1 - p)(\lfloor x/2 \rfloor + 2)$$

where p is the proportion of weight in the left child of the root and $(1 - p)$ is proportion of weight in the right child of the root. □

Interestingly, the linear degree cost is not 2-favorable. When a node has out-degree 3 and none of the three subtrees has probability greater than the sum of the remaining two subtrees (*i.e.*, the weight of the subtrees is somewhat uniform), we cannot transform the degree 3 node to a series of degree 2 nodes without increasing the overall cost of the tree. To see this, imagine the binary tree with three leaf nodes. Two paths from the root have length 2 and one path has length 1. The length 2 paths each have total cost 4, while the length 1 path has total cost 2. If one of the leaf nodes has weight over 1/2 we can choose it as the leaf on the length 1 path and the average path cost falls below 3, however when no such node exists it is advantageous to create a single node with three paths, each having cost 3 making the average cost 3

as well. It is worth noting that any instance of CSS where G is constraint-free and γ is 2-favorable reduces to the optimal prefix code problem for a binary alphabet with equal letter costs. In other words, Huffman's greedy algorithm (see [12]) solves these problems. Examples of degree costs that favor binary trees are $\gamma(x) = \lceil \log(x) \rceil$ and $\gamma(x) = e^x$.

3.2 Visualization of Results

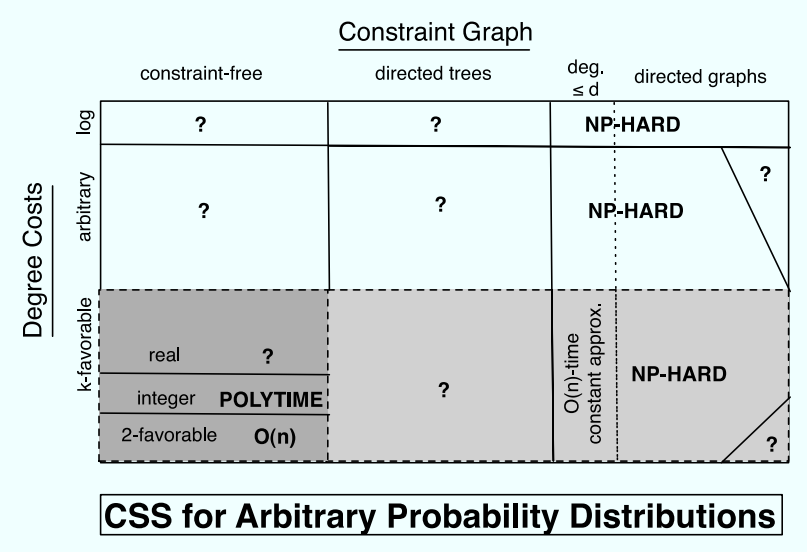

Figure 3.1. Time complexity results and open problems for instances of CSS under arbitrary probability distributions. The horizontal axis denotes changes to the constraint graph and the vertical axis denotes changes to the degree cost.

Figures 3.1 and 3.2 provide a visual display of the results and open problems in CSS. In particular, Figure 3.2 shows our results for instances of CSS with a uniform probability distribution and Figure 3.1 shows our results for instances of CSS with an arbitrary probability distribution. In the following sections we describe and explain these results.

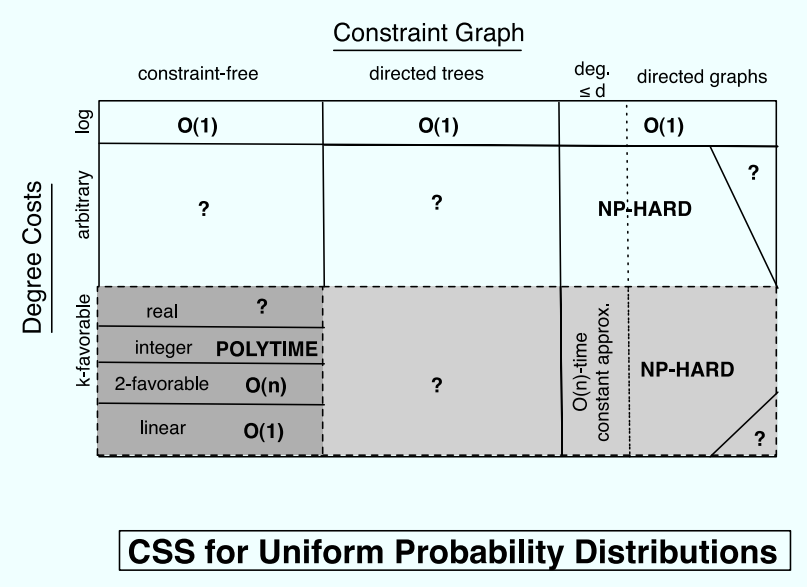

Figure 3.2. Time complexity results and open problems for instances of CSS under a uniform probability distribution. The horizontal axis denotes changes to the constraint graph and the vertical axis denotes changes to the degree cost.

3.3 Complexity

In this section we show that even when the leaf weights are equal, the CSS problem is **NP**-complete in the strong sense for a large class of degree functions. The reduction is from Exact Cover by 3-Sets (XC3) [23] which, when given a set X of $3k = n$ items and a set C of three item subsets of X, asks whether a subset of C exists that exactly covers X. The related decision problem for CSS asks whether a subtree of G exists with cost at most D.

Definition 5. *Let γ be a non-decreasing function. If for all positive integers k there exists some positive integer $s(k) \in k^{O(1)}$ such that*

$$\gamma(s(k) + k + 1) > \gamma(s(k) + k) + \gamma(3)\frac{3k}{s(k) + 3k},$$

then γ is degree-3-increasing.

Many degree costs are degree-3-increasing. For example, the linear degree cost, $\gamma(x) = x$, (choose $s(k) = 7k$), exponential degree cost $\gamma(x) = \exp(x)$ (again, $s(k) =$

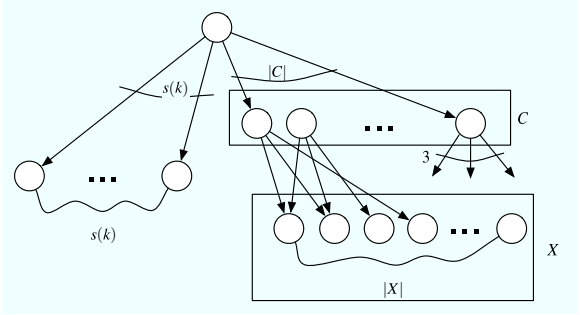

Figure 3.3. Setting up the constraint graph given an instance of Exact Cover by 3-sets.

$7k$ will work) and ceiling of the logarithm degree cost $\gamma(x) = \lceil \log_2(x) \rceil$ (choose $s(k) = 3k$) all meet the definition. The following theorem says that when γ is degree-3-increasing and in **NP**, that CSS with γ is **NP**-complete for any DAG and any probability distribution.

Theorem 1. *For any degree-3-increasing degree cost γ where γ is computable in non-deterministic polynomial time, CSS with γ is* **NP***-complete.*

Proof.

Lemma 2. *For any degree-3-increasing degree cost γ, CSS with γ is* **NP***-hard.*

Proof. The reduction is from Exact Cover by 3-sets (XC3). Let (X, C) be an instance of XC3 where $|X| = 3k = n$ with k a positive integer, and C is a collection of 3-element subsets of X. Furthermore, let γ be degree-3-increasing. We build a constraint graph such that (X, C) has an exact cover if and only if there is a subtree with cost at most

$$D = (s(k) + n)\gamma(s(k) + k) + n\gamma(3)$$

The idea is to create a constraint graph that affords a low cost solution when an exact cover exists, but increases in cost when no exact cover is available. Keeping this in mind, construct the constraint graph $G = (V, E)$ from (X, C) in the following way:

- Make a leaf node for each $x \in X$.

- Make an interior node for each $c \in C$.
- Make $s(k)$ additional leaf nodes.
- Make a root node r and connect it to the $s(k)$ leaf nodes and the c interior nodes.
- For each $c = \{x_i, x_j, x_k\} \in C$, create edges from c to x_i, x_j, and x_k.

The topology of the constraint graph is given pictorially in Figure 3.3. It has $s(k)+n$ leaf nodes and $|C|$ internal nodes, each having degree 3. Let $(G, \gamma, (w_i))$ be an instance of CSS where $w_i = 1$ for all i, G is the constraint graph formed from (X, C), and γ is degree-3-increasing.

Suppose (X, C) has an exact cover, then choose a subtree T from G by selecting the subtree with paths to the k interior nodes of G that partition X, along with their corresponding paths to the n leaf nodes. T must also have $s(k)$ paths to the remaining $s(k)$ leaf nodes. T has a root with degree $s(k) + k$ and k interior nodes each having degree 3. The length 1 paths each cost $\gamma(s(k) + k)$ and the n remaining paths, each have cost $\gamma(s(k) + k) + \gamma(3)$ making the total cost

$$(s(k) + n)\gamma(s(k) + k) + n\gamma(3)$$

Now suppose (X, C) does not have an exact cover. Any subtree of the transitive closure of G must have $s(k)$ edges to the $s(k)$ leaf nodes and at least $k+1$ additional edges to reach the remaining n leaves. Let e be the number of edges leading directly from the root to some subset of the remaining n leaves, let f be the number of edges from the root an interior node with degree 2, and let g be the number of edges from the

root to an interior node with degree 3. Note that $e+f+g \geq k+1$ and $e+2f+3g = n$. The cost of T is

$$(s(k)+n)\gamma(s(k)+e+f+g) + 2f\gamma(2) + 3g\gamma(3)$$

which, because γ is non-decreasing, is greater than or equal to

$$\begin{aligned}(s(k)+n)\gamma\Big(s(k)+k+1\Big) &> (s(k)+n)\Big(\gamma(s(k)+k) + \gamma(3)\frac{n}{s(k)+n}\Big) \\ &= (s(k)+n)\gamma(s(k)+k) + n\gamma(3) = D\end{aligned}$$

which is the cost of the subtree when an exact cover exists. □

Lemma 3. *For any non-deterministic polynomial time computable degree cost γ, CSS with γ is in* **NP**.

Proof. Let $I = (G, \gamma, (w_i))$ be an instance of CSS with γ in **NP** and D be an upper bound on the cost of a solution. Choose as the certificate, the optimal tree T and the certificates corresponding to the out-degree of each node in T for the polynomial-time verifiers of γ. It's easy to verify in polynomial time that T is a subtree of the transitive closure of G. Finally, we can use the polynomial-time verifiers for γ to verify the cost of each node and hence, verify the overall cost is at most D. □

The theorem follows immediately from the two previous lemmas. □

CSS is a number problem since γ, the probability distribution, and D are unbounded. We can remove the probability distribution from the input by tethering our attention to problems with equal leaf costs. Additionally, if we fix γ to be degree-3-increasing and in **NP** and further restrict $\gamma(s(n/2)+n/3)$ to be $n^{O(1)}$, then an encoding of CSS only involves the graph G. This encoding has length n. Instances of this type are **NP**-complete in the strong sense.

Theorem 2. *For any degree-3-increasing degree cost γ, γ in* **NP**, *if $\gamma(s(n/3)+n/3)$ is $n^{O(1)}$ then CSS with γ is* **NP***-complete in the strong sense.*

Proof. From Lemma 2 it's clear CSS remains **NP**-Hard when the n leaf weights are equal and since γ is fixed, we need not worry about its encoding. The only number we need to encode is the bound on cost. Any full tree with n leaves has at most $2n$ total nodes so the cost of the tree is bounded by a polynomial in n because γ is $n^{O(1)}$. This means the magnitude of the maximum element (either the cost or n) can be bounded above by a polynomial factor of the encoding of the instance. Since γ is degree-3-increasing and in **NP** we have that CSS remains in **NP** and the problem is **NP**-complete in the strong sense. Note that many degree costs make D polynomial in n. For example, $\gamma(x) = x$ keeps D quadratic in n. □

Finally, we note (without proof) that CSS is **NP**-Hard for many costs which are not degree-3-increasing (e.g., the logarithmic degree cost) when considering problem instances with an arbitrary probability distribution. The intuition here is that we can weight the $s(k)$ leaves heavily so that any unnecessary increase in degree at the root causes an unnecessary cost increase.

3.4 Subtree Selection without Constraints

Imagine we are building a website without any prior knowledge of the organization of the topics. The most natural solution is to build a website that minimizes the expected search time for the topics, but has no constraints on the topology. This design problem is an instance of CSS where any website is a subtree of the transitive closure of the constraint graph. In this section we show that these instances are solvable in polynomial time for a broad class of degree functions. This is interesting because it means that the graphical constraints add complexity to the problem.

The unconstrained CSS problem also bears a resemblance to the Optimal Prefix-free Code (OPC) problem. Both CSS and OPC attempt to minimize external weighted

path cost, but with CSS, path cost changes with a change in node degree. This distinction, along with the topological constraints, prevents us from casting CSS as an Optimal Prefix-free Code problem. Still, in many circumstances we can show instances of CSS that equate to finding optimal Huffman codes and that in other cases, we can give a polynomial time solution by adapting Golin and Rote's dynamic programming algorithm for optimal prefix-free codes with integer weights [25].

Theorem 3. *An instance $(G, \gamma, (w_i))$ of CSS where G is constraint-free and γ is k-favorable has an $O(n^{\gamma(k)+k})$-time solution.*

The solution uses a lopsided representation of the tree (cf. [11] for a discussion of lopsided trees and their history). With lopsided trees, a node's depth is equal to its path cost from the root. For example, a lopsided tree for the linear degree cost is given in Fig. 3.4 (a). We can view CSS trees as lopsided too, but for clarity retain the traditional meaning of depth (the number of edges between a node and the root) and use *level* to indicate a node's path cost. In other words, if $\mathcal{C}$ is the path cost from the root of T to some node v in T, then v is at *level* $\mathcal{C}$ and $\text{level}(v) = \mathcal{C}$.

A review of the dynamic programming algorithm for prefix-free codes is useful in explaining the CSS algorithm. The main idea is to grow lopsided trees level by level from the top down. The structure of the tree is captured with a signature which records the number of leaves up to the current level i and the relative levels of the nodes beyond level i. Nodes are only included beyond level i when they have a parent at or above level i. The structure of the tree beyond level i is called the frontier. One benefit of this representation is that the exact cost of the tree is known up to the frontier. This is because optimal trees are monotonic: higher leaves have larger probabilities. When a node is determined to be a leaf, the highest unassigned probability can be assigned to it and the exact cost of that path is known. The remaining paths are at least level i since they extend beyond level i, so the remaining mass contributes cost i to the total cost. Any two trees with identical signatures may

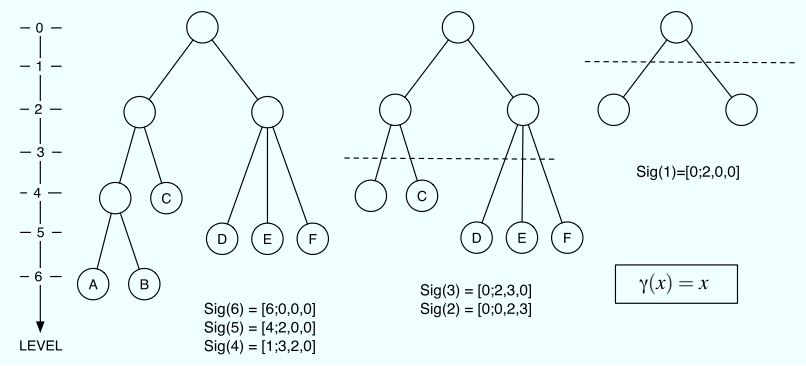

Figure 3.4. (a) A tree with 6 leaves depicted as a lopsided tree under the linear degree cost. Truncations at level-3 (b) and level-1 (c) of the tree in (a).

be thought of as equal since tree growth only occurs at the frontier. Hence, given two trees with identical signatures, we always prefer the one with smaller cost. An entry in the dynamic programming table corresponds to a tree signature. When growing a tree, the cost at one level can be written in terms of the cost of the previous level. This gives a natural method for filling in the table: Given a table entry, find all the entries which precede it structurally by single level and identify the one with the lowest cost. Use this parent entry to calculate the cost of the given entry. Repeat this process for all entries.

The CSS dynamic programming table is similar to Golin and Rote's but we are forced to fill in the entries differently. Notice that in the CSS problem changing the out-degree of a node changes the structure of the lopsided tree—adding or removing an edge from a node changes the cost of the entire subtree rooted at that node. With prefix-free trees, adding or removing an edge from a node has no effect on the existing tree. Golin and Rote exploit these properties when filling in table entries. We must take a different approach. For clarity and completeness, we give adapted definitions for *signature*, *truncation*, and *cost* of a tree, presented in [25].

Definition 6. *Let T be a tree. The* level-i-truncation *of T, denoted $Trunc_i(T)$, prunes away all nodes of T with parents at levels deeper than i.*

For example, the level-3- and level-1-truncations of the tree in Fig. 3.4 (a) are given in Figs. 3.4 (b) and (c) respectively. Truncation helps with the definition of tree cost and tree signature:

Definition 7. *Let T be a tree with n nodes, $v_1, \ldots, v_n$, given by level in increasing order. Let $w_1 \ldots, w_n$ be the leaf weights given in decreasing order. The* level-i-cost *of T is*

$$c_i(T) = \sum_{j=1}^{m} level(v_j) w_j + \sum_{s=m+1}^{n} i \cdot w_s$$

where m is the number of leaf nodes in $Trunc_i(T)$.

It's easy to see that the level-i-cost of that tree is a lower bound on the overall cost of a tree. It records the exact cost of the m leaves but only the cost to level i of the remaining $n - m$ leaves. When $m = n$, the cost of the tree is the same as the level-i-cost. We associate the cost of a tree with its signature:

Definition 8. *Let T be a tree. The* level-i-signature *of T is the $(\gamma(k) + 1)$ vector:*

$$sig_i(T) = (m, l_1, \ldots, l_{\gamma(k)})$$

where m is the number of leaf nodes at levels 0 through i and l_j is the number of nodes at level $i + j$ in $Trunc_i(T)$.

For example, the level-4-signature of the tree in Fig. 3.4 (a) is $(1, 3, 2, 0)$ and the level-3-signature of the same tree is $(0, 2, 3, 0)$. We equate the signature of a tree $(m, l_1, \ldots, l_{\gamma(k)})$ with an entry in the dynamic programming table $\text{MIN}[m, l_1, \ldots, l_{\gamma(k)}]$. This entry gives the minimum cost of all trees with signature $(m, l_1, \ldots, l_{\gamma(k)})$. Filling in the table is tantamount to finding the ways two trees with the same signature at level i can differ in their level-$(i + 1)$-signature.

