RESULTS ON APPROXIMATION ALGORITHMS

George Karakostas

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

November 2000

Abstract

This thesis presents approximation algorithms for three problems: the *Minimum Latency Tour* problem, the *k-Minimum Spanning Tree* problem on graphs and the implementation of the *Perfect-LFU (Least Frequently Used)* policy for Web caching.

The *Minimum Latency Tour* problem, also known as the *traveling repairman problem*, is a variant of the Traveling Salesman Problem. We are given a graph with non-negative edge weights and a starting node and the goal is to compute a tour on the graph that visits all the nodes and minimizes the sum of the *arrival times* at the other nodes. The first part of this thesis presents a *quasipolynomial-time approximation scheme* for this problem when the instance is a weighted tree and when the the nodes lie in the d-dimensional Euclidean space for some fixed d. Our ideas extend to other norms as well as to the case of planar graphs. We also present a polynomial time constant factor approximation algorithm for the general metric case, which achieves a slightly worse approximation factor than the currently best known (due to M.Goemans and J.Kleinberg), but is simpler. Finally we extend the definition of the problem to a more general *weighted* version and show how to apply our ideas in this more general setting.

In the second part we present an approximation algorithm for the problem of finding a minimum tree spanning any k vertices in a graph (k-MST) that achieves an approximation factor of $2 + \epsilon$, for any arbitrary constant $\epsilon > 0$ and runs in time $n^{O(1/\epsilon)}$, improving a 3-approximation algorithm by N.Garg. As in Garg's case, the algorithm extends to a $(2+\epsilon)$-approximation algorithm for the minimum tour that visits any k vertices, provided the edge costs satisfy the triangle inequality.

The last part presents a modification of the Perfect-LFU replacement policy for Web caching called *Window-LFU*. Unlike Perfect-LFU, our policy can be implemented in practice, and under certain assumptions we can prove that it can approximate the hit rate of Perfect-LFU within a factor of $1 - \epsilon$, using space polynomial on the *cache size* instead of

polynomial on the *total number of pages in the Web* (which is the case for Perfect-LFU). In addition to providing analytical bounds for this new policy, we provide experimental results which show that in practice our policy performs better than expected from its analytical study. This leads to a revision of our initial assumptions about the Web. More specifically, our assumption of *statistical independence* among the requests in a request stream does not seem to hold. Instead, there are dependencies due to locality and our policy takes advantage of them.

Acknowledgements

The past five years have been the most exciting period for me. I had to move to a different country, to a different way of life, to a different academic experience than the one I had in Greece. In short, my life has completely changed for the last five years. But all this change alone is not that much exciting in itself. What made it really worthy were the people I met along the way. Without them, this thesis wouldn't exist, so this is the right place for me to say 'thank you'. However, this presents me with a challenge far greater than the issues tackled in the pages that follow: I will have to fit my gratitude in a few lines of text! This is certainly a lost cause to fight, so all I can do is to ask for the understanding of those who are not mentioned here, although they should.

This thesis would not exist without the continuous help and encouragement of my adviser Sanjeev Arora. Usually a graduate student looks for three qualities in an adviser: a respected researcher, a motivating teacher and a good friend. I believe that a graduate student would agree with me if I'd say that an exceptional adviser has any two of these qualities. So I think I am justified to feel extremely lucky to have been associated with someone who has all three of them. Sanjeev has set the mark against which I will measure myself and the others in the future. I really doubt whether an adviser can do anything more for his student.

During my studies at Princeton I benefited from being the student of some of the greatest computer scientists. But I would like to especially thank Dick Lipton. Dick has been a great teacher and a continuous source for inspiration.

Finally, I would like to thank all those people that made life at Princeton so enjoyable. And I will start with those who made it much easier: Sandy, Melissa, Trish, Ginny thanks for everything! Also I would like to thank all my friends and fellow graduate students in the CS Department, and especially Tasos and Kostas. Among other achievements, together we helped Starbucks to open a few new branches. The Greek community of Princeton

University helped me to keep in touch with the tiny place I come from. Thanks to Dimitris, Aggelos, Evi, Andromache, Kuriakos, Christos and the rest of the 'old guard' for five years of fine dinning and movie going experiences. Also many thanks to the newer members of the Greek community. I will not refer to them individually (they know who they are, after all), but I expect them to mention me when they finish their thesis. And of course many thanks to Dimitris and Andreas for sharing with me their wit and experience, just as any good (former) soldier would.

It is customary for the author of a thesis to dedicate it to his/her parents. I am not going to do so, though. Everything I did, I do or I will ever do are due to my father and mother. A simple dedication does not come even close to a 'thank you for everything'

Funding acknowledgement: The research this thesis is based on was supported by Sanjeev Arora's NSF CAREER award NSF-CCR 9502747, an Alfred Sloan Fellowship, and a Packard Fellowship.

Contents

List of Figures

Chapter 1

Introduction

1.1 Approximation algorithms

One of the central questions in Computational Complexity is whether $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$. Although great effort has been put into tackling this question for decades, we are still far from answering it. In the meanwhile an ever increasing list of problems have been proven to be NP-hard. The existence of a polynomial time algorithm for any of these problems would immediately imply $\mathcal{P} = \mathcal{NP}$. Given our inability to prove the latter, it is highly unlikely that such algorithms will appear in the near future (if they exist, of course). So researchers have turned their attention into devising *approximation* instead of *exact* algorithms. These are algorithms that produce a solution *guaranteed* to be within a factor from the optimal solution efficiently (i.e. more efficiently than the (usually exponential) time needed to solve the problem exactly). The measure of the quality of an approximation algorithm is the quality of the approximation (i.e. the approximation achieved) and its running time (although usually when one refers to approximation algorithms he or she refers to a *polynomial* time algorithm, and hence running time usually is not a criterion for the quality of the algorithm). To make the lines above more precise we give the definition of an approximation algorithm:

Definition 1: *An approximation algorithm for a* minimization *problem achieves an approximation factor δ iff for any instance $\mathcal{I}$ of the problem, the following holds for the optimal solution $OPT(\mathcal{I})$ and the solution $SOL(\mathcal{I})$ produced by the algorithm:*

$$SOL(\mathcal{I}) \leq \delta \times \mathcal{OPT}(\mathcal{I})$$

Obviously, in this case $\delta > 1$. For a maximization *problem achieving an approximation factor of δ the following must hold:*

$$SOL(\mathcal{I}) \geq \delta \times \mathcal{OPT}(\mathcal{I})$$

and $\delta < 1$.