Golin and Rote show that if T is a tree with level-i-signature $\text{sig}_i(T) = (m, l_1, \ldots, l_{\gamma(k)})$, and T' is a tree such that $\text{Trunc}_i(T') = \text{Trunc}_i(T)$ then $\text{Trunc}_{i+1}(T')$ differs from T in

at most l_1 ways. This is because nodes at levels-$(i+2)$ and deeper cannot have children under a level-$(i+1)$-truncation—only the nodes at level $(i+1)$ are candidates. Some number q of these nodes are internal, and the choice of q uniquely determines the signature at level $(i+1)$. This is not true of the CSS problem. Because node cost is dynamic, we must develop a process quite different from the ones used for the OPC problem. We are left then, not only with choosing how many of the level $(i+1)$ nodes will be internal, but with explicitly choosing among those, which will have degree 2, degree 3, and so on. These choices are denoted with a $(k+1)$-vector called a *child vector*:

Definition 9. *Let T be a tree with $sig_i(T) = (m, l_1, \ldots, l_{\gamma(k)})$. Let $a = (a_0, \ldots, a_k)$ be a* level-i-child vector *of T where a_0 is the number of nodes at level-$(i+1)$ that are internal to T and each a_j is the number among those a_0 having degree j.*

Definition 9 gives us a way to talk about how signatures at level i relate to signatures at level $(i+1)$. First, note that $a_0 \leq l_1$ and that $a_1 = 0$ since there is always an optimal tree without single out-degree nodes. Also, since $\sum_{j=2}^{k} a_j = a_0$ we know there are $O(n^{k-1})$ choices for a. In other words, given a level-i-signature, it is the possible parent of $O(n^{k-1})$ level-$(i+1)$-signatures. The following lemma tells us exactly which signatures are children of the level-i-signature.

Lemma 4. *Let T be a tree with $sig_i(T) = (m, l_1, \ldots, l_{\gamma(k)})$ and $a = (a_0, a_1, \ldots, a_k)$ be the level-i-child vector of T yielding T', then $sig_{i+1}(T') = (m', l'_1, \ldots, l'_{\gamma(k)})$ where*

$$(m', l'_1, \ldots, l'_{\gamma(k)}) = (m + l_1, l_2, \ldots, l_{\gamma(k)}, 0) + b$$

and $b = (b_0, \ldots, b_{\gamma(k)})$ where $b_0 = -a_0$ and $b_{\gamma(i)} = i \cdot a_i$ for $2 \leq i \leq k$

Proof. Let T and be a tree with $\text{sig}_i(T) = (m, l_1, \ldots, l_{\gamma(k)})$ and $a = (a_0, \ldots, a_k)$ be the level-i-child vector of T yielding T'. Given a, it is straightforward to compute $\text{sig}_{i+1}(T')$: we know exactly how many of the l_1 nodes are internal nodes at

level $(i+1)$ and the distribution of degrees among them. To calculate $\text{sig}_{i+1}(T') = (m', l'_1, \ldots, l'_{\gamma(k)})$ we shift the levels from $\text{sig}_i(T)$ to the left one index, subtract away the number of internal nodes a_0 from $m + l_1$, and then use a to count the new nodes at each level. Note that if there are a_j degree-j nodes then we must add an additional $j \cdot a_j$ nodes to level $\gamma(j)$. This gives us the $(\gamma(k)+1)$-vector:

$$b = (-a_0, b_1, \ldots, b_{\gamma(k)}) \text{ where } b_{\gamma(j)} = j \cdot a_j \text{ for } 2 \leq j \leq k$$

Note that $b_0 = -a_0$ since a_0 of the nodes are internal. Adding b component-wise to the shifted signature of T at level i gives us:

$$(m', l'_1, \ldots, l'_{\gamma(k)}) = (m + l_1, l_2, \ldots, l_{\gamma(k)}, 0) + b$$

□

As an example, consider the tree in Fig. 3.4 (a) and its level-3-signature $(0, 2, 3, 0)$. The level-i-child vector for this tree is $(1, 0, 1, 0)$ and since one of l_1 the nodes is internal, and it is a degree 2 node, we have $b = (-1, 0, 2, 0)$. Shifting the level-3-signature and adding b gives us $(2, 3, 0, 0) + (-1, 0, 2, 0) = (1, 3, 2, 0)$ which is exactly the level-4-signature.

While Lemma 4 tells us how level-i-signatures relate to level-$(i+1)$-signatures, it does not tell us how the costs relate. The second part of Lemma 5 from [25] tells us that if T is a tree with $\text{sig}_i(T) = (m, l_1, \ldots, l_{\gamma(k)})$ then

$$c_{i+1}(T) = c_i(T) + \sum_{j=m+1}^{n} w_j \tag{3.1}$$

Fortunately, this result holds for all monotone, lopsided trees with level-i-costs defined as in Def. 7. To see why, recall that the level-i-cost of T gives the exact weighted cost of all m leaves at or shallower than level-i in addition to the cost up to

level i of the remaining $n - m$ leaves. When moving from level-i to level-$(i + 1)$, we know $l_1 - a_0$ of the nodes become leaves and we need to record their exact cost, but the remaining $n - m - (l_1 - a_0)$ leaves must also incremented by a level in the cost structure. Taken together, we need to update the $n - m$ deepest paths which gives the summation term in 3.1.

We have now established a method for filling in the dynamic programming table. What is left to give is an ordering of the table entries that is consistent with their dependency structure. Golin and Rote give a linear ordering of the table entries and demonstrate that it respects the dependencies. This ordering works for the CSS problem too, but their proof of this fact no longer applies because the CSS table entries have a different dependency structure. We repeat the ordering here and prove that under it, a node is never expanded until all its parents have been processed.

Definition 10. *Let* $S = (m, l_1, \ldots, l_{\gamma(k)})$ *and* $S' = (m', l'_1, \ldots, l'_{\gamma(k)})$. *We say that* $S \ll S'$ *if and only if*

$$(m + l_1 + \cdots + l_{\gamma(k)}, m + l_1 + \cdots + l_{\gamma(k)-1}, \ldots, m + l_1, m)$$

is lexicographically smaller than

$$(m' + l'_1 + \cdots + l'_{\gamma(k)}, m' + l'_1 + \cdots + l'_{\gamma(k)-1}, \ldots, m' + l'_1, m')$$

As an example, compare the vector for the level-3-signature of Fig. 3.4 (a) (5,5,2,0) with the level-4-signature of Fig. 3.4 (a) (6,6,4,1). Since $5 < 6$ we would expand the level-3-signature before the level-4-signature. If the first positions of the vectors had the same magnitude we would compare the the second positions and so on. This ordering guarantees that we will never expand a signature until all its parents are expanded. As noted before, we cannot apply Golin and Rote's result because of the differences in dependency structure.

INITIALIZATION

1 MIN$[m, l_1, \ldots, l_{\gamma(k)}] \leftarrow \infty$ for all $m + l_1 + \cdots + l_{\gamma(k)} \leq n$
2 **for** $j \leftarrow 2$ **to** $\gamma(k)$
3 **do** MIN$[m, l, 0, \ldots, 0] \leftarrow 0$ for $l = 2$ to $\gamma(k)$

BODY

1 **Foreach** $(m, l_1, \ldots, l_{\gamma(k)})$ in lexicographic order from $(0, 2, 0, \ldots, 0)$ to $(n, 0, \ldots, 0)$
2 **do** *cost* $\leftarrow$ MIN$[m, l_1, \ldots, l_{\gamma(k)}] + \sum_{s=m+1}^{n} w_s$
3 **Foreach** expansion vector $(a_0, \ldots, a_k)$ **where**
$a_0 \leq l_1$, $a_1 = 0$, and $\sum_{t=2}^{k} a_t = a_0$
4 **do** $b = (b_0, \ldots, b_{\gamma(k)})$ **where**
$b_0 = -a_0$ and $b_{\gamma(i)} = i \cdot a_i$ for $2 \leq i \leq k$
5 $(m', l'_1, \ldots, l'_{\gamma(k)}) \leftarrow (m + l_1, l_2, \ldots, l_{\gamma(k)}, 0) + b$
6 **if** $m' + l'_1 + \cdots + l'_{\gamma(k)} \leq n$
7 **then** MIN$[m', l'_1, \ldots, l'_{\gamma(k)}] \leftarrow \min(MIN[m', l'_1, \ldots, l'_{\gamma(k)}], \mathit{cost})$
8 STRUCT$[m', l'_1, \ldots, l'_{\gamma(k)}] \leftarrow (a_0, \ldots, a_k)$
9 MIN$[n, 0, \ldots, 0]$ is the cost of the optimal subtree
10 STRUCT$[n, 0, \ldots, 0]$ tells us how to create the optimal subtree

Figure 3.5. The algorithm for finding minimal cost subtrees with k-favorable degree costs from constraint-free graphs with arbitrary leaf weights

Lemma 5. *Let* $S = (m, l_1, \ldots, l_{\gamma(k)})$ *and* $S' = (m', l'_1, \ldots, l'_{\gamma(k)})$. *If* $a = (a_0, \ldots, a_k)$ *takes* S *to* S' *then* $S \ll S'$.

Proof. Let $S = (m, l_1, \ldots, l_{\gamma(k)})$ and $S' = (m', l'_1, \ldots, l'_{\gamma(k)})$ such that S yields S' with level-i-child vector $a = (a_0, \ldots, a_k)$. If $a_0 > 0$ then $m' + l'_1 + \cdots + l'_{\gamma(k)} > m + l_1 + \cdots + l_{\gamma(k)}$ since a_0 of the l_1 nodes are replaced by at least $2 \cdot a_0$ nodes so $S \ll S'$. If $a_0 = 0$ then each term s'_i in S' is greater than or equal to its corresponding term s_i in S with at least one of the final terms in S' exclusively greater since at least one of the l_i terms is non-zero. Again, in this case we have $S \ll S'$. □

Since we have a consistent ordering for filling in the table entries, as well as a method for knowing the dependencies among the entries, we can build trees, level by

level, using the level-i-signatures. A description of the algorithm is given in Figure 3.5. Note that all the table entries are initially set to ∞ save the entries of all the depth-one trees, which are set to 0.

As previously shown, checking the dependencies for a table entry takes time $O(n^{k-1})$ and there are $O(n^{\gamma(k)+1})$ entries to check, so the total time of the algorithm is $O(n^{\gamma(k)+k})$. Finally, note that we store the child vectors which lead to the minimum cost for each level in the STRUCT table entries. We can recreate the tree corresponding to the optimal cost by using the child vectors to backtrack through the table entries.

3.5 Approximations

Many hierarchies have the property that no category has more than a constant number of subcategories. This means the out-degree of every node in the constraint graph is bounded above by a constant. In this section we give two theorems dealing with such cases. The first theorem says that even if we restrict the problem to DAGs of constant maximum degree, CSS remains **NP**-Hard for certain degree costs. The second theorem gives an $O(\log(k)\gamma(d+1))$ approximation algorithm for all instances of CSS where the maximum degree of the constraint graph is bounded above by some constant d, and γ is k-favorable and has a lower bound of 1.

Let a cost function be *k-tree optimal* if, for all instances of CSS with constraint-free graphs and equal leaf weights, the unique optimal solution tree with k^c leaves, for any positive integer c, is a complete k-ary tree of depth c. For example, in subsection 3.7 we show that the linear degree function is 3-tree optimal.

Theorem 4. *For any cost function that is k-tree optimal, for any $k \geq 3$, the CSS problem is* **NP***-Hard even when restricted to the uniform probability distribution and DAGs with degree at most $k+5$.*

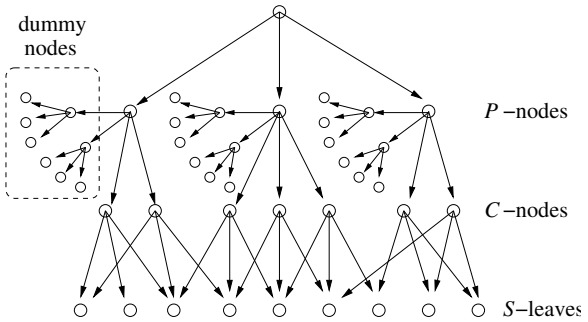

Figure 3.6. An example DAG construction for the case where $k = 3$ and q is a power of 3.

Proof. Consider the *Partitioned Exact Cover by 3 Sets* problem, which we define here, and abbreviate by PX3S. The input is a set S of $3q$ elements, where q is an integer, a collection C of subsets of S of size 3, and a partition P of the collection C into exactly q cells. We ask whether there is an exact cover of S that uses exactly one subset from each cell of P.

The proof is in two parts. We first show that the PX3S problem is reducible to the CSS problem with a k-tree optimal cost function, restricted to DAGs of degree at most $k+r-1$, where r is the maximum number of subsets in any cell of the partition P. We then show that the PX3S problem is **NP**-complete even when we restrict r to six.

We begin by constructing a CSS problem with the leaves of S augmented by a number of dummy leaves to be made precise below. The former are called S-leaves. We use the uniform probability distribution. The DAG has a single node for each cell of the partition P, called a P-node, as well as a single node for each of the subsets in C, called a C-node. Each P-node x points to the (at most r) C-nodes that belong to the cell for x. In addition, x points to $k-1$ dummy nodes only used by x, and each of these in turn point to k distinct dummy leaves. There are also an additional $k-3$ dummy nodes for x. Let y be a C node pointed to by x. y points to these $k-3$ dummy nodes, as well as to the 3 S-leaves for the subset represented by y.

If q is a power of k, then we are done merely by connecting to the P-nodes using a complete k-ary tree of depth $\log_3 q$. See Figure 3.6 for an example of this. If not, let t be the smallest power of k such that $t > q$. Add $t - q$ complete k-ary trees of height two to the DAG, and then connect to these nodes as well as the P-nodes using a complete k-ary tree of depth $\lceil \log_3 q \rceil$.

Claim 1. *This DAG has a complete k-ary tree as a subgraph iff the original PX3S input was a YES instance.*

To see that this claim is true, note that a YES instance of PX3S can be converted into a complete k-ary tree simply by having each P-node point to the single C-node that corresponds to the subset of S included in the solution to the PX3C input. Each P-node also points to all of its $k-1$ child dummy nodes, which in turn point to their dummy leaves.

For the other direction, note that for any possible complete k-ary tree, the distance from the root to any S-leaf must be exactly $\lceil \log_k q \rceil$. This implies that each S leaf must have a parent that is a C-node. Thus, to have a complete k-ary tree, each P-node must have exactly one C-node as a child. The set of such C-nodes defines a solution to the PX3S problem.

From this claim, one can see that PX3S reduces to CSS, since the complete k-ary tree is the unique optimal solution to the unconstrained version of the problem, and thus cannot be matched by any other solution.

We next show that PX3S is **NP**-complete by a reduction from Not-All-Equal-3-SAT (denoted here by NE3SAT). In order to do so, we start by taking the NE3SAT input ϕ, and converting it to the formula ϕ', where each clause in ϕ appears in ϕ' twice: once as it is in ϕ, and once with each literal negated from its value in ϕ. It is easy to see that ϕ' is valid iff ϕ is valid. Furthermore, we can take advantage of the facts that (a) each variable in ϕ' appears as many times negated as unnegated,

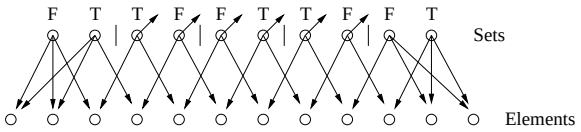

Figure 3.7. An example variable structure. The sets (in the top row) point to the elements (the bottom row) that they contain. This does not include the elements for each clause. Note also that the partition of the sets is depicted.

and (b) each such pair of clauses in ϕ' must have exactly 3 true literals in any valid assignment to the variables.

We next describe a variable structure used for each variable x. Let $n(x)$ be the number of appearances of x in the formula (exactly half of which will be negated). There will be $n(x)+4$ 3-sets for x, as well as $n(x)+6$ elements of S for x. Denote the 3-sets as $c_1, \ldots, c_\ell$, $\ell = n(x)+4$, and the elements as $s_1, \ldots, s_m$, $m = n(x)+6$. The set c_1 contains s_1, s_2 and s_3. The set c_2 contains s_1, s_2 and s_4. The set $c_{\ell-1}$ contains s_{m-3}, s_{m-1} and s_m. The set c_ℓ contains s_{m-2}, s_{m-1} and s_m. For $3 \leq i \leq \ell - 2$, c_i contains s_i and s_{i+2}, as well as a third element to be described later. An example of this is depicted in Figure 3.7.

It is now not difficult to see that in any possible exact three cover, it must be the case that the cover includes either exactly those sets c_j such that $j = 1 \mod 4$ or $j = 0 \mod 4$, or those sets c_j such that $j = 2 \mod 4$ or $j = 3 \mod 4$. The first of these will represent that variable being set to FALSE, and the second will represent that variable being set to TRUE. We can partition the sets $c_1, \ldots, c_\ell$ into consecutive pairs; each of these pairs forms one cell of the final partition P.

For each pair of clauses in ϕ' (that corresponded to a single clause in ϕ), we have a set of six elements: one for each literal. Each of these elements is used as the remaining element of a set c_i, for $3 \leq i \leq \ell - 2$. In particular, if the literal is $\overline{x}$, then we use a set c_i for the variable x such that $i = 1 \pmod 4$ or $i = 0 \pmod 4$. If the literal is x, then we use a set c_i for the variable x such that $i = 2 \pmod 4$ or $i = 3 \pmod 4$.

Finally, for each pair of clauses we have a set of six sets that form a single cell of the partition C. Since the literals in the first clause in the pair are negated versions of the literals in the second clause, there are exactly six ways to pick three literals set to false in a valid assignment of variables in this pair of clauses. Each of these last six sets contains the three elements corresponding to the false literals for one of these six settings.

It is now not difficult to see that there is an exact three cover for the constructed PC3S problem if and only if there is a valid (not all equal) assignment to the variables of ϕ. Furthermore, each cell of the partition P has at most 6 subsets. □

Theorem 5. *For any constraint graph G with m nodes where every node has out-degree at most d and for every k-favorable degree cost γ where γ is bounded below by 1, CSS with G and γ has an $O(m^2)$ time $O(\log(k)\gamma(d+1))$-approximation to the optimal solution.*

Proof. We begin by giving a lower bound on any instance of CSS where the degree cost is k-favorable and bounded below by 1. Take W as the probability distribution over leaf weights, $W(x)$ as the total weight of the leaves in the subtree rooted at x and H as the entropy function.

Lemma 6. *For any k-favorable degree cost γ with γ bounded below by 1, $\frac{H(W)}{\log(k)}$ is a lower bound on the cost of an optimal solution to CSS with γ.*

Proof. Let $I = (G, \gamma, (w_i))$ be an instance of CSS where γ is k-favorable, and $\gamma(x) \geq 1$ for all $x \geq 1$. Let T be an optimal tree for I. Now consider the tree T' corresponding to the optimal prefix-free code for n words weighted by (w_i) using a k-character alphabet where the cost of each character is 1. By Shannon's theorem, the expected path length of T' is bounded below by $\frac{H(W)}{\log(k)}$. If c is the cost function under γ and c' is the cost function for $\gamma(x) = 1$ then $c'(T)$ is a lower bound on $c(T)$. But $c'(T')$ must be a lower bound on $c'(T)$, so $\frac{H(W)}{\log(k)}$ is a lower bound on $c(T)$. □

The approximation algorithm also requires the following result.

Lemma 7. *For any tree with with weights on its m nodes, there exists one node, which, when removed, divides the tree into subtrees where every subtree has at most half the weight of original tree. Furthermore we can find this node in $O(m)$ time.*

Proof. Let T be a tree with weights on its m nodes. Let $\mathcal{W}$ be the sum of all the weights. If we do a post-order traversal of the tree, assigning interior nodes an additional new weight which is the sum of the additional weights of its children plus its own regular weight, then the first node we encounter with new weight exceeding half of $\mathcal{W}$ should be removed.

Claim 2. *The node we remove divides the tree into subtrees where each subtree has at most half the weight of the original tree.*

It's clear that the children of the removed node are all roots of new subtrees, each with weight less than half of $\mathcal{W}$ since their parent is the first node in the pre-order traversal having exceeding half the total weight. Likewise, the root of the original tree forms a new subtree with weight less than half of $\mathcal{W}$ because the total weight of the subtree rooted at the removed node exceeds half of $\mathcal{W}$. Calculating $\mathcal{W}$ and performing the pre-order traversal takes time $O(m)$. □

Let $I = (G, \gamma, (w_i))$ be an instance of CSS where where every node in G has out-degree at most d and γ is k-favorable. Extract any spanning tree T from G. Using Lemma 7 we can identity a node in T called the *splitter* which, when removed, divides T into subtrees where each subtree has at most half the probability mass of T. In this algorithm, we don't remove the splitter from the tree but rather, remove the edge(s) connecting it to its parent(s). We reconnect the splitter to the root of T. Recursively apply this procedure on the subtrees rooted by the children of the root of T and call the final tree T'. Note that T' is still a subtree of the transitive closure of G since the splitter node is always a descendent of the root of the tree under consideration.

If G has m nodes than extracting a spanning tree from G takes $O(m)$ time. The complete procedure still takes $O(m)$ time since a post-order traversal means we can keep enough information so that a node is examined only a constant number of times.

Claim 3. *If r and s are nodes in T' where r is the grandparent of s, then $W(r) \geq 2 \cdot W(s)$.*

This claim follows immediately from the construction of T' with respect to Lemma 7. Since any two hops in T' divides the probability mass of the subtree in half, we know the depth of leaf i is bounded above by $-2\log_2(w_i)$. Since each node in T' has degree at most $d+1$, the cost of T' is at most

$$2 \cdot \gamma(d+1) \sum_{i=1}^{n} w_i(-\log_2(w_i)) = 2 \cdot \gamma(d+1)H(W)$$

Since $O(\gamma(d+1)H(W))$ approximates the lower bound of $\frac{H(W)}{\log(k)}$ by a multiplicative factor of $O(\log(k)\gamma(d+1))$, we have an $O(m)$-time, $O(\log(k)\gamma(d+1))$ approximation algorithm for all instances of CSS where G has maximum degree d and γ is k-favorable.

□

3.6 Leaves of Equal Weights

It is easy to imagine nascent companies building websites without any prior popularity statistics on their products. To gather such statistics, they may want a website that puts all their products on an equal footing. Finding the optimal website for equally-weighted topics corresponds to instances of CSS with a uniform probability distribution over the leaves. We characterize optimal trees for these instances of CSS for the linear degree cost when the graph is constraint-free, and for the logarithmic degree cost for any DAG.

3.7 Linear Degree Cost

Characterizing the optimal tree for the linear degree cost in constraint-free graphs involves three parts. First, we push all the out-degree-two nodes in an optimal tree down toward the leaves so that all but a few interior nodes have out-degree-three. Next, we show that balancing the tree decreases its cost, and finally, we show an optimal arrangement of the leaves for the balanced tree. All the results in this section assume a constraint-free graph, equal leaf weights, and the linear degree function, $\gamma(x) = x$. Additionally, we use the term *degree* in lieu of *out-degree* since we are only concerned with the out-degree of any node. Since γ is 3-favorable, we always assume that optimal trees have only out-degree-two and out-degree-three nodes.

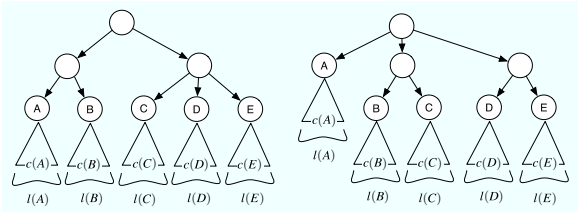

Figure 3.8. (left) A degree 2 node with one degree-two child and one degree-three child. (right) The node after the transformation.

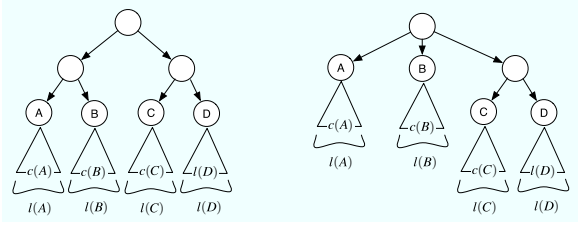

Figure 3.9. (left) A degree-two node with two degree-two children. (right) The node after the transformation.

Theorem 6. *If $(G, \gamma, (w_i))$ is an instance of CSS where G is constraint-free, $\gamma(x) = x$, and the n leaf weights are equal, then if $n \leq 2 \cdot 3^k$, where $k = \lfloor \log_3(n) \rfloor$, an optimal tree has cost*

$$3nk + 4(n - 3^k)$$

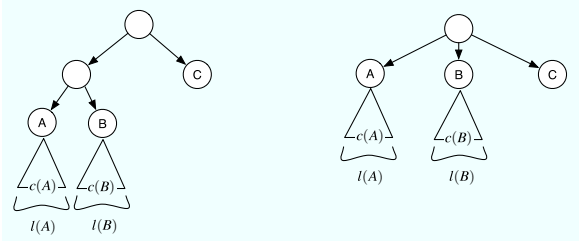

Figure 3.10. (left) A degree-two node with one degree-two child and one leaf child. (right) The node after the transformation.

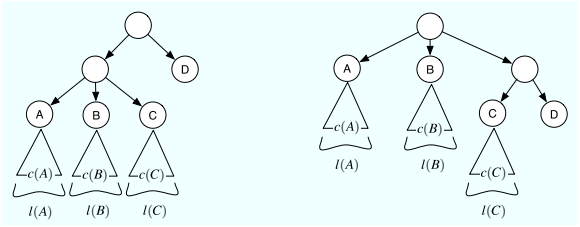

Figure 3.11. (left) A degree-two node with one degree-three child and one leaf child. (right) The node after the transformation.

otherwise it has cost

$$3(k+1)(3^{k+1}) - ((3^{k+1} - n)(3(k+1) + 2))$$

Proof. We begin by showing the existence of an optimal tree with the following property:

Property 1. *Every node has out-degree 2 or out-degree 3. Additionally, the only nodes with out-degree 2 are parents of leaves.*

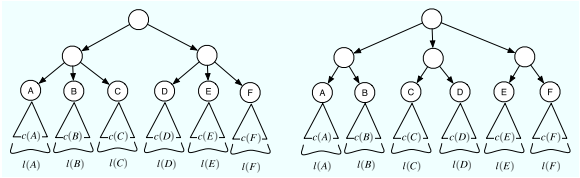

Figure 3.12. (left) A degree-two node with two degree-three children. (right) The node after the transformation.

Lemma 8. *If $(G, \gamma, (w_i))$ is an instance of CSS where G is constraint-free, the leaf weights are distributed uniformly, and $\gamma(x) = x$, then there is an optimal tree with Property 1.*

Proof. The proof is by cases. We know an optimal tree exists with only degree-two and degree-three nodes since $\gamma(x)$ is three-favorable. Suppose this optimal tree has an interior degree-two node which is the grandparent or some earlier ancestor of a leaf node. There are five possible combinations of node degrees for the children of this interior node. We'll show that in each case, one can push the degree two nodes down in the tree without increasing the overall cost. In this analysis, we'll let $l(a)$ be the number of leaves in subtree rooted at node a (which corresponds to the number of paths through node a) and we'll let $c(a)$ be the total non-weighted cost of the subtree rooted at a.

- Consider a degree-two node with one degree-two child and one degree-three child like the one given on the left in Figure 3.8. Let $l(A) \geq l(B)$ and $T = c(A) + c(B) + c(C) + c(D) + c(E)$. The cost of the tree is $4(l(A) + l(B)) + 5(l(C) + l(D) + l(E)) + T$, but the tree on the right has total cost $3l(A) + 5(l(B) + l(C) + l(D) + l(E)) + T$ which is no greater than the cost of the tree on the left since $l(A) \geq l(B)$.

- Consider a degree-two node with two degree-two children like the one given on the left in Figure 3.9. Let $l(A) \geq l(B) \geq l(C) \geq l(D)$ and $T = c(A) + c(B) + c(C) + c(D)$. The cost of the tree is $4(l(A) + l(B) + l(C) + l(D)) + T$, but the tree on the right has cost $3(l(A) + l(B)) + 5(l(C)l(D)) + T$. Since $l(A) + l(B) \geq l(C) + l(D)$, we know the cost of the tree never increases when transforming the structure from the left figure to the right figure.

- Consider a degree-two node with one degree-three child and one leaf child like the one given on the left in Figure 3.11 where $l(A) \geq l(B) \geq l(C) \geq l(D) = 1$ and $T = c(A) + c(B) + c(C) + c(D)$. The cost of the left tree is

$$\begin{aligned} 5(l(A) + l(B) + l(C)) + 2l(D) + T &\geq 4l(A) + 5(l(B)) + l(C)) + 3l(D) + T \\ &\geq 4(l(A) + l(B) + l(D)) + 5(l(C) + T \\ &\geq 3l(A) + 4l(B) + 5(l(C) + l(D)) + T \\ &> 3(l(A) + l(B)) + 5(l(C) + l(D)) + T \end{aligned}$$

  which is the cost of the right tree. So transforming the left tree to the right tree always decreases the total cost.

- Consider a degree-two node with one degree-two child and one leaf child like the one given on the left in Figure 3.10 where $l(A) \geq l(B) \geq l(C) = 1$ and $T = c(A) + c(B) + c(C)$. The cost of the tree on the left is

$$\begin{aligned} 4(l(A) + l(B)) + 2l(C) + T &\geq 3(l(A) + l(C)) + 4l(B) + T \\ &> 3(l(A) + l(B) + l(C)) + T \end{aligned}$$

  which is the cost of the tree on the right. So transforming the left tree to the right tree always decreases the cost.

- Consider a degree-two node with two degree-three children like the one on the left in Figure 3.12. Transforming the left tree into the right tree keeps the cost of the tree invariant because of the symmetry between two degree-three nodes and three degree-two nodes. In each arrangement, the path cost is always five from the root to the grandchildren.

Since these cases are exhaustive and since we never increase the cost of the tree by pushing degree-two nodes down in the topology, we can always transform an optimal

tree into another optimal tree where every ancestor above the parent of a leaf has degree three. □

Next we show that making a tree more balanced only improves the overall cost of the tree. In addition we characterize the fringe of the tree.

Lemma 9. *If $(G, \gamma, (w_i))$ is an instance of CSS where G is constraint-free, $\gamma(x) = x$, and the leaf weights are distributed uniformly, then any optimal tree meeting Property 1 has the following characteristics:*

(a) No two leaves differ in depth by more than one.

(b) Degree-two internal nodes have the same depth.

(c) No internal node has degree greater than any other internal node of less depth.

(d) No degree-three internal nodes have the same depth of a leaf.

Proof. We begin by proving (a). Let T be an optimal tree meeting Property 1. Such a tree exists by Lemma 8. Let a and b be leaf nodes and a' and b' be their respective parents. Suppose that $depth(a) > depth(b) + 1$. Then if the path cost to b' is $3k$ for some non-negative integer k, the path cost to a' must be at least $3(k + 2)$. We can always move a to b' and decrease the cost of the tree: If b' is a degree-two node, we can add a leaf and increase the overall cost by $3k + 5$. If b' is a degree-three node, we can make b an internal degree-two node and increase the overall cost by $3k + 7$. If a' is a degree-two node, removing a saves us $3k + 10$ and if a' is a degree-three-node, removing a saves $3k + 11$. In all the combinations of leaf removal from a' and leaf addition to b' decrease the cost of the tree, a contradiction of the optimality of T, so all leaves of T differ in depth by at most one.

Since all degree-two nodes are only parents of leaves, and all leaves differ in depth by at most one, to prove (b) and (c), we need to show that if b is a leaf at depth k with a degree-two parent b', then a' cannot be a degree-three or degree-two parent of

a $k+1$-depth leaf. Suppose this were true. Then we can add a leaf to b', increasing the cost by $3k+2$ and remove a leaf at a', decreasing the cost by at least $3k+4$, giving at least a net savings of two, a contradiction.

Since T has properties (a), (b), and (c), to prove (d), we need only show that if a is a leaf node at depth $k+1$ with a degree-three parent node a' at depth k, then no leaf b exists at depth k. Suppose this were true. Then we could make b an internal node (rename it b'') with two children a and b at a cost of $3k+4$, but removing a from a' saves $3k+5$, again, a contradiction of optimality. □

At this point we know the exact structure of an optimal tree T. Lemma 8 and property (a) of Lemma 9 tell us that the tree is balanced, and that any degree-two node must be the parent of only leaf nodes. Property (b) restricts degree-two nodes to parents of leaves at the lowest depth, so that if T has n leaves, it always begins with a complete ternary tree of height $\lfloor \log_3(n) \rfloor$. Finally, property (c) implies that a leaf has a degree-three parent, only when all the leaves of the tree have the same depth. In total, given n leaves, we build the largest complete ternary tree of size $k = \lfloor \log_3(n) \rfloor$, then add additional leaves by making leaves of the height-k ternary tree into degree-two parents of leaves, and finally adding an additional leaf to the binary nodes when no leaves at depth k remain. This means, the bottom row of the optimal tree gets filled with degree-two nodes first and then, when additional leaves are required, the degree-two nodes are expanded to degree-three nodes. Given this fixed structure, we can give an exact measure of its cost. When $n \leq 2 \cdot 3^k$ the bottom row is exclusively degree-two nodes, so the total cost of the optimal tree is $3nk + 4(n - 3^k)$ where the first component of the sum is the cost of the complete ternary tree, and the second component of the sum is the cost of the additional leaves. When the bottom row contains leaves coming from degree-three parents, we can think of the cost of the optimal tree in terms of subtracting away from the complete ternary tree of depth $k+1$. Making a degree three node a degree two node means removing one leaf which

reduces the overall cost by $3(k+1)$, but since the degree on the parent changes, each sibling path is reduced by one, making the overall savings $3(k+1)+2$. This means the total cost of the tree is $3(k+1)(3^{k+1})-((3^{k+1}-n)(3^{k+1}+2)$ where the first part of the sum is the cost of the complete ternary tree and the second part is the reduction in cost by removing the appropriate number of leaves. □

In Section 3.5 we defined a degree cost γ as k-tree optimal when, for constraint-free graphs and equal leaf weights, the unique optimal tree with k^c leaves, for any positive integer c, is a complete k-ary tree of depth c. Here we show that the linear degree cost is 3-tree-optimal.

Theorem 7. *The degree cost $\gamma(x) = x$ is 3-tree-optimal.*

Proof. Let T be an optimal tree for the CSS problem where the graph is constraint-free, the degree cost is $\gamma(x) = x$, and there are $n = 3^k$ leaves, for some positive integer k, of equal weight. T may have nodes with degree two, three, or four. Nodes with degree five or larger don't appear in an optimal tree since replacing them with degree-two and degree-three nodes always decreases the cost of the tree. We replace all degree-four nodes from T with complete binary trees of size three at no additional cost. By Lemma 8 we can push the degree-two nodes down to the fringe. Furthermore, since T is optimal, we won't match Figures 3.11 and 3.10 because they are strictly cost-decreasing transformations. Since the other transformations always preserve at least one of the degree-two nodes, any optimal tree with degree-two nodes has a corresponding tree of equal cost with at least one degree-two node and all degree-two nodes are parents of only leaves (*i.e.*, it exhibits Property 1). By Lemma 9, we know T has at least $3^{k-1}+1$ leaves, but no more than 3^k-1 leaves because there is at least one degree-two parent of only leaves. But T has $n = 3^k$ leaves, a contradiction, so T cannot have degree-two or degree-four nodes. This means the optimal tree has

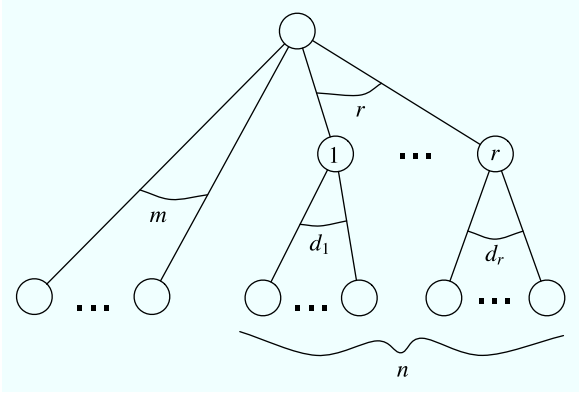

Figure 3.13. The topology of an mrn-tree.

only degree-three nodes, and by Theorem 6, is a complete ternary tree with $n = 3^k$ leaves. □

3.8 Logarithmic Degree Costs

Another natural choice of degree cost is $\gamma(x) = \lg(x)$ (where $\lg = \log_2$) because it gives the number of bits needed to encode the out-degree of the node. In this section we show the depth-one tree (where the root has n edges of its n leaves) is an optimal solution to any instance of CSS where the n leaf weights are equal and $\gamma(x) = \lg(x)$. This result holds for arbitrary graphs because the depth-one tree is always a subtree of the transitive closure.

Theorem 8. *Let $I = (G, \gamma, (w_i))$ be an instance of CSS where $\gamma(x) = \log(x)$ and the n leaf weights are equal. An optimal tree for I is the depth-one tree.*

Proof. For clarity, we distinguish between the leaves at depth one and the leaves at depth two. Let $T = (m, (d_1, \ldots, d_r)), m, d_i \in \mathbb{Q}^+$ be a depth-two tree with $m + n$ leaves: m depth-one leaves and $n = \sum_{i=1}^{r} d_i$ depth-two leaves. Each depth-two leaf has a single path from the root to it through one of the r intermediate nodes. Denote the degree of each intermediate node i as d_i. We call these trees mrn-trees and illustrate the general topology in Figure 3.13. We begin by proving an intermediate result on the nature of convex functions.

Lemma 10. *Let I be any real interval and let $(x-\alpha, x+c+\alpha)$ be a subinterval of I where c and α are real values with $c \geq 0$ and $\alpha > 0$. If $g : I \to \mathbb{R}$ is convex on I (and has a second derivative in I) we have*

$$g(x+c+\alpha) + g(x-\alpha) > g(x+c) + g(x)$$

Proof. Suppose $g(x)$ is defined over some real interval I and $(x-\alpha, x+c+\alpha)$ is a subinterval of I with $c \geq 0$ and $\alpha > 0$. Suppose further that $g(x)$ is convex on I (and has a second derivative on I), then $g'(x)$ is increasing so

$$\begin{aligned}
g'(x+c+y) - g'(x-\alpha+y) &> 0 \qquad c > 0, \alpha \geq 0 \text{ and } 0 \leq y \leq \alpha \\
\Leftrightarrow \int_0^\alpha g'(x+c+y) - g'(x-\alpha+y)dy &> 0 \\
\Leftrightarrow \int_0^\alpha g'(x+c+y)dy &> \int_0^\alpha g'(x-\alpha+y)dy \\
\Leftrightarrow g(x+c+\alpha) - g(x+c) &> g(x) - g(x-\alpha) \\
\Leftrightarrow g(x+c+\alpha) + g(x-\alpha) &> g(x+c) + g(x)
\end{aligned}$$

□

Lemma 10 means that making the degrees of the intermediate nodes of an *mrn*-tree proportional always improves its cost.