An algorithm achieving an approximation factor of δ is called a *δ-approximation algorithm*. The better (i.e. closer to 1) the δ of an algorithm is, the better the quality of the algorithm is. The best one can hope for is an approximation factor of $1 + \epsilon$ (for a minimization problem) or $1 - \epsilon$ (for a maximization problem), for *any constant* $0 < \epsilon < 1$[1]. Then the algorithm is called an *approximation scheme*. Of course the running time of the algorithm will depend on ϵ, so there is a trade-off between the quality of the solution and the efficiency of the algorithm. This trade-off is refined further in the following definitions:

Definition 2: *A family of approximation algorithms for a problem $\mathcal{P}$, $\{\mathcal{A}_\epsilon\}$, is called a* polynomial time approximation scheme (PTAS) *if algorithm $\mathcal{A}_\epsilon$ is an $(1+\epsilon)$-approximation algorithm and runs in time polynomial in the size of the input for a fixed ϵ.*

In this thesis we present approximation schemes for the Minimum Latency Tour problem (defined later) that run in quasi-polynomial time (instead of polynomial). In this case we talk about *quasi-polynomial time approximation schemes (QPTAS)*:

Definition 3: *A family of approximation algorithms for a problem $\mathcal{P}$, $\{\mathcal{A}_\epsilon\}$, is called a*

[1] Of course for an exact algorithm $\delta = 1$.

quasi-polynomial time approximation scheme (QPTAS) *if algorithm $\mathcal{A}_\epsilon$ is an $(1+\epsilon)$-approximation algorithm and runs in time quasi-polynomial in the size of the input for a fixed ϵ.*

Often it is possible to prove that a problem cannot have approximation schemes unless a Computational Complexity hypothesis holds (e.g. $\mathcal{P} = \mathcal{NP}$). The area of inapproximability is another area that has flourished during the last decade due to a wealth of breakthrough results like those in [11]. A class of problems that do not have PTAS's unless $\mathcal{P} = \mathcal{NP}$ is the class MAX-$\mathcal{SNP}$. The definition of this class and the notion of L-reductions (used to define completeness for it) can be found in [37].

For an introduction to the field of approximation algorithms the reader is referred to the textbook by C. Papadimitriou [36] and the excellent collection of surveys edited by D. Hochbaum [28]. More on hardness of approximation results can be found in [32] and the references in it.

1.2 Outline

This thesis presents approximation algorithms for two NP-hard problems: the Minimum Latency Tour (MLT) and the k-Minimum Spanning Tree (k-MST). In the first part we describe quasi-polynomial approximation schemes for some cases (e.g. trees or planar graphs) of the MLT and some of its extensions. In the second part we describe an approximation algorithm for the k-MST that achieves an approximation factor of $2+\epsilon$ for any $\epsilon > 0$ and runs in polynomial time for *fixed* ϵ. Most of the material in the first two parts is based on [9] and [10].

In the third part of this thesis we study the approximation of the Least Frequently Used (LFU) replacement policy for Web caching. Under certain assumptions we prove that the hit rate of Perfect-LFU can be approximated to within any constant $\epsilon > 0$ using space polynomial on the size of the *cache* instead of polynomial on the size of *Web sites*. Our experiments show that (a) certain of our assumptions do *not* seem to hold and (b) our proposed Web caching policy takes advantage of the fact that these assumptions do not hold and performs *better* than the theoretical analysis indicates. The material in this part is based on [41] and [29].

Chapter 2

The Minimum Latency Tour problem

The *minimum latency problem*, also known as *traveling repairman problem* [1], is a variant of the Traveling Salesman Problem (TSP) in which the starting node of the tour is given and the goal is to minimize the sum of the arrival times or *latencies* at the other nodes. (The *latency* of a node is the distance covered before reaching that node.) This natural combinatorial problem arises in many day-to-day situations, whenever a server (e.g. a repairman or a disk head) has to accommodate a set of requests (each represented by a point) so as to minimize their total (or average) waiting time. The tour that achieves this goal will henceforth be call the *minimum latency tour* (MLT for short).

More formally, the problem is defined as follows:

MINIMUM LATENCY TOUR

Input : A set of n points (one of them designated as the starting point p_1), a symmetric distance matrix $[d_{ij}]$

Output : A tour $p_1 \to p_2 \to \ldots \to p_n$ that visits *all* points and minimizes the total latency

$$\sum_{i=2}^{n} \sum_{j=1}^{i-1} d_{p_j, p_{j+1}} \tag{2.1}$$

By rearrangement, the objective function (2.1) can be rewritten as a weighted sum:

$$\sum_{i=1}^{n-1} (n-i) d_{p_i, p_{i+1}} \tag{2.2}$$

Here we study the restriction of the problem to distance matrices that satisfy the following conditions:

$$\begin{aligned} d_{ii} &= 0 \quad \forall i \\ d_{ij} &= d_{ji} \quad \forall i, j \\ d_{ij} &\leq d_{ik} + d_{kj} \quad \forall i, j, k \end{aligned} \tag{2.3}$$

i.e. the distances define a *metric*.

2.1 Comparison with the TSP

From the objective function formulation (2.2) it becomes apparent that the MLT problem is a weighted variation of the TSP (where the objective function to be minimized by the tour is $\sum_{i=1}^{n-1} d_{p_i, p_{i+1}}$) . However, it has a reputation for being much harder than the TSP.

For example, the MLT does not possess the locality property of the TSP for local changes in the structure of a metric space. Even a very small local change of a tour can affect the arrival time of *all* the points visited by the tour afterwards, and it may change dramatically the value of (2.2). On the other hand, local changes affect the *length* of the tour only locally. This lack of locality is conceivably prohibitive for the design of algorithms that 'divide and conquer' in order to solve the MLT problem: each one of the subproblems is profoundly related to the other subproblems and their optimal solution does *not* imply an optimal overall solution. This is not the case, for example, for the Euclidean TSP where resent 'divide and conquer' algorithms by Arora [7] (and, independently, Mitchell [34]) solve the problem almost optimally.

It is also easy to prove (Blum et al. [16]) that calculating the MLT is at least as hard as calculating the TSP: Given an instance of the TSP, create an instance for the MLT by adding m new points (for a large m) at infinity (i.e. far enough away from the points of the TSP instance) as in Fig. 1. Then the MLT on the augmented set of points will have to follow the TSP tour on the original points before visiting the new ones.