Lemma 11. *If $\gamma(x)$ is an increasing, differentiable function over the positive reals, then the r intermediate nodes of the minimum cost mrn-tree have out-degree n/r when the weights of the leaves are equal.*

Proof. Let $\gamma(x)$ be increasing and differentiable over the positive reals. Then $g(x) = x\gamma(x)$ has a second derivative on the positive reals, so it is convex. Let $T = (m, (d_1, \ldots, d_r))$ be an *mrn*-tree with $m+n$ equally weighted leaves. For the sake of

contradiction, suppose T has at least two intermediate nodes, i and j, with disproportionate out-degree, so that $d_i > d_j > 0$. Let $d_i = d_j + \beta$ for some positive $\beta \in \mathbb{Q}$. Shifting some rational part of the path, $0 < \alpha \leq \beta/2$ from node i to node j to make i and j more proportional, decreases the cost of the subtree rooted at i by

$$d_i\gamma(d_i) - (d_i - \alpha)\gamma(d_i - \alpha) \tag{3.2}$$

but the cost of the subtree rooted at j increases by

$$(d_j + \alpha)\gamma(g_j + \alpha) - d_j\gamma(d_j) \tag{3.3}$$

Letting $g(x) = x\gamma(x)$, we need to show that the decrease in cost is greater than the increase in cost:

$$g(d_i) - g(d_i - \alpha) > g(d_j + \alpha) - g(d_j) \tag{3.4}$$

Since $\alpha \leq \beta/2$ we have $\beta = 2\alpha + c$ for some $c \geq 0$ so $d_i = d_j + 2\alpha + c$. Letting $x = d_j + \alpha$ we have $d_i = x + \alpha + c$ and by rearranging the terms from the inequality in 3.4 we have

$$g(x + c + \alpha) + g(x - \alpha) > g(x + c) + g(x)$$

which holds from Lemma 10 and gives a contradiction, so the minimum cost mrn-tree with equal leaf weights must have proportional degrees among its r interior nodes. In other words, each interior node has out-degree n/r. □

From the proportional mrn-tree, we can move to the depth-one tree with $m + n$ leaves without increasing the overall cost:

Lemma 12. *If $\gamma(x) = \log(x)$ then any optimal mrn-tree with equal leaf weights has an equivalent cost depth-one tree with equal leaf weights.*

Proof. Let $\gamma(x) = \log(x)$ and $T = (m, (d_1, \ldots, d_r))$ be an optimal mrn-tree with equal leaf weights. By Lemma 11 we can transform T, at no additional cost, so that each intermediate node has out-degree n/r. This makes the cost of T:

$$(n+m)\log(r+m) + n\log(n/r)$$

whereas the cost of the equivalent depth-one tree with $(n+m)$ leaves is

$$(n+m)\log(n+m)$$

We'd like to show that the cost of the depth-one tree is no greater than the cost of the mrn-tree. Subtracting the cost of the depth-one tree from the cost of the mrn-tree gives us:

$$f(n,m,r) = (n+m)\log(r+m) + n\log(n/r) - (n+m)\log(n+m) \tag{3.5}$$

where n, m, r are non-negative rational values and $r \leq n$. Showing f never takes on negative values gives the desired result, so first we show it is non-decreasing in m and then show that $f(n,0,r) \geq 0$ for all appropriate values of n and r. Taking the partial derivative of f with respect to m give us:

$$\frac{\partial f}{\partial m} = \log(r+m) + \frac{n-r}{r+m} - \log(n+m) \tag{3.6}$$

and taking the partial derivative of 3.6 with respect to n gives us:

$$\frac{\partial^2 f}{\partial m \partial n} = \frac{1}{r+m} - \frac{1}{n+m} \tag{3.7}$$

which is greater than or equal to 0 since $r \leq n$. It immediately follows that 3.7 is non-decreasing, so we know 3.6 is non-decreasing in n. Since n is bounded below by

r, we let $n = r$ whence 3.6 equals 0. This means 3.5 is non-decreasing in m, so we let $m = 0$ in f:

$$f(n, 0, r) = n \log(r) + n \log(n/r) - n \log(n) = 0$$

Since f is non-decreasing in m and $f(n, 0, r) \geq 0$ for all appropriate n and r, the cost of the depth-one tree is no greater than the cost of the mrn-tree. □

We are now in a position to prove the result. Let $I = (G, \gamma, (w_i))$ be an instance of CSS where $\gamma(x) = \log(x)$ and G has n leaves of equal weight. Let T be an optimal tree for I containing the minimum number of nodes. Suppose T is not a depth-one tree. Then there exists a node having both (and only) children and grandchildren. This is an mrn-tree, and by Lemma 12, we can covert it into a depth-one tree at no additional cost, transforming T into a new optimal tree with less nodes, a contradiction, so T must be a depth-one tree. □

We can generalize Lemma 12 above to show that all functions which always increase at a rate slower than log take the depth-one tree as optimal:

Corollary 1. *Let γ be a differentiable degree cost such that for all positive integers x*

$$\frac{d}{dx}\gamma(x) \leq \frac{d}{dx}\log(x)$$

then the depth-one tree is optimal for γ and equal leaf weights and constraint-free graphs.

Finally, recall that CSS is **NP**-hard under $\gamma(x) = \lceil \log(x) \rceil$ even when considering the problem with equal leaf weights. We conclude with an interesting approximation result related to these instances:

Corollary 2. *If $(G, \gamma, (w_i))$ is an instance of CSS with $\gamma(x) = \lceil \log_2(x) \rceil$ and n leaf weights are equal, then the depth-one tree approximates the optimal cost tree within an additive constant of 1.*

Proof. Let $I = (G, \gamma, (w_i))$ be an instance of CSS with degree cost $\gamma(x) = \lceil \log_2(x) \rceil$ and n uniformly distributed leaf weights. Furthermore, let c be the cost of a tree under $\gamma(x) = \lceil \log_2(x) \rceil$ and c' be the cost of a tree under $\gamma(x) = \log_2(x)$. If T is an optimal solution for I and T' is a depth-one tree with n leaves, we have the following inequality:

$$\begin{aligned} \log_2(n) &= c'(T') \\ &\leq c'(T) \\ &\leq c(T) \\ &\leq c(T') \\ &= \lceil \log_2(n) \rceil \\ &\leq \log_2(n) + 1 \end{aligned}$$

which gives the desired lower and upper bounds. □

3.9 Future Directions: Dynamic CSS

In many environments, the popularity of information changes. This implies a dynamic version of CSS where the leaf weights change, but some property of the tree (like optimality) is preserved. If we restrict our attention to integer leaf weights (*i.e.* frequency counts) and updates which only increase a leaf's frequency by a single count, then perhaps the most natural question asks, given an optimal tree how much time does it take to maintain optimality after a frequency count update? Faller [17], Gallager [22], and Knuth [32] study this problem for Huffman codes with equal letter costs. Knuth shows that when the weights are integers and the weights increase by only a single unit, that the optimality of the code can be maintained in $O(h)$ time where h is the height of the tree (*i.e.*, the length of the longest code word). The motivation behind these dynamic Huffman trees is that they provide a

one-pass method for document encoding: both parties maintain the same optimal tree for the first t characters, they encode and decode the $(t+1)^{th}$ character using this tree, and finally, they independently update their trees to maintain optimality. Compare this with the traditional, static two-pass method where the document is first combed for frequency counts and then encoded using the code built from the counts. Vitter [46, 47] analyzes Knuth's one-pass algorithm and shows that the number of bits encoded with the dynamic version is (essentially) at most double compared with the static version. Vitter uses this analysis to construct a new algorithm which encodes the document in about the same number of bits as the static method. This algorithm uses a new data structure called a floating tree. To the best of our knowledge, this result has not been extended to Huffman codes with unequal letter costs. This isn't surprising since all the algorithms rely on the sibling property which states that a binary tree is a Huffman code if and only if

- Each leaf has a positive weight and each internal node has a weight equal to the sum of its children's weights; and
- The m nodes may be numbered from 1 to m in non-decreasing order by weight such that siblings have consecutive numbers and their parent has a higher number

This numbering corresponds to the merging of nodes in Huffman's greedy algorithm. Since the greedy algorithm doesn't work for unequal letter costs (because the edges have independent costs) the sibling property does not hold. Developing an algorithm for the unequal letter case seems like a good first step in developing a dynamic version of CSS in constraint-free graphs with integer weights (, rational probability distributions) with the goal of maintaining tree optimality. It's worth noting that every CSS tree obeys a general version of the sibling property, because all siblings

have identical path cost from the root, but clearly every CSS tree is not optimal so stronger characterizations of optimality are required.

We have considered an alternative formulation of the dynamic version of CSS where an approximation ratio is maintained. For example, is it possible to preserve the $O(\log(k)\gamma(d+1))$ ratio given by our approximation algorithm in Section 3.5 when the weights are integers and the updates come as unit increases? If we consider this problem for constraint-free graphs, then we can generalize Vitter's algorithm to alphabets of size k and the result is a $\gamma(k)$-approximation. This follows because $H(D)/\log(k)$ (where $H(D)$ is the entropy of the probability distribution on the leaves) is a lower bound on the cost of optimal trees (cf. Section 3.5) in constraint-free graphs with k-favorable degree costs which are bounded below by 1. Since the tree produced by Vitter's algorithm achieves this bound under $\gamma(n) = 1$, all other k-favorable degree costs are within a multiplicative factor of $\gamma(k)$. Adding graphical constraints, though, makes the problem harder since an optimal tree must be a subtree of the transitive closure of the constraint graph.

CHAPTER 4

THE CATEGORY TREE PROBLEM

4.1 Introduction

Consider the problem of correctly diagnosing a disease in an ailing patient through a series of tests. Running all available tests is one option but it is probably impractical: Some disease diagnoses might require significantly more tests or may be more common than others; some tests might be more expensive or more effective at distinguishing different symptoms than others. One really wants to adaptively arrange the tests based on the outcomes of previous tests so as to minimize the expected number of tests necessary to diagnose the disease (where the expectation is taken over the rarity of the disease). This disease diagnoses problem is an example of the well-studied Decision Tree (DT) problem. In this chapter we develop a problem similar in spirit to CSS called Category Tree (CT) that generalizes DT to domains where the cost of running a test is a function of the number of outcomes. That is, it incorporates the degree cost we developed for CSS.

Many of our CT results apply to DT. For example, our approximation algorithm for CT gives the first non-trivial upper bound on the approximation ratio for DT. This result is especially poignant given that DT is **NP**-complete and the primary motivation for this 30-year old hardness result was because "of the large amount of effort that [had] been put into finding efficient algorithms for constructing optimal binary decision trees" [30]. In addition, we give the first non-trivial lower bounds on the approximation ratio of DT. We close with a new lower bound for a related decision tree problem called MinDT.

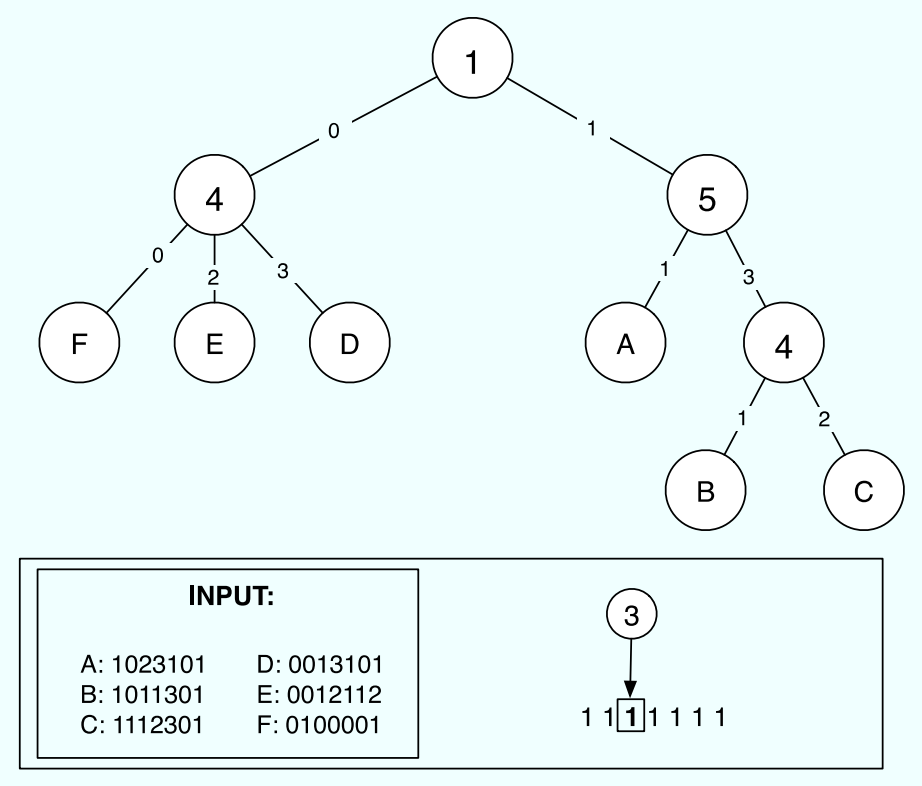

Figure 4.1. A valid Category Tree for the given 6 input strings. This input has 7 categories and each category has at most 4 values. The number inside each interior tree node indicates category (bit position) and each edge label indicates an appropriate categorical value. Assuming equal leaf weights of 1, this tree has cost 14 for the constant degree cost $\gamma(x) = 1$ and cost 31 for the linear degree cost $\gamma(x) = x$.

4.2 Definitions

An instance of Category Tree is a triple $\mathrm{CT}(\gamma, X, (w_i))$ where γ is the degree cost, X is a set of n items where each item is a string of length m over a finite alphabet of size k and $(w_i) = (w_1, \ldots, w_n)$ is a set of n positive item weights. When it is convenient, we'll view the weights as a probability distribution over the items.

We call bit positions categories because each position often represents a conceptual category or attribute. Hence, the integer values for a bit position enumerate the corresponding categorical values. For example, consider the problem of searching for a song in a personal digital music repository like iTunes: each song has a title, a genre, an artist, an album, a date, a track number, etc. that uniquely identifies it among

a collection of other songs. It is common to amass thousands of songs. Because of the sheer volume of songs, searching exclusively by title or even artist is inefficient. A better method might begin by first selecting among genres, then winnowing the search down by date range, then artist, and so on until only a handful of songs remain. CT models this and other categorical search problems.

The solution to CT is a full tree with n leaves with where each interior node is a category with out-degree at most k, each leaf is an item, and each edge is an appropriate value (from 1 to k) for the parent category. A path from the root to a leaf yields a sequence of categories and values that completely disambiguates that item from all other items. A valid category tree disambiguates all the items from one another. For example the tree in Figure 4.1 is a valid category tree for the given six inputs. Note that categories may appear multiple times along different branches of the tree.

The cost of a category tree is identical to the cost of a CSS tree. That is, tree cost is just the sum of the individual path costs. A path cost is the sum of the degree costs of the internal nodes along the path. Recall that degree cost, γ, is a function of the out-degree of a node. For example, if $\gamma(x) = 1$ then the cost of the tree in Figure 4.1 is 14; if $\gamma(x) = x$ then the cost is 31. In the weighted version of the problem, we weight each path cost by the leaf's weight. For example, if leaf A in Figure 4.1 has weight 2 and all the other leaves have weight 1, then the cost of the tree under $\gamma(x) = 1$ is 16. The goal of CT is to find a tree that minimizes tree cost. As with CSS, we often refer to tree cost as deliberation cost.

4.2.1 Binary Strings, Equal Weights, and Non-Decreasing Degree Costs

As defined in Garey and Johnson [23], DT instances are CT instances where X are binary strings, $\gamma(x) = 1$, and the leaf weights are all 1. However, the problems remain

equivalent even when the degree cost restriction is relaxed to include all positive, non-decreasing functions over $\{1, 2\}$, and the leaf weights are equal.

To see this, observe that CT instances of this type have solution trees that are *full* — trees where every internal node has at least (and in this case exactly) two children. This is because any internal node with one child may be removed to produce a new solution with lower cost. Since every solution tree is full, the degree cost becomes a constant — $\gamma(2)$ — which we can factor out of the cost and safely ignore for the purposes of optimization. In addition, positive, equal leaf weights always form a uniform probability distribution over the n leaves so we can safely ignore them too.

For convenience, we refer to CT instances restricted to binary strings, equal leaf weights, and positive, non-decreasing degree costs as DT instances. We also call the DT cost *total external path length* in keeping with convention.

4.3 Approximating DT

In this section we give a $\ln n + 1$ approximation for DT — CT instances where categories are binary, the leaf weights are equal, and the degree cost is positive and non-decreasing. Given a set of binary m-bit strings S, choosing some position i always partitions the items into two sets S^0 and S^1 where S^0 contains those items with position $i = 0$ and S^1 contains those items with position $i = 1$. A greedy strategy for splitting a set S chooses the bit i which minimizes the difference between the size of S^0 and S^1. In other words, it chooses the bit which most evenly partitions the set. Using this strategy, consider the following greedy algorithm for constructing decision trees given a set of n items X:

A straightforward implementation of this algorithm runs in time $O(mn^2)$. While the algorithm does not always give an optimal solution, it does approximate it within a factor of $\ln n + 1$.

```
GREEDY-DT(X)
1  if X = ∅
2     then return NIL
3     else Let i be the bit which most evenly partitions X into X^0 and X^1
4          Let T be a tree node with left child left[T] and right child right[T]
5          left[T] ← GREEDY-DT(X^0)
6          right[T] ← GREEDY-DT(X^1)
7          return T
```

Figure 4.2. A greedy algorithm for constructing decision trees.

Theorem 9. *If X is an instance of DT with n items and optimal cost $\mathcal{C}^*$ then* GREEDY-DT(X) *yields a tree with cost at most* $(\ln n + 1)\mathcal{C}^*$

Proof. We begin with some notation. Let $\mathcal{T}$ be the tree constructed by GREEDY-DT on X with cost $\mathcal{C}$. An unordered pair of items $\{x, y\}$ (hereafter just *pair of items*) is *separated* at an internal node S if x follows one branch and y follows the other. Note that each pair of items is separated exactly once in any valid decision tree. Conversely, each internal node S defines a set $\rho(S)$ of pairs of items separated at S. That is

$$\rho(S) = \{\{x, y\} \mid \{x, y\} \text{ is separated at } S\}$$

For convenience we also use S to denote the set of items in the subtree rooted at S. Let S^+ and S^- be the two children of S such that $|S^+| \geq |S^-|$. Note that $|S| = |S^+| + |S^-|$. The number of sets to which an item belongs equals the length of its path from the root, so the cost of $\mathcal{T}$ may be expressed as the sum of the sizes of each S:

$$\mathcal{C} = \sum_{S \in \mathcal{T}} |S|$$

Our analysis uses an accounting scheme to spread the total cost of the greedy tree among all unordered pairs of items. Since each set S contributes its size to the

total cost of the tree, we spread its size uniformly among the $|S^+||S^-|$ pairs of items separated at S. Let c_{xy} be the pair cost assigned to each pair of items $\{x, y\}$ where

$$c_{xy} = \frac{1}{|S^+_{xy}|} + \frac{1}{|S^-_{xy}|}.$$

and S_{xy} separates x from y. We can now talk about the cost of a tree node S by the costs associated with the pairs of items separated at S. Summing the costs of these pairs is, by definition, exactly the size of S:

$$\sum_{\{x,y\}\in\rho(S)} c_{xy} = |S^+||S^-|\Big(\frac{1}{|S^+|} + \frac{1}{|S^-|}\Big) = |S|$$

Because two items are separated exactly once, $\mathcal{C}$ is exactly the sum of the all pair costs:

$$\mathcal{C} = \sum_{\{x,y\}} c_{xy}$$

Now consider the optimal tree $\mathcal{T}^*$ for X. If Z is an internal node of $\mathcal{T}^*$ then we also use Z to denote the set of items that are leaves of the subtree rooted at Z. Following our notational conventions, we let Z^+ and Z^- be the children of Z such that $|Z^+| \geq |Z^-|$ and $|Z| = |Z^+| + |Z^-|$. The cost of the optimal tree, $\mathcal{C}^*$, is

$$\mathcal{C}^* = \sum_{Z\in\mathcal{T}^*} |Z| \tag{4.1}$$

Since, every feasible tree separates each pair of items exactly once, we can rearrange the greedy pair costs according to the structure of the optimal tree:

$$\mathcal{C} = \sum_{Z\in\mathcal{T}^*} \sum_{\{x,y\}\in\rho(Z)} c_{xy} \tag{4.2}$$

If Z is a node in the optimal tree, then it defines $|Z^+||Z^-|$ pairs of items. Our goal is to show that the sum of the c_{xy} associated with the $|Z^+||Z^-|$ pairs of items

split at Z (but which are defined with respect to the greedy tree) total at most a factor of $H(|Z|)$ more than $|Z|$ where $H(d) = \sum_{i=1}^{d} 1/i$ is the d^{th} harmonic number. This is made precise in the following lemma:

Lemma 13. *For each internal node Z in the optimal tree:*

$$\sum_{\{x,y\}\in\rho(Z)} c_{xy} \leq |Z|H(|Z|)$$

where each c_{xy} is defined with respect to the greedy tree $\mathcal{T}$.

Proof. Consider any node Z in the optimal tree. For any unordered pair of items $\{x, y\}$ split at Z, imagine using the bit associated with the split at Z on the set S_{xy} separating x from y in the greedy tree. Call the resulting two sets $S_{xy}^{Z^+}$ and $S_{xy}^{Z^-}$ respectively. Since the greedy split at S_{xy} minimizes c_{xy}, we know

$$c_{xy} = \frac{1}{|S_{xy}^{+}|} + \frac{1}{|S_{xy}^{-}|} \leq \frac{1}{|S_{xy}^{Z^+}|} + \frac{1}{|S_{xy}^{Z^-}|} \leq \frac{1}{|S_{xy} \cap Z^{+}|} + \frac{1}{|S_{xy} \cap Z^{-}|}.$$

Hence

$$\sum_{\{x,y\}\in\rho(Z)} c_{xy} \leq \sum_{\{x,y\}\in\rho(Z)} \frac{1}{|S_{xy} \cap Z^{+}|} + \frac{1}{|S_{xy} \cap Z^{-}|} \tag{4.3}$$

One interpretation of the sum in (4.3) views each item x in Z^+ as contributing

$$\sum_{y\in Z^-} \frac{1}{|S_{xy} \cap Z^{-}|}$$

to the sum and each node y in Z^- as contributing

$$\sum_{x\in Z^+} \frac{1}{|S_{xy} \cap Z^{+}|}$$

to the sum. For clarity, we can view Z as a complete bipartite graph where Z^+ is one set of nodes and Z^- is the other. Letting $b_{xy} = 1/(|(S_{xy} \cap Z^{-}|)$ and $b_{yx} = 1/(|S_{xy} \cap Z^{+}|)$

we can think of every edge (x, y) where $x \in Z^+$ and $y \in Z^-$ as having two costs: one associated with x (b_{xy}) and the other associated with y (b_{yx}). Thus, the total cost of Z is at most the sum of all the b_{xy} and b_{yx} costs. We can bound the total cost by first bounding all the costs associated with a particular node. In particular, we claim:

Claim 4. *For any $x \in Z^+$ we have*

$$\sum_{y \in Z^-} b_{xy} = \sum_{y \in Z^-} \frac{1}{|S_{xy} \cap Z^-|} \leq H(|Z^-|)$$

Proof. If Z^- has m items then let $(y_1, \ldots, y_m)$ be an ordering of Z^- in reverse order from when the items are split from x in the greedy tree (with ties broken arbitrarily). This means item y_1 is the last item split from x, y_m is the first item split from x, and in general y_{m-t+1} is the t^{th} item split from x. When y_m is split from x there must be at least $|Z^-|$ items in S_{xy_m} — by our ordering the remaining items in Z^- must still be present — so $Z^- \subseteq S_{xy_m}$. Hence b_{xy_m}, the cost assigned to x on the edge (x, y_m), is at most $1/|(Z^-)|$ and in general, when y_t is separated from x there are at least t items remaining from Z^-, so the cost b_{xy_t} assigned to the edge (x, y_t) is at most $1/t$. This means, for any $x \in Z^+$

$$\sum_{y \in Z^-} b_{xy} \leq H(|Z^-|)$$

which proves the claim. □

We can use the same argument to prove the analogous claim for all the items in Z^-. With these inequalities in hand we have

$$\begin{aligned}
\sum_{\{x,y\} \in \rho(Z)} \frac{1}{|S_{xy} \cap Z^+|} + \frac{1}{|S_{xy} \cap Z^-|} &\leq |Z^+|H(|Z^-|) + |Z^-|H(|Z^+|) \\
&< |Z^+|H(|Z|) + |Z^-|H(|Z|) \\
&< |Z|H(|Z|) \qquad (\text{since } |Z^+| + |Z^-| = |Z|))
\end{aligned}$$

□

Substituting this result into the initial inequality completes the proof of the theorem.

$$\sum_{Z \in \mathcal{T}^*} \sum_{\{x,y\} \in \rho(Z)} c_{xy} \leq \sum_{Z \in \mathcal{T}^*} |Z| H(|Z|) \leq \sum_{Z \in \mathcal{T}^*} |Z| H(n) \quad = \quad H(n)\mathcal{C}^* \leq (\ln n + 1)\mathcal{C}^*$$

□

4.3.1 Tests with Weights

In many applications, different tests may have different execution costs. For example, in experiment design, a single test might be a good separator of the items, but it may also be expensive. Running multiple, inexpensive tests may serve the same overall purpose, but at less cost. To model scenarios like these we associate a weight $w(k)$ with each bit k and without confusion take $w(S)$ to be the weight of the bit used at node S. We call this problem DT with weighted tests. In the original problem formulation, we can think of each test as having unit weight, so the cost of identifying an item is just the length of the path from the root to the item. When the tests have non-uniform weights, the cost of identifying an item is the sum of the weights of the tests along that path. We call this the path cost. The cost of the tree is the sum of the path costs of each item. When all the tests have equal weight, we choose the bit which most evenly splits the set of items into two groups. In other words, we minimize the pair cost c_{xy}. With equal weights, the cost of an internal node is just its size $|S|$. With unequal weights, the cost of an internal node is the weighted size $w(S)|S|$, so assuming S separates x from y the pair cost becomes

$$c_{xy} = \frac{w(S)}{|S^+|} + \frac{w(S)}{|S^-|} \tag{4.4}$$

and our new greedy algorithm recursively selects the bit which minimizes this quantity. This procedure yields a result equivalent to Theorem 9 for DT with weighted tests. A straightforward implementation on this algorithm still runs in time $O(mn^2)$.

Theorem 10. *The greedy algorithm which recursively selects the bit that minimizes Equation 4.4 yields a* $(\ln n + 1)$*-approximation to DT with weighted tests.*

Proof. Following the structure of the proof for Theorem 9 leads to the desired result. The key observation is that choosing the bit that minimizes Equation 4.4 yields the inequality

$$c_{xy} \leq w(Z)\Big(\frac{1}{|S_{xy} \cap Z^+|} + \frac{1}{|S_{xy} \cap Z^-|}\Big). \tag{4.5}$$

Since the weight term $w(Z)$ may be factored out of the summation

$$w(Z) \sum_{\{x,y\} \in \rho(Z)} \frac{1}{|S_{xy} \cap Z^+|} + \frac{1}{|S_{xy} \cap Z^-|}$$

we can apply the previous claim and the theorem follows:

$$\sum_{Z \in \mathcal{T}^*} \sum_{\{x,y\} \in \rho(Z)} c_{xy} \leq \sum_{Z \in \mathcal{T}^*} w(Z)|Z|H(n) \leq (\ln n + 1)\mathcal{C}^*$$

Here $C^* = \sum_{Z \in \mathcal{T}^*} w(Z)|Z|$ is the cost of the optimal tree. □

Unfortunately, our analysis does not hold for instances of the problem with arbitrary leaf weights. We discuss this further in Section 6.2.2.

4.4 Approximating DT is Hard

In this section we show that there exists a universal constant $\delta > 0$ such that DT has no $(1 + \delta)$ factor approximation unless P=**NP**. This immediately implies that if DT has a PTAS, then P=**NP**. We give a gap-preserving reduction to DT from MAX-3SAT5 which Feige defines in [18]:

Input: A set of n variables $X = \{x_1, \ldots, x_n\}$ and m clauses $C = \{C_1, \ldots, C_m\}$ where each clause has exactly three literals (a literal is a variable or its negation), no variable appears more than once in a clause, and each variable appears in exactly 5 clauses. Note that $m = \frac{5n}{3}$

Output: The maximum number of clauses which can be satisfied simultaneously by some variable assignment

Feige shows that for some $\epsilon > 0$ it is **NP**-hard to distinguish between those 3SAT5 formulas which are satisfiable and those which have at most $(1-\epsilon)|C|$ clauses satisfied simultaneously. Hence, we will restrict our attention to just those instances which are either satisfiable or which, for any assignment, have at least $\epsilon|C|$ clauses that are not satisfied.

The idea is to reduce 3SAT5 to a covering problem which we then reduce to a decision tree. The form of the covering problem is important: with total external path length, cost is a function of leaf depth so to control the cost, we require some control over the depth of the leaves. We show how to reduce instances of 3SAT5 to instances of set cover where every set has size 6 and where every satisfiable instance of 3SAT5 has an exact cover and every unsatisfiable instance requires some constant factor number of sets more for a cover.

The reduction from 3SAT5 to the bounded-size set cover problem uses a variation of the reduction used by Sieling [45] to show that MinDT (a problem quite similar, but distinct from ConDT) has no PTAS.

4.4.1 Reduction from 3SAT5 to Set Cover

Given a 3SAT5 formula ϕ with n variables and m clauses, we define the following polynomial time reduction $f(\phi)$ to a set cover instance (U, Z) where U is a set of items and Z is a collection of subsets of U. Let $Y(i)$ be the indices of the 5 clauses in which x_i appears. First we define the set of items. For each variable x_i, create 11

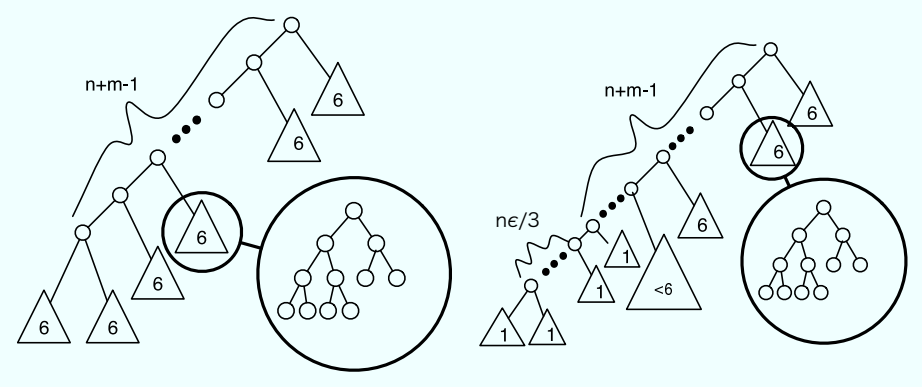

Figure 4.3. (left) The topology of the optimal tree. Here the $n + m - 1$ bits along the left branch correspond to the sets in the set cover. (right) This tree represents the *best case* when $n + m + \frac{n\epsilon}{3}$ sets are required to cover the items. To minimize the cost in this scenario, the extra bits cover a single item and the top of the tree is filled with subtrees of size 6.

items: y_i, a_{ij} (for j in $Y(i)$), and b_{ij} (again, for j in $Y(i)$). For each clause C_j create three items c_{j_1}, c_{j_2}, and c_{j_3}. In total, there are $11n + 3m = 16n$ items, so $|U| = 16n$.

Now we define the sets. For each variable x_i, create two sets:

$$S_i^a = \{y_i\} \cup \{a_{ij} \,|\, j \in Y(i)\} \qquad \text{and} \qquad S_i^b = \{y_i\} \cup \{b_{ij} \,|\, j \in Y(i)\}$$

Call these sets the *variable* sets since they are constructed from the variables of the formula. For each clause C_j, create seven sets called the *clause sets*. Construct each set in the following way: Let x_{u_1}, x_{u_2}, and x_{u_3} be the variables appearing in clause j. For each local satisfying assignment $z(x)$ (there are exactly seven), create the set

$$S_j^z = \{c_{j_1}, c_{j_2}, c_{j_3}\} \cup \{d_1, d_2, d_3\} \quad \text{where } d_k = \begin{cases} b_{u_k j} & \text{if } z(x_{u_k}) = 1 \\ a_{u_k j} & \text{if } z_(x_{u_k}) = 0 \end{cases}$$

For example if $C_j = (x_1 \vee \bar{x_2} \vee x_3)$ then there are eight possible local assignments to the variables. The assignment $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ satisfies the clause so the set $S_j^{100} = \{c_{j_1}, c_{j_2}, c_{j_3}\} \cup \{b_{1j}, a_{2j}, a_{3j}\}$ is included. However, the assignment $x_1 = 0$, $x_2 =$

1, and $x_3 = 0$ fails to satisfy the clause, so the set $S_j^{010} = \{c_{j_1}, c_{j_2}, c_{j_3}\} \cup \{a_{1j}, b_{2j}, a_{3j}\}$ is not included. Since there are seven locally satisfying assignments per clause, there are a total of $7m$ sets in Z. Note that every set has exactly 6 items. The key to the reduction is that the a_{ij} correspond to a positive assignment of x_i and the b_{ij} correspond to a negative assignment to x_i. The clause sets include the items which are opposite of the assignment, so when a formula is satisfiable, it is always possible to cover it with $n+m$ sets. However, if no satisfying assignment exists for an assignment then more sets are required. Though our reduction is not identical to the one given by Sieling, the following theorem still holds:

Theorem 11 ([45])**.** *If ϕ is a satisfiable 3SAT5 formula then there is a solution to $f(\phi)$ of size $n + m$. If, for any assignment to ϕ, at most $(1 - \epsilon)m$ clauses may be satisfied simultaneously, then a solution to $f(\phi)$ has size at least $n + m + \frac{\epsilon n}{3}$.*

Proof. It's clear that $n + m$ sets are necessary to cover U, but when ϕ is satisfiable, $n+m$ sets are also sufficient. If z is a satisfying assignment to ϕ then for each variable x_i, choose S_i^a if $z(x_i) = 1$ or choose S_i^b if $z(x_i) = 0$. For each clause j, choose S_j^z where S_j^z is the set for clause j corresponding to assignment z. It's clear that all the y_i and c_j items are covered. Every a_{ij} item is also covered: If $z(x_i) = 1$ then S_i^a covers it. If $z(x_i) = 0$ then S_j^z covers it. The b_{ij} cases are symmetric, so $n + m$ sets are sufficient. If ϕ is not a satisfying assignment then more sets are required. To see this, let Y be a solution to the the set cover problem. Let Y' be a subset of Y such that for all i, Y' contains the first occurrence of a set covering y_i and, for all j, the first occurrence of the set covering c_j. The variable sets of Y' define an assignment z to the variables. At least ϵm clauses aren't satisfied by this assignment. Let C_j be such a clause, say (x_1, x_2, x_3). Since it isn't satisfied, $z(x_i) = 0$ for $i \in [1, 3]$. So all the b_i for $i \in [1, 3]$ are covered by the variables sets, but the clause set j cannot cover the remaining a_i because, by construction, it doesn't exist. To cover the remaining items requires at least one more variable set. This has the potential of covering at most 5

other uncovered items. So if ϕ is unsatisfiable we require at least $\frac{\epsilon m}{5} = \frac{\epsilon n}{3}$ more sets. For more details, we refer the reader to [45]. Note that when ϕ is satisfiable the set cover is an *exact* cover meaning no two sets share the same item. □

We now define a gap-preserving reduction g from instances of set cover (U, Z) of the form given by f to instances X of DT. Let $|U| = n'$ and $|Z| = m'$. For each item u in U create a binary string of length $4m'$. The first m' bits correspond to the m' sets of Z so we call them *Z-bits*. A string for item u has Z-bit i set to 1 if and only if u is in set Z_i. In other words, the first m' bits denote set membership in the m' sets in Z. The final $3m'$ bits allow us to disambiguate the items from one another—recall that each set has six items, so if we use a Z-bit at node T in the decision tree, then at most 6 items may follow the 1-branch of T. We want to control the shape of the subtree formed by these items. That is, we want them to form as near a complete tree as possible. Hence, each set Z_i in Z adds 3 more bits per string. These bits are set to 0 for all items not appearing in Z_i. For those items which do appear in Z_i, we assign the bits so that a near-complete tree of height 3 is always possible (e.g., half the items will have the first bit set to 1, the other half will have them set to 0, and so on. See Figure 4.3 for details). Together, these n' strings comprise the DT instance X. Note that the number of items does not change and that the reduction is polynomial in n' and m'.

Claim 5. *if ϕ is a satisfiable 3SAT5 formula then $g(f(\phi))$ has a tree with cost at most $\frac{64n^2}{3} + \frac{152n}{3} - 6$.*

Proof. All we need to do is construct the cascading tree pictured on the left in Figure 4.3. The bits along the left side correspond to the optimal set cover, except for the final 3 bits (which are used to disambiguate the final remaining set). Selecting the $n + m - 1 = n + (5/3)n - 1$ sets yields a cost of

$$\mathcal{C} = (-6) + \sum_{i=1}^{n+5/3n} 6i + 16 = \frac{64n^2}{3} + \frac{152n}{3} - 6.$$

□

In fact, we can also show that $\mathcal{C}$ is the minimum cost. To see this, imagine a tree $\mathcal{T}$ corresponding to some sub-optimal cover of the items. This tree has depth at least $n + m + 3$. Since every bit partitions off at most 6 items, we know $\mathcal{T}$ is also a cascading-style tree, but that some of its subtrees have fewer than 6 items. It's easy to show that shifting a leaf lower in the tree to a leaf higher in the tree decreases the total cost. This is because the complete binary tree minimizes total external path length. Hence it is advantageous to maximize the number of subtrees of size 6 that occur higher in the tree. Since the tree corresponding to the optimal set cover is composed entirely of subtrees with size 6, it has minimum cost. What's left to show is that g preserves the gap in cost when the set cover requires at least $n+m+\frac{\epsilon n}{3}$ sets.