This reduction proves the NP-hardness of the MLT for all the metric spaces where TSP is NP-hard. But even for metrics where the TSP is solvable trivially, the computation and the structure of the MLT seems to be much more complex. The case of weighted trees is an illustrative example. A tree with non-negative weights on its edges defines a metric very simply as follows: the points are the vertices of the tree and the distance between any two points is the total weight of the unique path that connects these points. Such a tree is shown in Figure 2.1. In any tree, the TSP has a very simple structure; it is just a depth-first walk on the tree. For example, a TSP tour on the tree of Figure 2.1 is the walk $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 1$. But the MLT for this tree is much more complicated: $1 \to 2 \to 3 \to 4 \to 7 \to 8 \to 5 \to 9 \to 6 \to 1$. In the case of points on the Euclidean plane, there is always a TSP tour that is not self-crossing, while the MLT may cross itself many times.

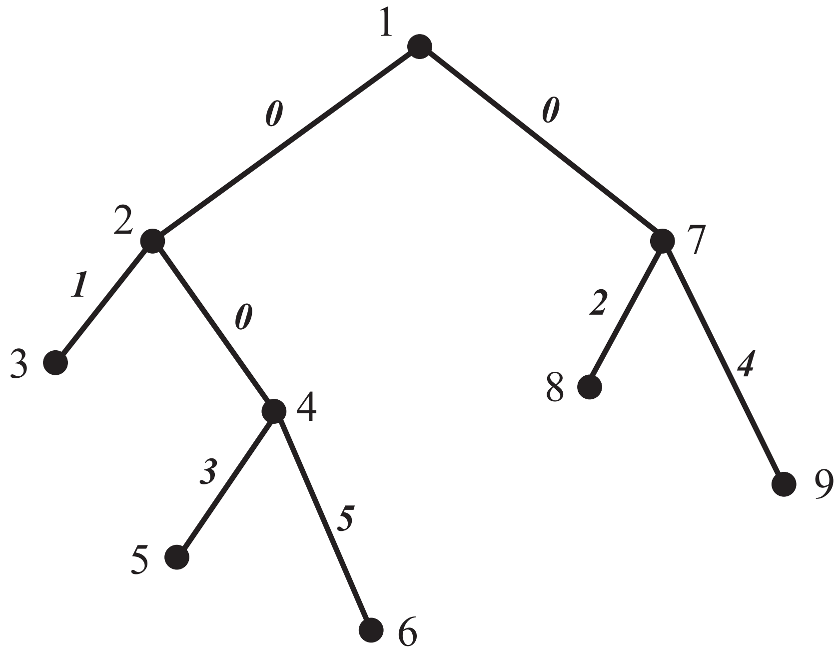


Figure 2.1: A weighted tree

2.2 History

The MLT problem has appeared in the literature under the names of the *deliveryman* or *traveling repairman* problem [33, 31, 15]. [1] gives a simple dynamic programming algorithm that solves the problem when the points are on a line in polynomial time. [33, 16] prove that in the case of *unweighted* trees (trees whose edges have unit length), any depth-first walk on the tree is an optimal MLT. [33] gives also an exponential time algorithm that solves the problem exactly for any weighted tree. For general metric spaces, [15, 31] give exponential time algorithms that find a MLT.

The apparent difficulty of the MLT problem even for simple cases like trees has driven the research towards approximation algorithms. The general metric case of the latency problem is MAX-SNP-hard (this follows from the reduction that proves the MAX-SNP-hardness of TSP with distances either 1 or 2 [38]) and therefore the results of Arora et al. [11] imply that unless P = NP, a polynomial time approximation scheme (PTAS) is unlikely to exist. Blum, Chalasani, Coppersmith, Pulleyblank, Raghavan and Sudan gave a 144-approximation algorithm for the metric case and a 8-approximation for weighted trees. Goemans and Kleinberg [25] then gave a 21.55-approximation in the metric case. The Goemans-Kleinberg algorithm requires as a subroutine a good approximation algorithm for the k-TSP problem ("given n nodes and a number k, find the shortest salesman tour containing k nodes"). Their algorithm achieves a 3.59 approximation ratio for the tree case and a $3.59 \times$ (*appr. ratio for k-MST*) approximation ratio for general metric spaces. Currently the best approximation ratio for the general metric k-MST is $2+\epsilon$ for any constant $\epsilon > 0$ due to Arora and Karakostas ([10] and also Chapters 6, 7 of this thesis) and $1+\epsilon$ for any $\epsilon > 0$ for constant-dimensional Euclidean spaces due to Arora [7] (for planar instances, see also Mitchell [34]). These can be used to improve the approximation ratio achieved by the Goemans-Kleinberg algorithm in these cases, achieving a 7.18-approximation in the metric case and 3.59-approximation in the Euclidean case.

This thesis presents a new and very simple technique that leads to *approximation schemes* for minimum latency for all weighted trees and constant-dimensional Euclidean spaces. To compute a $(1+\epsilon)$-approximation for the problem on n nodes, the algorithm requires $n^{O(\log n/\epsilon)}$ time on weighted trees and $n^{O(\log n \log\log n/\epsilon^2)}$ time in $\mathbb{R}^2$. We also present a 17.24-approximation in the metric case. Though this approximation ratio is worse than that of the algorithm in [25], our algorithm seems simpler.

Previous papers have taken the approach of computing solutions to a sequence of k-TSP instances, for $k = 1, 2, \ldots, n$, and concatenating some of these tours to get the final tour. The main intuition in these algorithms may be described as: visit the nodes closest to the start node as soon as possible. The main idea in our algorithm is easier to state. The algorithm finds the tour as a union of $O(\log n/\epsilon)$ tours, containing $n_1, n_2, \ldots$ nodes. The important difference is that the choice of $n_1, n_2, \ldots,$ does not depend on the instance; it depends only on n, ϵ. (Thus, as noted in Section 3.2, our algorithm can be viewed as an approximation-preserving reduction to a version of vehicle routing.) In fact, in the metric case, the first salesman tour contains more than $n/2$ nodes. This seems to go against the received intuition of "visit the nodes closest to the start node first." Conceivably, our technique could be combined with that intuition to get better algorithms, but we do not currently know how.

2.3 Search ratio of graphs.

Koutsoupias, Papadimitriou, and Yannakakis[30] consider a graph exploration problem, in which an explorer is presented with a weighted graph on n nodes. One of the nodes contains a treasure, which the explorer will recognize only when he sees it. The goal is to design a walk on the graph such that the explorer arrives at the treasure as quickly as possible. The *search ratio* of the graph is the worst-case ratio of the arrival time and the distance of the treasure to the start node. The *randomized search ratio* is defined similarly, except the walk may be randomized and so we need the *expected* arrival time at the treasure.