Claim 6. *if ϕ is a 3SAT5 formula such that, for any assignment to ϕ, at most $(1-\epsilon)|C|$ clauses are simultaneously satisfied, then any solution to $g(f(\phi))$ has cost at least $\mathcal{C} + \frac{n^2\epsilon^2}{18} + \frac{n\epsilon}{6} - 1$.*

Proof. From Theorem 11 we know that at least $\frac{n\epsilon}{3}$ additional sets are required to cover all the items. Given that the tree must have depth at least $n+m+n\epsilon/3-1$ for large enough n and that any bit can partition off at most 6 items, what is the minimum cost tree which adheres to these constraints? In the best case, each of the additional $\frac{n\epsilon}{3}$ internal nodes partitions off only a single element with the remaining $n' - \frac{n\epsilon}{3} + 1$ items partitioned off by the first $n + m$ internal nodes. In the best case, these will be subtrees of size 6 where possible. This is illustrated on the right in Figure 4.3. Another way to think about this tree is starting with the optimal tree from Claim 5 and repeatedly creating a new leaf at a lower level by deleting a leaf at a higher level. This is tantamount to shifting a leaf. This process is a convenient way to analyze the

cost since the first leaf shifted adds at least 1 to the total cost, the second leaf shifted adds at least 2 to the total cost, and so forth. Hence, the cost of the optimal tree for the gap case is at least $\mathcal{C} + 1 + 2 + \ldots + \frac{n\epsilon}{3} - 1 = \mathcal{C} + \frac{n^2\epsilon^2}{18} + \frac{n\epsilon}{6} - 1$. □

Combining Claims 5 and 6 yields the desired result:

Theorem 12. *There exists a $\delta > 0$ such that DT cannot be approximated in polynomial time within a factor of $(1 + \delta)$ unless P=**NP**.*

Proof. Suppose that for all $\delta > 0$, DT could be approximated in polynomial time within a factor of $(1 + \delta)$. Let $0 < \epsilon < 1$ be given so that the ϵ-gap in satisfiable 3SAT5 formulas and unsatisfiable 3SAT5 formulas exists. Choose $\delta < \frac{\epsilon^2}{912}$. Now any satisfiable instance has cost at most $(1 + \delta)\mathcal{C} < \mathcal{C} + \frac{n^2\epsilon^2}{18} + \frac{n\epsilon}{6} - 1$ for $n > \frac{6}{\epsilon}$. By Claim 6, this gives us a polynomial time procedure for distinguishing satisfiable 3SAT5 formulas from unsatisfiable formulas — a contradiction unless P=**NP**. □

4.4.2 ConDT under Total External Path Length

Recall from Section 2.4 that ConDT instances are n binary strings, each of length m, augmented with a TRUE/FALSE label. A solution is a binary tree where each internal node is a bit position, each leaf is a label, and the tree correctly labels all the binary strings. The size of a tree is the number of leaves. The goal is to find a tree of minimal size. Alekhnovich et. al. [1] showed it is not possible to approximate size s decision trees by size s^k decision trees for any constant $k \geq 0$ unless **NP** is contained in DTIME[2^{m^ϵ}] for some $\epsilon < 1$.

In this section we show that these hardness results also hold for ConDT under minimum total external path length. Our theorem relies on the observation that if I is an instance of ConDT with minimum total external path length s then I has minimum tree size at least $\Omega(\sqrt{s})$. If it didn't, a tree of smaller size would have smaller total external path length, a contradiction. The case where minimum total

external path length s corresponds to minimum size $\Omega(\sqrt{s})$ is a cascading tree; that is, a tree with exactly one leaf at each depth save the deepest two.

Theorem 13. *If there exists an s^k approximation for some constant $k > 0$ to decision trees with minimum total external path length s then* **NP** *is contained in DTIME[2^{m^ϵ}] for some $\epsilon < 1$.*

Proof. Let I be an instance of ConDT with minimum total external path length $s = r^2$. It follows that I has minimum tree size at least $\Omega(r)$. Now, if an s^k approximation did exist for some k then there would exist an $\Omega(r^{2k}) = r^{k'}$ approximation for some constant k' for ConDT under minimum tree size; a contradiction. □

4.5 New Lower Bounds on Approximating MinDT

A problem similar to ConDT is finding small, equivalent decision trees. This problem, called MinDT, takes as input a decision tree T over n variables of the ConDT type (*i.e.*, a function from $\{0,1\}^n \to \{0,1\}$) and seeks the smallest decision tree T' (smallest, again, in terms of number of leaves) which is functionally equivalent to T. Here, functionally equivalent means that T and T' compute the same function on all binary strings of length n. Zantema and Bodlaender [49] first defined this problem and showed it to be **NP**-Hard. Sieling [45] showed that MinDT has no constant factor approximation unless P=**NP**. This negative result follows from a self-improving property of decision trees which is consistent with the squaring results from [28]. We review the self-improving property here, show how to generalize it, and then apply techniques from [28] to show that no $2^{log^\delta s}$ approximation exists (where s is the size of the optimal tree and $\delta < 1$) unless **NP** $\subseteq$ DTIME[$2^{\text{polylog}\, n}$].

Lemma 14 ([45])**.** *Let f and g be boolean functions over n and m disjoint binary variables respectively. If MIN($f \oplus g$) is the size of the minimum decision tree computing $f \oplus g$ then MIN($f \oplus g$) = MIN(f) $\cdot$ MIN(g). Similarly, if T is a tree of size $|T|$*

which computes $f \oplus g$ then T_f, a tree which computes f and T_g, a tree which computes g can be constructed in polynomial time such that $|T_f| \cdot |T_g| \leq |T|$.

The proof of Lemma 14 divides up T into f-regions and g-regions based on the largest contiguous parts of the tree which examine bits exclusively from f or g. The idea is that regions near the leaves may be switched with their parents to form larger contiguous regions until T_f and T_g are easily identifiable. The generalization of this result requires a few more details:

Lemma 15. *Let F be a set of r binary functions such that $f_i : \{0,1\}^{n_i} \to \{0,1\}$. Let $g = \oplus_{i=1}^{r} f_i$. If MIN(g) is the size of the minimum decision tree computing g then MIN(g) $= \prod_{i=1}^{r}$ MIN(f_i). Similarly, if T is a tree of size $|T|$ which computes g then in polynomial time, we can construct a tree T_i for each f_i such that $\prod_{i=1}^{r} |T_i| \leq |T|$.*

Proof. The proof is similar to the generalization of the squaring technique used in [28]. Call a tree *reduced* when no path contains the same variable twice. In this proof we'll assume all trees are reduced since one can reduce a tree very quickly. It is easy to show that $\text{MIN}(g) \leq \prod_{i=1}^{r} \text{MIN}(f_i)$ — just build the tree using concatenated copies of the functions in F. This yields a tree for g with size exactly $\prod_{i=1}^{r} \text{MIN}(n_i)$. The other direction is more complicated. Let x_i denote a variable from function f_i. A tree is in standard form if, on any path from the root to a leaf, variable x_i is followed only by variables $x_{i'}$ where $i' \geq i$. If the root of a tree (or a subtree) is x_i then let $s(x_i) = (s_1 \ldots s_t)$ denote the roots of the subtrees of x_i which are the first nodes along a path from the root having variables which differ from i.

A tree rooted at x_i is called *subtree equivalent* if every tree rooted at one of $s(x_i)$ is structurally identical except for the leaf labels. Once a tree is in standard form and is subtree equivalent it is easy to recover functions computing each f_i. Hence any subtree equivalent tree computing g in standard form must have size greater than or equal to the product of the size of its composite functions. We often denote a tree

by its root, so saying x_i *is in standard form* really means *the tree rooted at* x_i *is in standard form.* We now show how to take any tree T computing g and turn it into a tree which is subtree equivalent and in standard form. The proof is by induction on the height of subtrees of T which are in standard from and subtree equivalent. It is trivially true that each leaf is subtree equivalent and in standard form. Now consider an internal node x_i with left child $x_{i'}$ and right child $x_{i''}$ both in standard form and subtree equivalent. Without loss of generality, let $i' \leq i''$. We must consider two cases: If $i \leq i'$ then x_i is in standard form. Furthermore, we make the following claim:

Claim 7. *Any two trees from* $s(x_i)$ *are functionally equivalent up to polarity.*

Proof. Suppose two of the $s(x_i)$ trees compute different functions. Then there are two strings w_1 and w_2 that differ only in the f_i variables (specifically those at x_i and beyond) and have different outputs on $g \oplus f_i$ (g with f_i removed). This is a contradiction since $g \oplus f_i$ ignores the f_i variables so $g \oplus f_i(w_1) = g \oplus f_i(w_2)$. □

Now we can take the smallest of the $s(x_i)$ and replicate it across the remaining $s(x_i)$, negating the output on the leaves when appropriate. This tree rooted at x_i is now subtree equivalent and the induction holds. On the other hand, if $i > i'$ then x_i is not in standard form. Using the same type of argument from Claim 7 we can show that $x_{i'}$ and $x_{i''}$ are functionally equivalent up to differences in f_i. If f_i is not constant with respect to x_i then find the tree among $x_{i'}$ and $x_{i''}$ with the smallest total number of nodes leading up to its f_i regions. Without loss of generality, say this is $x_{i'}$. Insert x_i above each f_i subtree in $x_{i'}$ and make f_i the left child of x_i. Now, choose the root of any f_i region from $x_{i''}$ (they're all the same except for the labels) and make it (and its subtree) the right child of each x_i, again, negating the labels when appropriate. It's clear that for $x_i = 0$, the tree rooted at $x_{i'}$ computes the same value as the old tree. To see why it's true for $x_i = 1$, recall that $x_{i'}$ and

$x_{i''}$ are functionally equivalent up to f_i so every f_j region ($j \neq i$) must independently be able to compute its proper label, hence all the f_j are sufficient discriminators. By construction $x_{i'}$ is still in standard form and subtree equivalent. We delete $x_{i''}$, promote $x_{i'}$ to root, and by virtue of it having the smaller subtree leading up to each f_i, we know it is smaller than the original x_i tree. □

With Lemma 15 in hand, it's possible to prove the following theorem:

Theorem 14. *For any $\delta < 1$, there is no $2^{\log^{\delta} s}$ approximation for MinDT where s is the size of the optimal tree, unless* **NP** *is quasi-polynomial.*

The proof is essentially identical to one given for the analogous ConDT result. We give it here for completeness.

Proof. Suppose such an approximation did exist. Take the given tree T of size n and raise it to the power d where $d = \log^r(n)$ and where $r = 2\delta/(1-\delta)$. This takes time $n^d = n^{\log^r(n)} = 2^{\text{polylog}(n)}$. The smallest solution for T^d has size s^d so our approximation gives us a solution of size at most $s^d 2^{\log^{\delta}(s^d)}$ from which we can find a tree for T of size:

$$s2^{d^{\delta-1}\log^{\delta}(s)} = s2^{\log^{\frac{2\delta(\delta-1)}{1-\delta}}(n)\log^{\delta}(s)} = s2^{\log^{-2\delta}(n)\log^{\delta}(s)} = s2^{\log^{-2\delta}(n)\log^{\delta}(s)}$$

which is $O(s)$. This contradicts the no constant factor approximation result by Sieling. □

Lemma 15 is tantamount to repeatedly squaring the problem and improving the approximation. Here we show that no $\log(n)$ approximation exists for MinDT unless **NP** is contained in $O(n^{\log\log(n)})$ time. By increasing the number of iterations we can also prove Theorem 14.

Theorem 15. *If an instance of MinDT with smallest solution size s has a $\log(s)$ approximation then* **NP** *is $n^{O(\log\log(n))}$.*

Proof. Suppose a $\log(s)$ approximation exists. Then after the kth iteration of squaring, we have an $(\log(s))^{1/2^k}$ approximation. So after $k = \log(\log(\log(s)))$ iterations we arrive at a constant factor approximation. Since each iteration effectively doubles the problem size, the total time for k iterations is

$$n^{2^k} = n^{2^{\log\log\log(s)}} = n^{\log(\log(s))} = n^{O(\log\log(s))}$$

The theorem follows since $s \leq n$. □

CHAPTER 5

SEARCH SPACE REDUCTIONS IN R-TREES

With the prevalence and demand for applications featuring image, sound, and video data pervading the computing world, so too arises a demand for data structures and algorithms that enhance the storage and retrieval of multimedia data. Often data of this type has an intrinsic geometric flavor or is transformed into some geometric representation like a vector of features. An appropriate and popular structure for storing and querying geometric data is the R-tree [27, 4].

A common operation on R-trees (and other indices) is the similarity query. For example, if we are an online retailer of music then we might want to recommend recording artists to patrons based on music they've previously purchased. If we characterize popular songs by some high-dimensional feature vector (where features have semantic meaning) then similar songs occupy locations near each other in feature space. Given a song, the goal is to search a collection music and identify similar songs, or rather, to find the closest songs in feature space. This type of search is called a nearest-neighbor query.

5.1 R-trees

R-trees and their variants [27, 44, 3] are data-structures for organizing spatial objects in Euclidean space. They support dynamic insertion and deletion. Like most trees, R-trees are decomposed into internal and leaf nodes. Both types of nodes contain records, but records of internal nodes differ slightly from those of leaf nodes. A record r, belonging to an internal node, is a tuple $\langle M, \mu \rangle$ where μ is a pointer to the

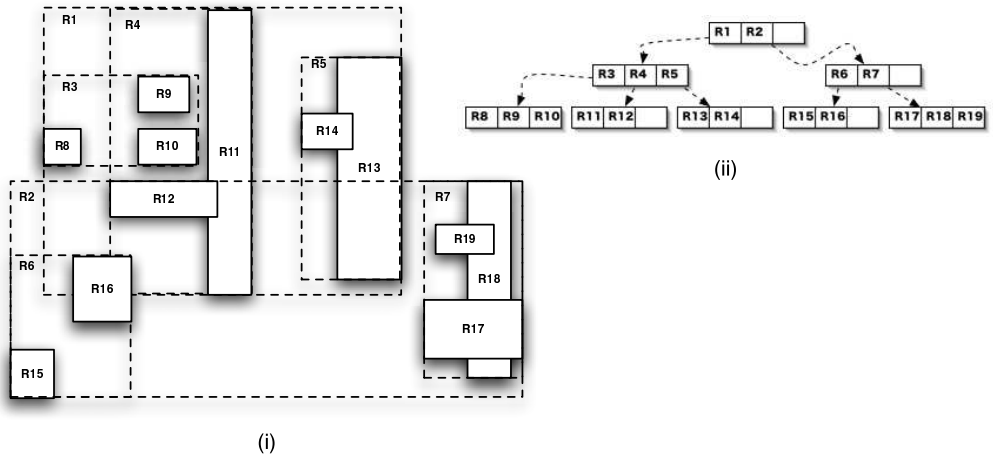

Figure 5.1. (i) A collection of spatial objects (solid lines) and their hierarchy of minimum bounding rectangles (dashed lines). (ii) The R-tree for the objects in (i).

child node of r and M is an n-dimensional minimum bounding rectangle denoted by MBR. M tightly bounds the spatial objects located in the subtree of r. For example, given the points $(1, 2)$, $(4, 5)$, and $(3, 7)$ in 2-space, the MBR would be $\langle(1, 4), (2, 7)\rangle$. The records of leaf nodes are also tuples but have the form $\langle M, o\rangle$ where o is either the actual spatial object or a reference to it. M is the minimum bounding rectangle of o. For our purposes, we also assume that spatial objects are unique to a particular R-tree. That is, no two leaf node records refer to the same spatial object in memory.

The number of records in a node is its branching factor. Every node of an R-tree contains between b and B records where $b \leq \lfloor \frac{B}{2} \rfloor$. One exception is the root node which must have at least two records. R-trees are completely balanced, meaning all leaf nodes have the same depth. Figure 5.1 depicts an example collection of spatial objects, their MBRs, and their R-tree.

For explanatory purposes, we often refer to the minimum bounding rectangle of a node instead of the minimum bounding rectangle of the record pointing to that node. Similarly, when we refer to the children of a node, we mean the children pointed to by the records of the node.

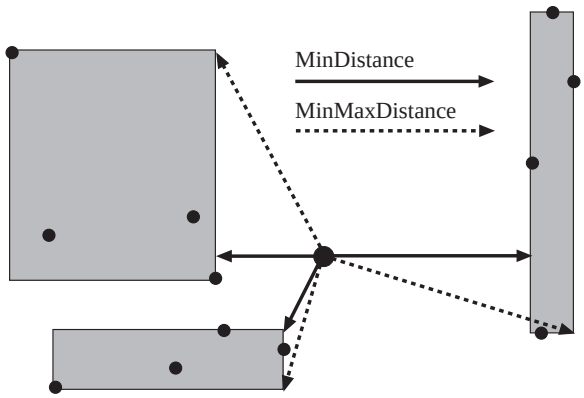

Figure 5.2. A visual explanation of MINDISTANCE and MINMAXDISTANCE in two dimensions.

Since every non-root node is tightly enclosed by a minimum bounding rectangle, meaningful spatial relationships exist among nodes. We exploit these relationships when performing nearest-neighbor queries on R-trees.

5.2 Pruning Strategies

Generally, nearest neighbor queries have the form: *Find the k nearest items with respect to some other item.* For R-trees, nearest neighbor queries have the form: *Given a query point q and an R-tree T with spatial objects of matching dimension to q, find the k nearest objects to q in T.* Nearest here and throughout the rest of the paper, is defined by *Euclidean distance.*

As mentioned in Section 2.6, pruning strategies for nearest neighbor queries in R-trees employ two metrics. The first metric, MINDISTANCE(q, M), is the length of the shortest line between the query point q and the nearest face of the MBR M. When the query point lies within the MBR, MINDISTANCE is 0. Figure 5.2 shows the MINDISTANCE values for a query point and three minimum bounding rectangles. Because an MBR tightly encapsulates the spatial objects within it, each face of the MBR must touch at least one of the objects it encloses [42]. This is called the MBR face property.

Figure 5.2 shows that MINDISTANCE is a lower bound, or optimistic estimate of the distance between the query point and some spatial object inside its MBR. However, the actual distance between a query point and the closest object may be much larger.

The second metric, MINMAXDISTANCE, is an upper bound, or pessimistic estimate of the distance between the query point and some spatial object within its MBR. Figure 5.2 depicts these distances for three MBRs and a query point. From its name one can see MINMAXDISTANCE is calculated by finding the minimal distance from a set of maximal distances.