More formally, the two problems are defined as follows:

SEARCH RATIO

Input : A Graph G with distances on edges, and a root vertex r.

Output :

$$\sigma(G,r) = \min_{\pi} \max_{v \in G} \frac{d_\pi(r,v)}{d(r,v)},$$

where $d(r,v)$ denotes the distance from r to v and $d_\pi(r,v)$ denotes the distance from r to v in the walk π.

RANDOMIZED SEARCH RATIO

Input : A Graph G with distances on edges, and a root vertex r.

Output :

$$\rho(G, r) = \min_{\Delta} \max_{v \in G} \frac{E_{\Delta}[d_{\pi}(r, v)]}{d(r, v)},$$

where Δ ranges over *distributions of walks* and $E_{\Delta}[d_{\pi}(r, v)]$ denotes the *expected* distance from r to v in the walk π, when π is drawn randomly according to distribution Δ.

As shown in [30], computing the search ratio and the randomized search ratio of a graph G with respect to a root node r is NP-complete and MAXSNP-hard. They also show that the minimum latency problem is the polyhedral separation problem of the dual of the randomized search ratio problem. Indeed, the randomized search ratio can be expressed as the solution of the following $(n + 1) \times n!$ linear program:

$$\begin{aligned}
&\min\ \rho\ \text{ s.t.} \\
&\textstyle\sum_{\text{walk } \pi} x_{\pi} d_{\pi}(r, v) \leq \rho \cdot d(r, v), \quad \forall v \in V \\
&\textstyle\sum_{\text{walk } \pi} x_{\pi} = 1 \\
&x_{\pi} \geq 0
\end{aligned}$$

Its dual is

$$\begin{aligned}
&\min \textstyle\sum_{v} d(r, v) y_v - z \ \text{ s.t.} \\
&\textstyle\sum_{v} d_{\pi}(r, v) y_v - z \geq 0, \quad \forall \text{ walk } \pi \\
&y_v \geq 0
\end{aligned}$$

To solve the dual by using the ellipsoid algorithm using the general framework of [27], we should be able to check whether a given point $(\vec{y}, z) \in \mathbb{R}^{n+1}$ lies is a feasible solution or not. In case it is *not* feasible, we need a violated inequality, i.e. we should be able to decide whether $\min_{\pi} \sum_{v} d_{\pi}(r, v) y_v \leq z$. [30] observe that this decision problem is *polynomially equivalent* to the MLT problem, under some very mild restrictions that do not affect approximability.

Using the general framework for convex optimization in Grötschel, Lovász, and Schrijver [27], É. Tardos has observed [30] that if the minimum latency problem has a polynomial time approximation scheme for a certain class of metrics, then the randomized search ratio problem has an approximation scheme for that same class of metrics. Thus our algorithm implies the existence of a quasipolynomial time approximation scheme for the randomized search ratio for trees and Euclidean spaces. We do not know of a previous use of the framework in [27] to design an approximation scheme.

Chapter 3

The main idea: Local structure does not matter

It is well-known that minimizing the total tour length may give a tour of very high latency. In this chapter we show that the strategy of minimizing tour lengths works so long as it is done in a local fashion. Namely, to find a $(1+\epsilon)$-approximate minimum latency tour, it suffices to find the tour as a union of $O(\log n/\epsilon)$ segments, where the number of nodes in successive segments decreases geometrically. Within each segment the order of visits to the nodes does not matter, as long as the total length is close to minimum. Of course, we have not specified thus far how to partition nodes into the segments in the correct way. In the Euclidean and tree-metric cases, we can do this with a simple dynamic programming. In the metric case, we recourse to a greedy strategy that introduces another source of suboptimality.

We note that the idea of finding a low latency tour as a union of salesman tours/paths is present in all earlier papers [16, 25, 30]. However, those earlier strategies decided in an adaptive fashion which salesman tours to combine. In contrast, our algorithm can decide at the very start how many nodes must be present in each salesman path.

3.1 The main idea

We prove that we can break an MLT tour into segments so that local changes within a segment doesn't affect the total latency by much, and then replace each segment by an optimum salesman path, so that the new tour is still near optimal.

Let $\mathcal{T}$ be an optimal tour with total latency OPT. Let $\epsilon > 0$ be any parameter. Break this tour into k segments, so that in segment i we visit n_i nodes, where

$$\begin{aligned} n_i &= \left\lceil (1+\epsilon)^{k-1-i} \right\rceil \text{ for } i = 1 \ldots k-1 \\ n_k &= \lceil 1/\epsilon \rceil . \end{aligned}$$

To simplify the calculations we will assume that ceilings are not needed in the expressions above. This assumption will not affect our results. Let the length of the i^{th} segment be T_i. If we let $n_{>i}$ denote the total number of nodes visited in segments numbered $i+1$ and later, then a simple calculation shows that (and this was the reason for our choice of n_i's)

$$n_{>i} = \sum_{j>i} n_j \leq \frac{n_i}{\epsilon}, \text{ for every } i = 1 \ldots k-1 \tag{3.1}$$

Now imagine doing the following in each segment except the last one: replace that segment by the minimum-cost traveling salesman path on the same subset of nodes, while maintaining the starting and ending points. We claim that the new latency is at most $(1+\epsilon)$OPT. First, note that $\sum_{j=1}^{m-1} T_j$ is a lower bound on the latency of any node in the m^{th} segment. Adding over all segments, we get the following lower bound on OPT:

$$\text{OPT} \geq \sum_{i=1}^{k} n_{>i} \cdot T_i. \tag{3.2}$$

Consider the effect of replacing the ith segment with the shortest salesman path on that subset of nodes. The length of the segment cannot increase, and so neither can the latency of nodes in later segments. The latency of nodes within the segment can only rise by $n_i T_i$.

Thus the new latency is at most the lower bound in (3.2) plus

$$\sum_{i=1}^{k-1} n_i \cdot T_i. \tag{3.3}$$

Now condition (3.1) implies that the new latency is at most $(1+\epsilon)$OPT, as claimed, and the proof of the main idea is complete:

Theorem 1: *Let OPT be the total latency of the MLT. There exists a tour that is a concatenation of $O(\frac{\log n}{\epsilon})$ optimal salesman paths, and whose total latency is at most $(1+\epsilon)OPT$.*

The importance of this structure theorem for the design of approximation schemes is apparent for cases where the TSP is efficiently solvable (e.g. weighted trees). All one has to do is to decide which node goes into which segment and then calculate the optimal salesman path for each segment. Even if, instead of the optimum salesman path in each segment, we use a $(1+\gamma)$-approximate salesman path, then the latency of the tour of Theorem 1 is $(1+\gamma\cdot\epsilon+\gamma+\epsilon)$OPT. This is the case, for example, for Euclidean spaces of fixed dimension, where there are approximation schemes for the TSP (cf. [7]). For these cases the difficulty lies in the placement of nodes into the appropriate segments. This has the flavor of a vehicle routing problem, and this will become more precise in the following section.

3.2 Reduction from Minimum Latency to Weighted Vehicle Routing

The purpose of this section is to note that our technique described above implies a quasipolynomial-time approximation-preserving reduction from Minimum Latency to a version of Weighted Vehicle Routing with *per-mile costs.* We do not know of a prior result along these lines.

WEIGHTED VEHICLE ROUTING WITH PER-MILE COSTS

Input : A set of n clients, who have to be visited by a fleet of m vehicles. Vehicle i has a designated depot s_i to start from and another depot t_i to finish at. It also has a *capacity* (which is the number of clients it can visit) c_i and a *per-mile cost* d_i.

Output : Assign clients to vehicles so as to respect the capacity constraints and minimize the total cost, which is the sum of distances covered by the vehicles, weighted by the per-mile cost

$$\sum_{i=1}^{m} d_i T_i$$

where T_i is the distance traveled by vehicle i.

Suppose we are given an oracle for this version of vehicle routing. We can use this oracle to solve the MLT problem. Given a set of n nodes, the reduction proceeds as follows, where $k, n_i, n_{>i}$ have the same meaning as in section 3.1: "Let p_0 denote the starting node of the minimum latency tour. For every sequence of k nodes $p_1, p_2, \ldots, p_k$, use the Vehicle Routing Oracle to construct a solution to the instance in which there are k vehicles, and the capacity of the ith vehicle is n_i, its per-mile cost is $n_{>i}$, and its starting and end depots are p_i and p_{i+1} respectively. At the end, output the lowest cost solution found (over the choice of all sequences of k points)."

Clearly, if the vehicle routing oracle computes a ρ-approximation in polynomial time, our reduction will lead to a $\rho(1+\epsilon)$-approximate solution for MLT in $n^{O(\log n/\epsilon)}$ time. This follows from Theorem 1, since it suffices to go over all possible sequences $p_1, p_2, \ldots, p_k$, and this increases the running time by a factor of at most $O(n^k) = n^{O(\frac{\log n}{\epsilon})}$.

Chapter 4

Approximation algorithms for the MLT

The essence of Theorem 1 is that there is a near optimal latency tour of a potentially simpler structure, since it is the collection of just a few optimal salesman paths. We take advantage of this simpler structure in order to compute a near optimal tour for the cases of trees, Euclidean (and other norm) spaces, and planar graphs. The algorithms for these cases are quasi-polynomial time *approximation schemes.* We also give a constant factor polynomial time approximation algorithm for the general metric case as an immediate result of Theorem 1. Although its approximation factor is somewhat worse than the approximation factor achieved in [25], it seems simpler.

4.1 The Tree Case

The optimum salesman tour on a tree may in general need to visit a node unbounded number of times. For example in a star graph, it must visit the center node $n-1$ times. However, the tour never needs to visit an edge more than twice, as is easily checked. Thus the optimum tour has very simple structure and can be found by depth-first search.

A minimum latency tour, on the other hand, could have a very complicated structure.

Consider for example a complete binary tree in which all edges have zero weight except those attached to leaves. The minimum latency tour will first visit all the internal nodes in some arbitrary order, and then all the leaf nodes in sorted order by weight, thus crossing the root node $n/2$ times.

However, our technique from Chapter 3 allows us to view a near-optimum latency tour as a union of $O(\log n/\epsilon)$ salesman paths. By observing that within each salesman path an edge is only visited twice, we can then ensure that each edge is only visited $O(\log n/\epsilon)$ times overall. This idea underlies the proof of our *Structure Theorem* below and the approximation scheme for trees.

4.1.1 The Structure theorem for trees

In this subsection we prove that there is a tour on a weighted tree whose latency is near optimal but crosses each node of the tree only a few times.

Before we define what we mean by 'crossing', we define the notion of an '$\alpha : \beta$-partition' of a tree.

Definition 4: *An $\alpha : \beta$-partition of a tree $\mathcal{T}$ with n nodes is the recursive partition of $\mathcal{T}$ into two subtrees with a common root, so that for each subtree*

$$\alpha\, n \leq \textit{(size of subtree)} \leq \beta\, n.$$

Since the partition is recursive, every node of the tree is going to become a separator at some level of the recursion.

It is not obvious that we can recursively partition any tree in this fashion for *any* pair α, β (actually we can't!). But it turns out that for *any* tree we can find a $\frac{1}{3} : \frac{2}{3}$-partition. In other words, we can easily prove the following:

Lemma 1: *In any tree we can always find a node with the following property: Let $m \geq 2$ denote its degree and $f_1, \ldots, f_m$ denote the sizes of the subtrees attached to the node. There* *exists a subset* $S \subseteq \{1, \ldots, m\}$ *such that*

$$\lfloor \frac{n}{3} \rfloor \leq \sum_{i \in S} f_i \leq \lceil \frac{2n}{3} \rceil.$$

Proof: Start by picking a node of the tree. If there is a subtree with more than $\lceil \frac{2n}{3} \rceil$ nodes pick the root of this subtree and continue. If there is a subtree with number of nodes between $\lfloor \frac{n}{3} \rfloor$ and $\lceil \frac{2n}{3} \rceil$ then the current node satisfies the lemma requirements. If all subtrees have less than $\lfloor \frac{n}{3} \rfloor$ nodes (obviously $m \geq 3$ in this case) it is easy to see that the lemma holds for the current node.

□

We designate this node as a *separator node* and the $|S|$ components as the *left* of the tree and the other $m - |S|$ as the *right* of the tree. (The node itself is copied twice and appears in both sides.) Then we recur on the two sides. This gives a recursive partition of the tree. We say that a tour *crosses* the separator node if it goes from the left side to the right. Notice that a node may be visited many times before it is crossed.