This set of maximal distances is formed as follows: Suppose we have an n-dimensional minimum bounding rectangle. If we fix one of the dimensions, we are left with two $n-1$ dimensional hyperplanes; one representing the MBR's lower bounds, the other representing its upper bounds. We know from the MBR face property that at least one spatial object touches each of these hyperplanes. However, given only the MBR, we cannot identify this location. But, given a query point, we can say that an object is at least as close as the distance from that point to the farthest point on the closest hyperplane. This distance is an upper bound on the distance between the query point and a spatial object located within the MBR. By iteratively fixing each dimension of an MBR and finding the upper bound, we can form the set of maximal distances. Since each maximal distance is an upper bound, it follows that the minimum of these is also an upper bound. This minimum distance is what we call the MINMAXDISTANCE(q, M) of a query point q and an MBR M.

Pruning strategies based on these metrics potentially remove large portions of the search space in nearest neighbor queries. The following three strategies were originally defined in [42] for use in the depth-first search RKV algorithm. All assume a query point q and a list of MBRs $\mathcal{M}$ (to potentially prune) sorted by MINDISTANCE. The

latter assumption is based on empirical results from both [42] and [29]. In addition, two strategies, S2 and S3, assume a current neighbor estimate e.

Definition 11 (S1)**.** *Discard any MBR $M_i \in \mathcal{M}$ if there exists $M_j \in \mathcal{M}$ such that MINDISTANCE$(q, M_i) >$ MINMAXDISTANCE(q, M_j)*

Definition 12 (S2)**.** *Replace e with MINMAXDISTANCE(q, M_i) if there exist $M_i \in \mathcal{M}$ such that MINMAXDISTANCE$(q, M_i) < e$.*

Definition 13 (S3)**.** *Discard any minimum bounding rectangle $M \in \mathcal{M}$ if MINDISTANCE$(q, M) > e$*

Generalizing S1 and S3 to the k nearest neighbors is straight-forward enough: replace e with a priority queue L of k closest neighbor estimates and alter the definitions as follows.

Definition 14 (K1)**.** *Discard any minimum bounding rectangle $M_i \in \mathcal{M}$ if there exists $\mathcal{M}' \subseteq \mathcal{M}$ such that $|\mathcal{M}'| \geq k$ and for every $M_j \in \mathcal{M}'$ it is the case that MINDISTANCE$(q, M_i) >$ MINMAXDISTANCE(q, M_j)*

Definition 15 (K3)**.** *Discard any minimum bounding rectangle $M_i \in \mathcal{M}$ if MINDISTANCE$(q, M_i) >$ Max(L) where Max returns the largest estimate in the priority queue.*

Both [7] and [29] show that S3 dominates S1 for the 1-nearest neighbor query. Because of this, we take RKV to mean the original algorithm given in [42] sans S1. A proof of the general case is a simple extension:

Theorem 16. *Given a query point q, a list of MBRs $\mathcal{M}$, and a priority queue L of k closest neighbor estimates L, any MBR pruned by K1 in the depth-first RKV algorithm is also pruned by K3.*

Proof. Suppose we're performing a k nearest neighbor search with query point q and K1 prunes MBR M from $\mathcal{M}$. From Definition 14 there exists $\mathcal{M}' \subset \mathcal{M}$ such that $|\mathcal{M}'| \geq k$ and every M' in $\mathcal{M}'$ has MINMAXDISTANCE(q, M') < MINDISTANCE(q, M). Since for any MBR N, $MINDISTANCE(q, N) \leq MINMAXDISTANCE(q, N)$, each M' in $\mathcal{M}'$ will be searched before M because $\mathcal{M}$ is sorted by MINDISTANCE. Because each M' in $\mathcal{M}'$ is guaranteed to contain a spatial object with actual distance at most that of than an object found in M and since we have $|\mathcal{M}'| \geq k$ we know Max$(L) < \delta(q, M)$. Therefore, from Definition 15, M would also be pruned using K3. □

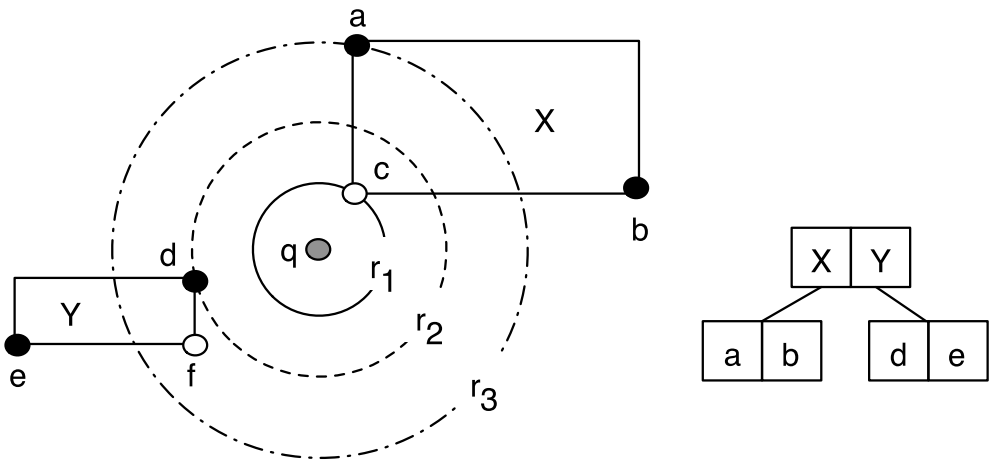

Figure 5.3. Blindly inserting promises into the queue, without removing them correctly, wreaks havoc on the results. For example, when performing a 2-nearest neighbor search on the tree above, a promise with distance f is placed in the queue at the root. After investigating X, the queue retains f and a. However, if f is not removed before descending into Y, the final distances in the queue are e and f — an incorrect result.

Like S1, S2 doesn't perform direct pruning, but instead, updates the neighbor estimate when distance guarantees can be made. If the MINMAXDISTANCE of a node is less than the current distance estimate, then we can update the estimate because the future descent into that node is guaranteed to contain an object with actual distance at most MINMAXDISTANCE. We call estimates in updates of this

form *promises* because they are not actual distances, but are upper bounds on distances. Moreover each estimate is a promise of, or place holder for, a spatial object that is as least as good the promise's prediction. A natural but incorrect generalization of S2 places a promise in the priority queue whenever the maximum-distance element is further away than the MINMAXDISTANCE. This leads to two problems. First, multiple promises may end up referring to the same spatial object. Second, a promise may persist past its time and eventually refer to a spatial object already in the queue. The key to avoiding both problems is to always remove a promise from the queue before searching the node which generated it; it will either be replaced by a better estimate or by an actual object. The leads us to the following generalization of S2 which we call PROMISE-PRUNING:

Definition 16 (PROMISE-PRUNING)**.** *If there exists $M_i \in \mathcal{M}$ such that $\delta(q, M_i) = MINMAXDISTANCE(q, M_i) < Max(L)$, then add a promise with distance $\delta(q, M_i)$ to L. Additionally, replace any promise with distance $\delta(q, M_i)$ from L with ∞ before searching M_i.*

Figure 5.4 describes the RKV algorithm for k nearest neighbor queries augmented with PROMISE-PRUNING. We call the algorithm RKV-PP. Line 10 performs K3 pruning while lines 7-8 perform PROMISE-PRUNING. Note the importance of line 12. If the check is not performed, the results are erroneous as witnessed by the example in Figure 5.3. However, when performing a 1-nearest neighbor query, lines 11-12 may be safely ignored since an actual object will eventually replace the promise. In this case, RKV-PP becomes RKV.

As an aside, the description of RKV given in [4] differs from the one given above (and hence the original) as it performs S2 after processing a node. This has no benefit since any promise offered by the node will already be realized. It is equivalent to RKV with S2 removed.

```
RKV-PP(k, q, T)
    ▷ k is the desired number of nearest neighbors, Q is a query point, and T is an R-tree
1   L ← empty priority queue
2   TRAVERSE(k, q, root[T], L)
3   return L


KNEARESTTRAVERSAL(k, q, n, L)
     ▷ n is an R-tree node
 1   if LEAFNODE(n)
 2      then for⟨M, μ⟩ in records[n]        ▷ M is a MBR, μ is a spatial object
 3                do if DIST(q, M) < MAX(L)
 4                      then INSERT(L, μ)
 5      else  ABL ← SORT(records[n])
               ▷ Sort the records by MINDISTANCE into an Active Branch List
 6             for⟨M, μ⟩ in ABL
 7                do if MINMAXDISTANCE(q, M) < MAX(L) ▷ PROMISE-PRUNING
 8                      then INSERT(L, PROMISE(MINMAXDISTANCE(q, M)))
 9             for⟨M, μ⟩ in ABL
10                do if MINDISTANCE(q, M) < MAX(L) ▷ K1 pruning
11                      then if L contains a promise generated from M
12                              then REMOVE(L, PROMISE(MINMAXDISTANCE(q, M)))
13                           KNEARESTTRAVERSAL(k, q, μ, L)
```

Figure 5.4. The RKV-PP algorithm: RKV, without S1, extended to k nearest neighbors and augmented with PROMISE-PRUNING.

5.3 The Power of MINMAXDISTANCE

In the previous section we showed that K3 dominates K1. The result was based on proofs from [7] and [29] that showed that S3 dominates S1. Those papers also show that S3 dominates S2 on 1-nearest neighbor queries. We provide a counterexample to this claim. In fact, our counterexample constructs a class of R-trees where RKV with S2 provides an exponential improvement over RKV without S2 on a 1-nearest neighbor query.

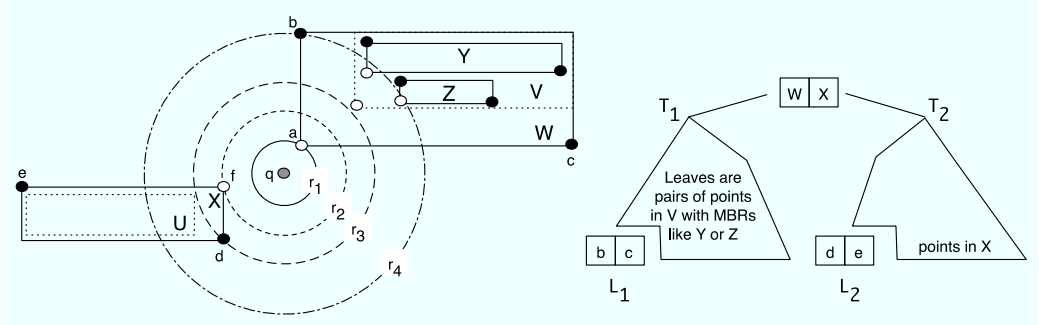

Figure 5.5. A visual explanation of the tree construction

Theorem 17. *There exists a family of R-tree and query point pairs* $\mathcal{T} = \{(T_1, q_n), \ldots, (T_m, q_m)\}$ *such that for any* (T_i, q_i)*, RKV without S2 examines* $O(n)$ *nodes and RKV with S2 examines* $O(\log n)$ *nodes on a 1-nearest neighbor query.*

Proof. For simplicity, we restrict our attention to R-trees composed of points in $\mathbb{R}^2$ so that all the MBRs are rectangles. Also, we only construct complete binary trees where each node has two records. The construction follows the illustration in Figure 5.5. Let $\delta(i, j)$ be the Euclidean distance from point i to point j and let (i, j) be their rectangle. Let q be the query point. Choose three points a, b, and c such that $\delta(q, a) = r_1 < \delta(q, b) = r_4 < \delta(q, c)$ and (b, c) forms a rectangle W with a corner a. Similarly, choose three points d, e, and f such $r_1 < \delta(q, f) = r_2 < \delta(q, d) = r_3 < r_4 < \delta(q, e)$ and (d, e) forms a rectangle X with corner f. Let T be a complete binary tree over n leaves where each node has two records. Let T_1 be the left child of T and let T_2 be the right child of T. Let L_1 be the far left leaf of T_1 and let L_2 be the far left leaf of T_2. Place b and c in L_1, and d and e in L_2. In the remaining leaves of T_1, place pairs of points (p_i, p_j) such that p_i and p_j are interior to V, $\delta(q, p_i) > r_4$ and $\delta(q, p_j) > r_4$ but (p_i, p_j) form a rectangle with corner p' such that $r_3 < \delta(q, p') < r_4$. Rectangles Y and Z in 5.6 are examples of this family of point pairs. In the remaining leaves of T_2 place pairs of points (p_k, p_l) such that p_k and p_l are interior to U (*i.e.*, so that MINDISTANCE$(q, (p_k, p_l)) > r_3$). The construction yields a valid R-tree because

we place pairs of points at the leaves and build up the MBRs of the internal nodes accordingly.

Claim 8. *Given a tree T and query point q as constructed above, RKV with and without S2 both prune away all of T_2 save the left branch down to L_2 on a 1-nearest neighbor query.*

Proof. Note that d is the nearest neighbor to q in T so both algorithms will search T_2. L_2, by construction, has the least MINDISTANCE of any subset of points in T_2, so both algorithms, when initially searching T_2, will descend to it first. Since $\delta(q, d)$ is the realization of this MINDISTANCE and no other pair of points in X has MINDISTANCE $< \delta(q, d)$, both algorithms will prune away the remaining nodes using S3. □

Now we'll show that RKV without S2 must examine all the nodes in T_1 while RKV with S2 can use information from X to prune away all of T_1 save the left branch down to L_1.

Lemma 16. *Given an R-tree T and query point q as constructed above, RKV without S2 examines every node in T_1 on a 1-nearest neighbor query.*

Proof. Since MINDISTANCE$(q, W) = \delta(q, a)$, RKV descends to L_1 first and claims b as its nearest neighbor. However, RKV is unable to prune away any of the remaining leaves of T_1. To see this, let $L_i = (p_{i1}, p_{i2})$ and $L_j = (p_{j1}, p_{j2})$ be distinct leaves of T_1 (but not L_1). Note that MINDISTANCE$(q, (p_{i1}, p_{i2})) < r_4 < \min(\delta(q, p_{j1}), \delta(q, p_{j2}))$ and MINDISTANCE$(q, (p_{j1}, p_{j2})) < r_4 < \min(\delta(q, p_{i1}), \delta(q, p_{i2}))$. This means that the MINDISTANCE of any leaf node is at most r_4 but every point is at least r_4 so RKV must probe every leaf. As a result, it cannot prune away any branches. □

Lemma 17. *Given a tree T and query point q as constructed above, RKV with S2 prunes away all nodes in T_1 except those on the branch leading to L_1 in a 1-nearest neighbor query.*

Proof. RKV with S2 uses the MINMAXDISTANCE information from X as an indirect means of pruning. Before descending into T_1, the algorithm updates its neighbor estimate with $\delta(q, d)$ (*i.e.*, it adds a promise with distance MINMAXDISTANCE$(q, X) = \delta(q, d)$ into its 1-best priority queue). Like RKV without S2, RKV with S2 descends into T_1 directly down to L_1 since $\delta(q, W) < \delta(q, d)$ and W has the smallest MINDISTANCE of all the nodes. Unlike RKV without S2, it reaches and ignores b because $\delta(q, d) < \delta(q, b)$. In fact, the promise granted by X allows us to prune away all other branches of the tree since the remaining nodes are all interior to V and $\delta(q, d) <$ MINDISTANCE(q, V). □

The theorem follows from Lemma 16 and Lemma 17. RKV without S2 searches all of T_1 ($O(n)$ nodes) while RKV with S2 searches only the paths leading to L_1 and L_2 ($O(\log n)$ nodes). As a consequence, RKV with S2 can reduce the search space exponentially over RKV without S2. □

5.4 Search Space Reductions with RKV-PP

Here we show that the benefits RKV reaps from MINMAXDISTANCE (and in particular pruning strategy S2) extend to k nearest neighbors queries when S2 is properly generalized under Promise-Pruning. In particular, we construct a class of R-trees where RKV-PP reduces the number of nodes visited exponentially when compared with RKV sans Promise-Pruning (and where the naive generalization of S2 — where the promise is not removed prior to descent — yields erroneous results).

Theorem 18. *There exists a family of R-trees and query point pairs $\mathcal{T} = \{(T_1, q_n), \ldots, (T_m, q_m)\}$ such that for any (T_i, q_i) RKV without*

PROMISE-PRUNING *(hereafter just RKV) examines* $O(n)$ *nodes and RKV-PP examines* $O(\log n)$ *nodes on a 2-nearest neighbor query.*

Proof. The proof follows the outline of Theorem 17. The construction is identical to Figure 5.5 except that b shifts slightly down and V shifts slightly up so that $\delta(q, b) <$ MINDISTANCE(q, V). We illustrate this in Figure 5.6 and give details where the two proofs differ.

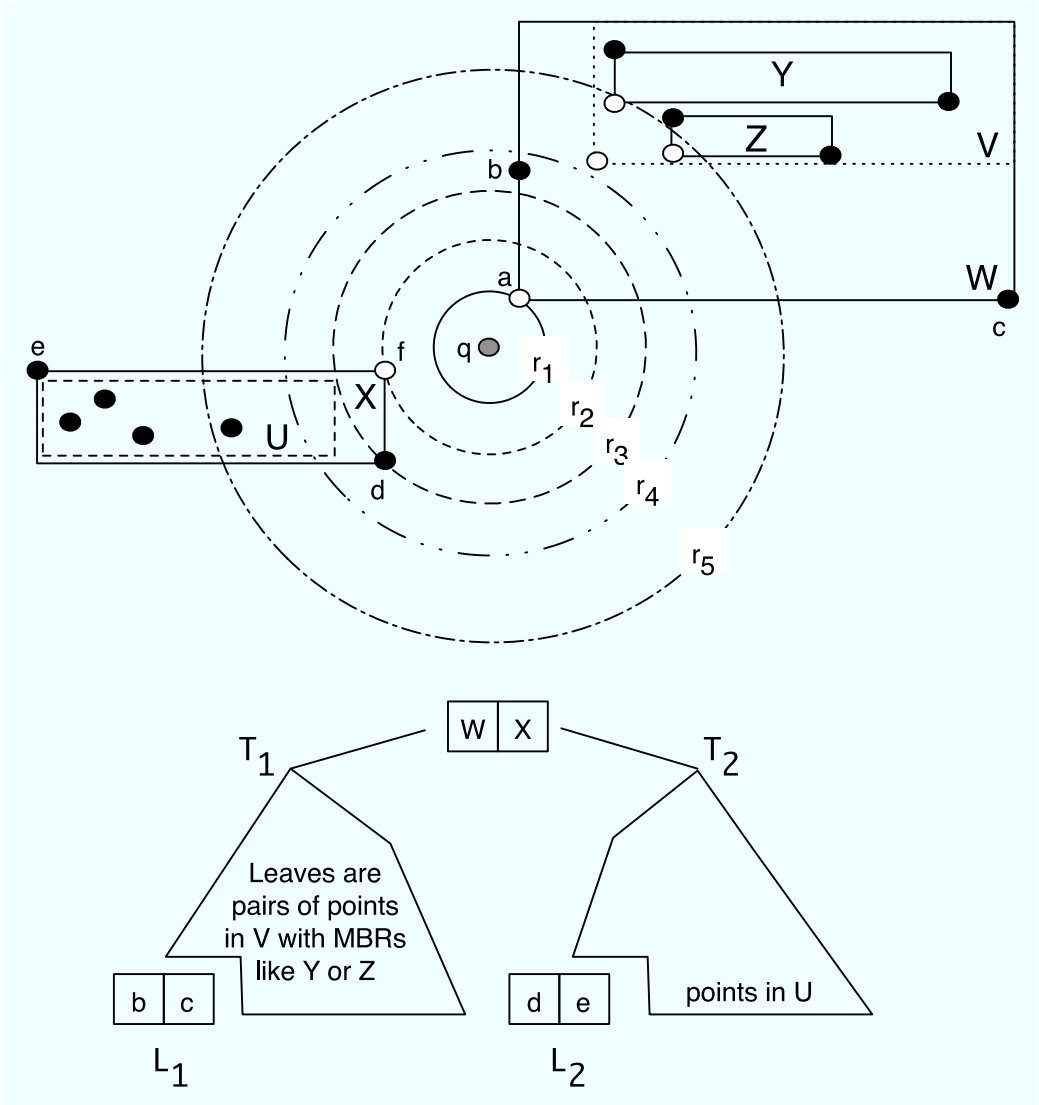

Figure 5.6. A visual explanation of the tree construction used in Theorem 18.