It is obvious that since a minimum salesman tour is obtained by depth-first-search, it needs to cross each separator node only twice. The next theorem says that there is a tour whose latency is within a factor $(1+\epsilon)$ of the minimum latency and needs to cross each separator only $O(\log n/\epsilon)$ times, for any $\epsilon > 0$.

Theorem 2: (Structure theorem for Weighted Trees) *The following is true for every integer* $n > 0$ *and every* $\epsilon > 0$*: For every weighted tree on* n *nodes with a node-separator based partition as defined above, a tour exists with latency at most* $(1+\epsilon)OPT$*, that crosses each separator node only* $O(\log n/\epsilon)$ *times.*

Proof: Let $\mathcal{T}$ be the optimum tour. Divide it into $O(\log n/\epsilon)$ segments as in Chapter 3 and replace each segment by the optimum salesman path. Now use the fact that a minimum salesman path does not cross a separator node going from the left side to the right (or vice versa) of the partition at that node more than twice. □

4.1.2 The algorithm for trees

A simple dynamic programming approach that relies on the structure Theorem 2 can now be used to compute a $(1+\epsilon)$-approximate latency tour for general weighted trees.

ALGORITHM 1

1. Identify a recursive $\frac{1}{3} : \frac{2}{3}$-partition of the tree.

   Do the following in a "bottom-up" fashion starting from the bottom level of the partition:

2. Identify a separator at the current level of the partition.

3. "Guess" the number of times the tour crosses this node, and for each crossing, the length of the tour portion after it and the number of nodes on that portion.

4. Search the dynamic programming look-up table for subtours consistent with the "guesses".

5. Combine the subtours found to create a new bigger subtour. Store this new subtour in the look-up table and go to step 2.

□

Of course, by "guessing" in Step 3 we refer to exhaustive enumeration of all possible values for the triple (# of crossings, length, # of nodes). In the end of the enumeration, the algorithm will have created a collection of candidate solutions (one for each set of values for our "guesses"). Its output will be the tour with the minimum total latency. One of these tours calculated by the algorithm must be the near optimal tour guaranteed by the Structure theorem 2, since the algorithm is bound to encounter the specific set of "guesses" defined by it due to exhaustive enumeration. Hence the tour it produces is at least as good as this tour and the algorithm is indeed an *approximation scheme.* The only thing remaining to prove is that it is a *quasi-polynomial* approximation scheme.

The running time of the algorithm is obviously dominated by the number of possible "guesses". The number of crossings through a node is at most $O(\log n/\epsilon)$ (Theorem 2) and the number of nodes visited between two crossings cannot be greater than n. But notice that the length of the tour between two crossings can be exponential on the size of the input. This is the case when an edge has weight exponential on the size of the input. Then the number of guesses is also exponential and the algorithm runs in exponential time, instead of quasi-polynomial. In order to get around this problem, we *round* the given instance by running the following **rounding procedure**:

1. Let L be the length of the longest path in the tree and $\delta > 0$ any constant smaller than ϵ. Merge (by contracting edges) all pairs of nodes with internode distance at most $\delta L/n^2$.

2. Round each edge weight to its closest multiple of $\delta L/n^2$.

3. Divide all edge weights by $\delta L/n^2$.

After solving the problem on the rounded instance, we reinstate the merged edges to output the tour computed on the original instance.

Lemma 2: *Let OPT be the length of the MLT in the given instance and L, δ are as in the above procedure. Then the MLT on the rounded instance and OPT differ by at most $O(\delta\ OPT)$. Moreover, its maximum internode distance is at most $O(n^2/\delta) = O(n^2/\epsilon)$.*

Proof: The second part of the lemma is obvious. Also, it is easy to see that the latency of each node in the original MLT has not changed by more than $O(\delta L/n)$, for a total change of $O(\delta L) = O(\delta \mathrm{OPT})$. Notice that when we reinstate the merged edges, the latency increase cannot be more that $O(\delta\text{OPT})$. This means that we need to compute an $(1+\epsilon-\delta)$-approximation tour on the rounded instance instead of a $(1+\epsilon)$-approximation. But we certainly can do that since δ is an arbitrary constant.

□

Thus if we run Algorithm 1 on the new rounded instance, the running time is quasi-polynomial:

Theorem 3: *Algorithm 1 runs in time* $n^{O(\log n/\epsilon)}$ *and computes a tour whose latency is at most* $(1+\epsilon)$ *OPT*.

Proof: If δ is the constant in the rounding procedure then we apply Theorem 2 so that the approximation factor for the rounded instance is $1+\epsilon-\delta$. Because of Lemma 2 the approximation factor for the tour we compute in the original instance will be $(1+\epsilon)$.

The dynamic programming algorithm builds a look-up table in a bottom-up fashion (starting from the bottom level of the $\frac{1}{3}:\frac{2}{3}$-partition of the tree towards the top). For each separator considered by the algorithm, we guess the number of crossings and for each crossing the length and the number of vertices visited. So the total number of guesses for each crossing is $O(\frac{n^2}{\epsilon}\cdot n) = O(\frac{n^3}{\epsilon})$ (because of Lemma 2), for a total of $O(\frac{\log n}{\epsilon}\cdot(\frac{n^3}{\epsilon})^{O(\log n/\epsilon)}) = n^{O(\log n/\epsilon)}$ guesses for a node. The costs of subtours consistent with each one of these guesses are already in the look-up table, so we can look them up in constant time. So the overall running time of Algorithm 1 when run on a rounded instance is $O(n\cdot n^{O(\log n/\epsilon)}) = n^{O(\log n/\epsilon)}$.

The reader probably notices that the look-up table stores *costs* instead of subtours, so at the end the algorithm has computed the *cost* of a near-optimal tour and not the tour itself. But it is easy to reconstruct this tour from the look-up table and the decision made at each step of the dynamic programming.

□

4.2 The Euclidean Space

In Chapter 3, we reduced the minimum latency problem to the problem of finding a covering of the n nodes using $O(\log n/\epsilon)$ salesman paths. Now we use a simple modification of Arora's [7] ideas for the Euclidean TSP to show that this set of $O(\log n/\epsilon)$ salesman paths together have a very simple structure. Thus they can be computed by dynamic programming in quasi-polynomial time in a way similar to the tree case. We will start by describing the algorithm for the Euclidean plane. The generalization to spaces of higher dimension will follow. Our exposition follows the exposition in [7].

4.2.1 The partition

In order to apply dynamic programming techniques we need to recursively partition the instance into smaller subinstances. We managed to do this in the tree case by exploiting the existence of an $\frac{1}{3}:\frac{2}{3}$-partition. Here we describe a very simple *geometric* partition based on the well known quadtree. In what follows we assume that all coordinates of the given points are integral. Later we will show how to "perturb" the instance so that this assumption is fulfilled without increasing the cost of our solution by too much.

Suppose that the given instance is bounded by a square and let L be the size of the smallest axis-aligned bounding square. Then the *dissection* of the bounding square is its recursive partition into smaller squares in the obvious way: break the bounding square into 4 equal squares, then break each smaller square into 4 equal squares and so on until the smallest squares have side size ≤ 1 (and thus cannot contain more than 1 point). This partition defines a tree: the root is the bounding square and the nodes of the tree at each level are the squares created at the same level of the dissection. Each square has 4 children. Obviously the tree has $O(L^2)$ nodes and its depth is $O(\log L)$. If we stop partitioning a square as soon as it contains at most one node, then this truncated version of the dissection tree is called a *quadtree*. Examples of a dissection and a quadtree are shown in Figure 4.1.

The partition we are going to use is a randomized version of the quadtree. Imagine that we pick randomly two integers a, b in $[0, L)$ and then we *shift* the dissection defined above along the $x-$ and $y-$axis by a and b respectively. At the same time we "wrap-around" this shifted dissection as in Figure 4.1(c). Thus every horizontal line with initial y-coordinate y_1 will now have a new y-coordinate $(y_1 + b) \mod L$ and every vertical line with initial x-coordinate x_1 will now have a new x-coordinate $(x_1 + a) \mod L$. We call this dissection a *randomly shifted dissection* and the corresponding quadtree (i.e. the quadtree resulting by cutting of the partitioning at squares that contain at most 1 point) is called a *randomly shifted quadtree*. The random (a, b)-shift is crucial for the algorithm, but since we still treat each "wrapped-around" square as a single region the reader can think of the quadtree as the unshifted one in much of what follows below.

The geometric partition we just defined resembles in many ways the recursive partition in the tree case. But while a tour on a tree could cross partition boundaries only through a tree node, the boundaries of the quadtree partition are lines that can be crossed at any of an infinite number of points. In what follows we will prove that there is a near optimal tour that not only crosses the boundary of each square in the quadtree only a few times, but also that these crossings happen at a small set of prespecified points called *portals*. Each square has 4 portals on its corners and m other equally-spaced portals on each side, where m is a power of 2. A portal of a square is a portal in every descendent of the square in the quadtree.

Definition 5: *Let m, k be positive integers. An (m, k)-*light tour *is one that crosses each quadtree boundary at most k times, and always at one of its m portals.*

In what follows we prove the existence of an (m, k)-light tour (for appropriate values of m, k) that has near-optimal latency. Then we will describe a dynamic programming algorithm that can compute such a tour in quasi-polynomial time.

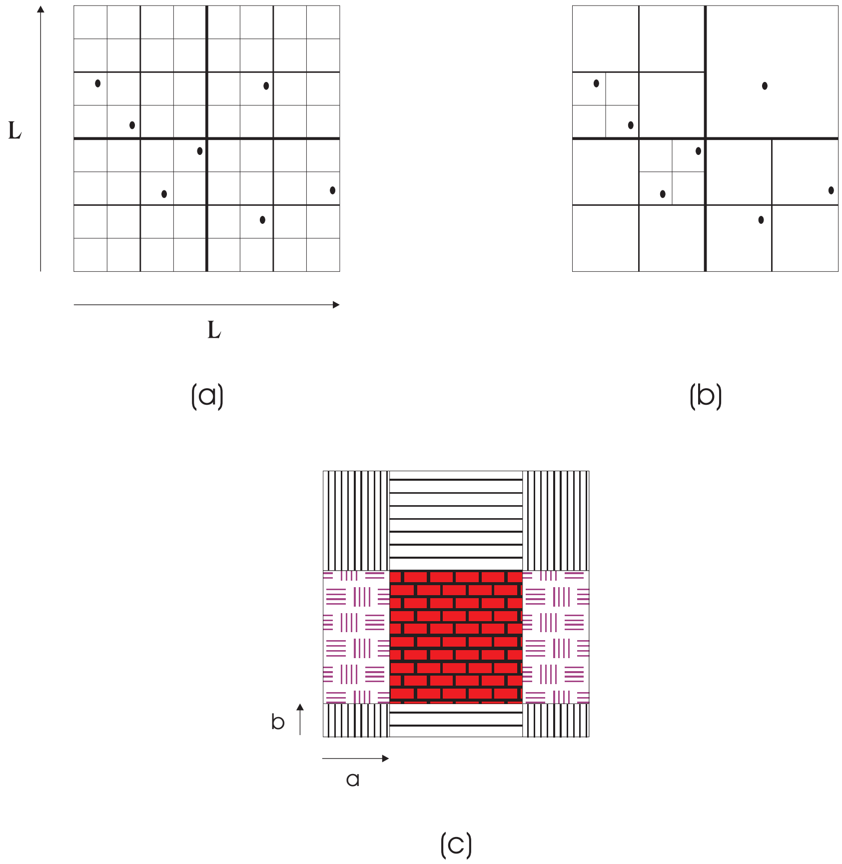


Figure 4.1: A dissection **(a)**, the corresponding quadtree **(b)**, and its shift by (a, b) **(c)**.

4.2.2 The Structure theorem for the Euclidean plane

In section 4.2.1 we assumed that the given points have integral coordinates. We will also demand that all internode distances are between 1 and $O(n^2)$. Such an instance is called *well-rounded*. Obviously the given instance need not be well-rounded, but by adding a *rounding step* similar to the one used in the tree case the algorithm can transform the given instance into a well-rounded one without increasing the MLT by too much. We prove the following structure theorem for the Euclidean plane:

Theorem 4: (Structure theorem for Euclidean plane) *There exist constants c, f such that the following is true for every integer $n > 0$ and every $\epsilon > 0$: For every well-rounded Euclidean instance with n nodes, a randomly-shifted dissection has with probability at least $1/2$ an associated tour that is $(c \log n/\epsilon, f \log n/\epsilon)$-light, and whose latency is at most $(1+\epsilon)OPT$, where OPT denotes the latency of the minimum latency tour. The tour crosses each portal at most $O(\log n/\epsilon)$ times.*

Proof: The main ideas are the same as in Arora's proof [7] after we break up the minimum latency tour into $O(\log n/\epsilon)$ salesman paths. Another further observation is that Arora's proof relies on an expectation calculation, and we will use *linearity of expectations.*

Let $\mathcal{T}$ be the optimum tour. As described in Chapter 3, we break it up into $k = O(\log n/\delta)$ segments, where the ith segment has n_i nodes. We replace each segment by the optimum salesman path for that segment to get a new tour of total latency within a factor $(1+\delta)$ of the optimum. Now we use Arora's proof of his structure theorem. He shows how a salesman tour/path can be modified so that it crosses the boundary of each dissection square at most $O(1/\gamma)$ times, and always at one of the m portals. First he shows how to replace all the crossings through line segments of the randomly shifted quadtree by just *two* crossings in such a way that the following patching lemma (Lemma 3 in [7]) holds:

Lemma 3 (Patching lemma): *Let S be any line segment of length l and π be a path that* *crosses S at least thrice. Then there exist line segments on S whose total length is $3l$ and whose addition to π changes it into a path π' that crosses S at most twice.*

(For the exact way of performing this transformation the reader is referred to [7]).