Claim 9. *Given a tree T and query point q as constructed in Figure 5.6, both RKV-PP and RKV prune away all of T_2 save the left branch down to L_2 on a 2-nearest neighbor query.*

Proof. Note that b and d are the two nearest-neighbors to q in T. Both algorithms will search T_1 first because MINDISTANCE$(q, W) <$ MINDISTANCE(q, X). Since both algorithms are depth-first searches, b is in tow by the time T_2 is searched. Because d has yet to be realized, both algorithms will search T_2 and prune away the remaining nodes just as in 8. □

Just as before, we'll show that RKV must examine all the nodes in T_1 while RKV-PP can use information from X to prune away all of T_1 save the left branch down to L_1.

Lemma 18. *Given an R-tree T and query point q as constructed above, RKV examines every node in T_1 in a 2-nearest neighbor search.*

Proof. Since MINDISTANCE$(q, W) = \delta(q, a)$, RKV descends to L_1 first and inserts b (and c) into its 2-best priority queue. However, RKV is unable to prune away any of the remaining leaves of T_1 because every pair of leaf points have MINDISTANCE at most r_5 but all points in T_1 (besides b) lie outside r_5 . As a result RKV must probe every leaf. □

Lemma 19. *Given a tree T and query point q as constructed above, RKV-PP prunes away all nodes in T_1 except those on the branch leading to L_1 in a 2-nearest neighbor search.*

Proof. Before descending into T_1, RKV-PP inserts a promise with distance MINMAXDISTANCE$(q, X) = \delta(q, d)$ into its 2-best priority queue. The algorithm descends into T_1 directly down to L_1, finding b and inserting it into its 2-best priority queue. Unlike RKV, the promise granted by X allows us to prune away

all other nodes of the tree since the remaining nodes are all interior to V and $\delta(q, d) < \text{MINDISTANCE}(q, V)$. □

The theorem follows from Lemma 18 and Lemma 19. Note that if RKV-PP did not remove the promise granted by d at X the final result would be the point d and its promise – an error. □

We can generalize the construction given in Theorem 18 so that the exponential search space reduction holds for any k nearest neighbor query. In particular, for any constant $k > 0$ there exists a class of R-tree and query point pairs for which RKV-PP reduces the search space exponentially. A simple way of accomplishing this is to place $k - 1$ points at b and insert them into the far left leaves of T_1.

CHAPTER 6

CONCLUSION AND DISCUSSION

6.1 Summary

The previous chapters established a rigorous theoretical framework for discussing information organization problems. We focused on two key components of improving access to organized information:

1. Finding good organizational structures; and

2. Improving search procedures on existing structures.

In support of (1) we developed a new model called Constrained Subtree Selection (CSS) for optimally arranging hierarchical data so navigation is natural and popular items are quickly and easily accessible. CSS has applications in web site design and directory structure layout. We also extended the decision tree model to domains where cost is a function of the number of outcomes present at any decision point. We call this extension Category Tree (CT).

Within these models we proved several results. We gave a polynomial time algorithm for instances of CSS where the leaf weights are equal and the degree cost favors trees with degree at most k. We also gave a sufficient condition on the degree cost that makes CSS **NP**-complete even for instances where the leaf weights are equal. Because of this negative result we gave a polynomial time $O(\log(k)\gamma(d+1))$ approximation to CSS where the original hierarchy had maximum degree d. We also gave exact characterizations of the optimal tree for equal leaf weights under both the linear and logarithmic degree cost. Within CT we concentrated on instances where the

degree cost is positive, non-decreasing, the input is binary, and the leaves have equal weight. These instances correspond to the well-studied Decision Tree (DT) problem. We gave a $\ln n + 1$ approximation for DT and showed it does not afford a PTAS unless $\mathbf{P} = \mathbf{NP}$. These results give the first non-trivial upper and lower bounds on the approximation ratio of DT. In addition, they show that DT has a fundamentally different approximation complexity when compared against a similar decision tree problem we call ConDT.

In support of (2) we developed a new pruning technique called Promise-Pruning for depth-first k nearest neighbor queries on R-trees. We constructed a class of tree and query point pairs for which Promise-Pruning reduces the search space exponentially when compared against the same algorithm without the strategy.

6.2 Open Problems

In this section we highlight some attractive open problems for both the Constrained Subtree Selection problem and the Category Tree problem.

6.2.1 The Constrained Subtree Selection Problem

Currently, there is no polynomial time algorithm for constructing an optimal tree for any increasing degree cost when the leaves have equal weight and the graph is constraint-free. There is some evidence that such an algorithm exists: in Section 3.6 we give characterizations of the optimal tree for the linear degree cost and the logarithmic degree cost, which, while varying in structure, suggest a trial-and-test type algorithm to find an optimal tree. Additionally, in the related domain of prefix-free codes, there are near-linear time algorithms for Varn codes—Huffman codes with unequal letter costs, but equal weights. As noted in Chapter 2, instances of CSS where the graph is constraint-free resemble prefix-free coding problems with unequal letter costs. However, the algorithms for Varn codes hinge on properties of lopsided trees

which are not present in our problem — namely that the optimal Varn code is a cut of the infinite lopsided tree. In Varn codes, if the optimal tree has m internal nodes, then those must be the m highest nodes in the infinite tree. If they weren't we could substitute a higher internal node and decrease the cost of the tree. The value of m is constrained by the number of leaves n and the size of the encoding alphabet r so searching for an optimal code means considering different values of m. This approach does not have an analogue in CSS because the lopsided trees are not static; there is no notion of an infinite lopsided tree.

Thinking about the problem in terms of internal nodes, though, has provided some insight. For example, when the degree cost is increasing the optimal CSS tree is fairly *balanced*. Here, balanced means that each path cost does not differ much from the others. For example, if the path cost to a leaf is less than the path cost to any other interior node, then the tree is suboptimal. Moving the subtree rooted at the interior node to this leaf decreases the overall cost. This property is succinctly stated as follows:

Property 2. *No leaf may appear above the level of any other internal node in constraint-free graphs with equal leaf weights and increasing degree costs.*

This means when the degree cost is k-favorable, the leaves must all be within $\gamma(k)$ of one another. Similarly, all parents of leaves must be within $\gamma(k)$ of each other. It's clear that $\gamma(k) \cdot \lceil \log_k(n) \rceil$ is an upper bound on path cost, but a different, perhaps better bound, exists: If h is the level of the minimum level tree T with m leaves where each node has out-degree at most k, and the optimal tree for n leaves has m internal nodes, then those m internal nodes must lie at or below level $h + \gamma(k)$. Suppose this weren't the case. Then there is at least one internal node which lies below $h + \gamma(k)$. But by Proposition 2 all leaves must lie below $h + \gamma(k)$. This yields a contradiction since adding n leaves to T yields a tree where every leaf has path cost at most $h+\gamma(k)$.

It is tempting to imagine that the optimal CSS tree for n leaves with m internal nodes is an extension of the minimum level tree with m leaves. For example, this is the case for the linear degree cost $\gamma(x) = x$. But there is no reason to believe this in general—one can imagine that at times it may be better to increase the path cost of an internal node because there is a corresponding decrease in other internal nodes which may lead to an overall savings in cost. While a general, polynomial time method for finding optimal trees in constraint-free graphs with equal leaf weights eludes us, we have made some small steps which may help the pursuit. For example, if γ is k-favorable and not $(k-1)$-favorable then the unique optimal tree for k leaves of equal weight is the depth-one tree. It is easy to prove this result with the following lemma in hand:

Lemma 20. *If the depth-one tree is optimal for some distribution D then it is also optimal for the uniform probability distribution.*

Proof. Let D be a distribution over n leaves and let C be the cost of the depth-one tree. Suppose the depth-one tree is optimal for D. Since each path has the same cost, the depth-one tree under the uniform probability distribution has the same cost. Now, suppose that this is not an optimal tree for the uniform probability distribution, and instead there is some tree T' with cost $C' < C$. But this means that we can assign D to T' so that the cost is at most C', a contradiction, so the depth-one tree is optimal for the uniform probability distribution. □

Our instinct is that the optimal tree begins with a repeating structure which optimizes some notion of the breadth-to-depth tradeoff; with k-favorable but not $(k-1)$-favorable degree costs, this seems to be the degree k node. Again, this is the case for the linear degree cost, but it may not be true in general.

Almost nothing is known for cases where the DAG is a directed tree and the degree cost is $\omega(\log n)$. We have no hardness results nor polynomial-time algorithms;

even the most simple cases, such as when the graph is a complete binary tree and the degree cost is linear, remain open. In the complete binary tree case, we suspect that the complete binary tree itself is optimal.

6.2.2 The Category Tree Problem

The most prominent open problem concerning CT is the gap between the upper and lower bounds in the approximation ratio of DT instances. Amplifying the gap using techniques from [28] for ConDT does not work. There, one squares an instance of ConDT, applies an α-approximation, and recovers a solution to the original instance which is a $\sqrt{\alpha}$-approximation. Repeating this procedure yields better and better approximations which eventually lead to a PTAS for ConDT which is a contradiction. This does not work for CT because the average path length only doubles when squaring the problem, so solving the squared problem with an α-approximation and recovering a solution to the original problem simply preserves (and unfortunately does not improve) the approximation ratio. The hardness results from [1] rely on the construction of a binary function which is difficult to approximate accurately when certain instances of a hitting-set problem have large solutions. These techniques do not work for CT either. A small step forward toward decreasing the gap is showing that our analysis in the $\ln n$ approximation is tight. Constructing examples where the analysis holds has eluded us to this point.

The second open problem concerning CT involves instances with unequal weights. While our analysis of the approximation algorithm generalizes to tests with unequal weights, it does not work for items with unequal weights. This is because our method does not allow us easily to compare the weight of a node in the greedy tree with the weight of a node in the optimal tree when the weight of a node is determined not by the number of items in its subtree, but by the total weight of the items in the subtree. Overcoming this obstacle presents an interesting analytical challenge.

BIBLIOGRAPHY

[1] Alekhnovich, Misha, Braverman, Mark, Feldman, Vitaly, Klivans, Adam R., and Pitassi, Toniann. Learnability and automatizability. In *Proceedings of the 45th Annual Symposium on Foundations of Computer Science* (2004), IEEE Computer Society Press, Los Alamitos, CA, pp. 621–630.

[2] Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., and Protasi, M. *Complexity and Approximation*, 1st ed. Springer-Verlag, 1999.

[3] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1990), pp. 322–331.

[4] Böhm, Christian, Berchtold, Stefan, and Keim, Daniel A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR) 33*, 3 (2001), 322–373.

[5] Bose, P., Czyzowicz, J., Gasienicz, L., Kranakis, E., Krizanc, D., Pelc, A., and Martin, M. Vargas. Strategies for hotlink assignments. In *Algorithms and Computation, 11th International Conference* (2000), D. T. Lee and Shang-Hua Teng, Eds., vol. 1969 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 23–34.

[6] Bose, Prosenjit, Krizanc, Danny, Langerman, Stefan, and Morin, Pat. Asymmetric communication protocols via hotlink assignments. *Theory of Computing Systems 36*, 6 (2003), 655–661.

[7] Cheung, King Lum, and Fu, Ada Wai-Chee. Enhanced nearest neighbour search on the r-tree. *SIGMOD Record 27*, 3 (1998), 16–21.

[8] Chikalov, Igor. On decision trees with minimal average depth. In *Revised Papers from the Second International Conference on Rough Sets and Current Trends in Computing* (1998), L. Polkowski and A. Skowron, Eds., Springer Verlag, pp. 506–512.

[9] Chikalov, Igor. An algorithm for constructing of decision trees with minimal number of nodes. In *Revised Papers from the Second International Conference on Rough Sets and Current Trends in Computing* (2001), Springer Verlag, pp. 139–143.

[10] Chikalov, Igor. On average depth of decision trees implementing boolean functions. *Fundam. Inf. 50*, 3 (2002), 265–284.

[11] Choi, Siu-Ngan, and Golin, M. Lopsided Trees I: a Combinatorial Analysis. *Algorithmica 31* (2001), 240–290.

[12] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*, 2nd ed. The MIT Press/McGraw-Hill Book Company, 2001.

[13] Crescenzi, Pierluigi, and Kann, Viggo. A Compendium of NP Optimization Problems. http://www.nada.kth.se/~viggo/problemlist/.

[14] Czyzowicz, J., Kranakis, E., Krizanc, D., Pelc, A., and Martin, M. Vargas. Evaluation of hotlink assignment heuristics for improving web access. In *Second International Conference on Internet Computing* (2001), Hamid R. Arabnia and Youngsong Mun, Eds., vol. 2, CSREA Press, pp. 793–799.

[15] Czyzowicz, J., Kranakis, E., Krizanc, D., Pelc, A., and Martin, M. Vargas. Enhancing hyperlink structure for improving web performance. *Journal of Web Engineering 1*, 2 (2003), 93–127.

[16] Ehrenfeucht, Andrzej, and Haussler, David. Learning decision trees from random examples. *Information and Computation 82*, 3 (1989), 231–246.

[17] Faller, N. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers* (1973), pp. 593–597.

[18] Feige, Uriel. A threshold of ln n for approximating set cover. *Journal of the ACM 45*, 4 (1998), 634–652.

[19] Feige, Uriel, Lovász, László, and Tetali, Prasad. Approximating min-sum set cover. *Algorithmica 40*, 4 (September 2004), 219 – 234.

[20] Fleischer, Rudolf. Decision trees: Old and new results. *Information and Computation 152*, 1 (1999), 44–61.

[21] Fuhrmann, Sven, Krumke, Sven Oliver, and Wirth, Hans-Christoph. Multiple hotlink assignment. In *Graph-Theoretic Concepts in Computer Science* (2001), A. Brandstädt and V.B. Le, Eds., vol. 2204, pp. 189–200.

[22] Gallager, R. G. Variations on a theme by Huffman. *IEEE Transactions on Information Theory IT-24*, 6 (1978), 668–674.

[23] Garey, Michael R., and Johnson, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, New York, 1979.

[24] Golin, Mordecai J., Kenyon, Claire, and Young, Neal E. Huffman coding with unequal letter costs. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing* (2002), ACM Press, pp. 785–791.

[25] Golin, Mordecai J., and Rote, Gunter. A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs. *IEEE Transactions on Information Theory 44*, 5 (1998), 1770–1781.

[26] Golin, Mordecai J., and Young, Neal E. Prefix codes: Equiprobable words, unequal letter costs. *SIAM Journal on Computing 25*, 6 (1996), 1281–1292.

[27] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1984), pp. 47–57.

[28] Hancock, Thomas, Jiang, Tao, Li, Ming, and Tromp, John. Lower bounds on learning decision lists and trees. *Information and Computation 126*, 2 (1996), 114–122.

[29] Hjaltason, Gísli R., and Samet, Hanan. Distance browsing in spatial databases. *ACM Transactions on Database Systems 24*, 2 (1999), 265–318.

[30] Hyafil, L., and Rivest, R. Constructing optimal binary decision trees is np-complete. *Information Processing Letters 5*, 1 (1976), 15–17.

[31] Karp, Richard. Minimum-redundancy coding for the discrete noiseless channel. *IRE Transactions on Information Theory IT*, 7 (1961), 27–29.

[32] Knuth, Donald E. Dynamic Huffman coding. *Journal of Algorithms 6*, 2 (1985), 163–180.

[33] Kranakis, Evangelos, Krizanc, Danny, and Martin, Miguel Vargas. The hotlink optimizer. In *Second International Conference on Internet Computing* (2002), Hamid R. Arabnia and Youngsong Mun, Eds., vol. 2, CSREA Press, pp. 87–94.

[34] Kranakis, Evangelos, Krizanc, Danny, and Shende, Sunil. Approximate hotlink assignment. In *12th International Symposium on Algorithms and Computation* (2001), vol. 2223, pp. 756–767.

[35] Kranakis, Evangelos, Krizanc, Danny, and Shende, Sunil. Approximate hotlink assignment. *Information Processing Letters 90* (2004), 121–128.

[36] Moret, Bernard M. E. Decision trees and diagrams. *ACM Comput. Surv. 14*, 4 (1982), 593–623.

[37] Moshkov, Mikhail, and Chikalov, Igor. Bounds on average weighted depth of decision trees. *Fundam. Inf. 31*, 2 (1997), 145–156.

[38] Munagala, Kamesh, Babu, Shivnath, Motwani, Rajeev, and Widom, Jennifer. The pipelined set cover problem. In *ICDT* (2005), pp. 83–98.

[39] Murthy, Kolluru Venkata Sreerama. *On growing better decision trees from data*. PhD thesis, The Johns Hopkins University, 1996.

[40] Perkowitz, Mike, and Etzioni, Oren. Towards adaptive web sites: Conceptual framework and case study. *Artificial Intelligence 118* (2000), 245–275.

[41] Quinlan, J. R. Induction of decision trees. *Mach. Learn. 1*, 1 (1986), 81–106.

[42] Roussopoulos, Nick, Kelley, Stephen, and Vincent, Frédéric. Nearest neighbor queries. In *Proceedings ACM SIGMOD Internaiontal Conference on the Management of Data* (1995), pp. 71–79.

[43] Russell, S. J., and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[44] Sellis, T., Roussopoulos, N., and Faloutsos, C. R+-tree: A dynamic index for multidimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases* (1988), pp. 507–518.

[45] Sieling, Detlef. Minimization of decision trees is hard to approximate. In *18th Annual IEEE Conference on Computational Complexity* (2003), IEEE Computer Society, pp. 82–92.

[46] Vitter, Jeffrey Scott. Design and analysis of dynamic Huffman codes. *Journal of the Association for Computing Machinery 34*, 4 (1987), 825–845.

[47] Vitter, Jeffrey Scott. Algorithm 673: Dynamic Huffman coding. *ACM Transactions on Mathematical Software 15*, 2 (1989), 158–167.

[48] Yao, Andrew Chi-Chih. Decision tree complexity and Betti numbers. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing* (1994), ACM Press, pp. 615–624.

[49] Zantema, H., and Bodlaender, H. L. Finding small equivalent decision trees is hard. *International Journal on Foundations of Computer Science 11*, 2 (2000), 343–354.