Then he proceeds into describing how to apply this *local* transformation to the whole path in a structured way so that not to increase the length of the salesman path by too much, and so that the resulting path is (m, r)-light, for $m = O(\log n/\epsilon)$ and $r = O(1/\gamma)$.

Arora's main result is to show that this can be done so that the expected increase in the length of the path is

$$E_{a,b}[\text{increase of path cost}] \leq \frac{6\ OPT}{r} \tag{4.1}$$

where OPT is the cost of the optimal salesman path.

We apply his method simultaneously to our $k = O(\log n/\delta)$ salesman paths $T_1, T_2, \ldots, T_k$, thus ending up with a collection of paths which cross each portal at most $2k$ times (since a path never needs to cross a portal more than twice), and cross the boundary of each quadtree square at most $O(kr) = O(k/\gamma)$ times. For the effect on the latency, note that we are interested in a *weighted* sum of path lengths, where the weight assigned to the ith salesman path is $n_i + n_{>i}$ (these are the vertices whose latency is affected by the change in length of T_i). Arora's main result (4.1) implies that the expected latency increase due to the length increase of path T_i is

$$\begin{aligned} E_{a,b}[\text{total latency increase due to } T_i] &\leq \frac{6}{r}(n_i + n_{>i})OPT_i \\ &\overset{(3.1)}{\leq} \frac{6(1+\delta)}{r} n_{>i} OPT_i \end{aligned} \tag{4.2}$$

where OPT_i is the length of the optimum (i.e. shortest) T_i. By linearity of expectation, the total expected latency increase is

$$\begin{aligned} E_{a,b}[\text{total latency increase due to } T_1, \ldots, T_k] &\leq \frac{6(1+\delta)}{r} \sum_{i=1}^{k} n_{>i} OPT_i \\ &\leq \frac{6(1+\delta)}{r} OPT \end{aligned}$$

because of lower bound (3.2) in Chapter 3.

By picking the appropriate constants γ, δ we conclude that with probability at least $1/2$, the increase in latency is a factor at most $(1+\epsilon)$ and the new tour satisfies the theorem requirements. $\square$

Thus in this section we proved that there is a tour with latency within a factor $1+\epsilon$ of the optimal, that crosses each square boundary in the quadtree only a few times and always through a portal. In the next section we show how to use dynamic programming in order to compute this tour.

4.2.3 The algorithm for the plane

In this section we describe our QPTAS for the Euclidean plane.

Till now we assumed that the given instance is *well-rounded*. Namely it satisfies two conditions: (i) all nodes have integral coordinates, and (ii) all internode distances are between 1 and $O(n^2/\epsilon)$. This assumption is easily met if the first step of the algorithm is a **rounding procedure** very similar to the rounding procedure used in the tree case:

1. Let L be the length of the side of the bounding box. Place a grid of granularity $g = \Theta(\delta L/n^2)$ where δ is a constant $(0 < \delta < \epsilon)$.
2. Move each point to its nearest gridpoint.
3. Divide all edge weights by g.

By an analysis similar to that in the proof of Lemma 2 we get

Lemma 4: *The procedure above transforms a given instance of the MLT into a well rounded-instance. Moreover the MLT in the rounded instance and the original instance differ by at most $O(\delta\ OPT) = O(\epsilon\ OPT)$.*

The well-rounded instance has one more property that we will use in the analysis of the algorithm. Namely, the set of possible lengths for parts of the tour is *discrete* with at most $n^{O(1)}$ elements and can be generated quite easily in polynomial time. Hence when we refer to "guesses" for subtour lengths we will refer to values from this polynomially large set.

The algorithm takes a well-rounded instance as its input, and is very similar to Algorithm 1:

Algorithm 2

1. Build a randomly shifted quadtree for the given instance. Since $L = O(n)$ the quadtree depth is $O(\log n)$ and the number of its nodes is $O(n \log n)$ since it has only n leaves.

   Do the following in a "bottom-up" fashion starting from the leaves of the quadtree:

2. Identify a node (i.e. square) at the current level of the quadtree.

3. "Guess" how many times the tour enters the square, and for each of these times, the portals it crosses to enter and leave the square, what is the length of the tour portion after each crossing, and how many points are on that tour portion.

4. Search the dynamic programming look-up table for subtours consistent with the "guesses".

5. Combine the subtours found to create a new bigger subtour. If the new subtour is consistent with the fact that we are looking for a $(c \log n/\epsilon, f \log n/\epsilon)$-light tour, store this new subtour in the look-up table and go to step 2.

□

Again, by "guessing" we refer to exhaustive enumeration of all possible values of the guessed quantities. This enumeration will produce possibly more than one candidate tours.

We pick the tour with the minimum total latency as our solution, and this will be the output of Algorithm 2.

At a first glance the algorithm seems to miss a key ingredient for the calculation of a tour from its subtours: the order in which the portals are visited in each square. In fact this piece of information is implicit, since the guessed number of nodes visited after each portal crossing also tells us the order in which they occur in the tour (the first portal visited is the one with the longest subsequent tour length guessed, the second is the one with the second longest subsequent tour length and so on). Now it is easy to see that at every step of the algorithm we have the information necessary to reconstruct a bigger subtour from the subtours already calculated. Like in the tree case, the exhaustive enumeration guarantees that one of the tours constructed is the tour of Structure theorem 4, so Algorithm 2 is indeed an approximation scheme.

It remains to prove that Algorithm 2 is a *QPTAS*.

Theorem 5: *Algorithm 2 runs in time* $n^{O(\frac{\log n}{\epsilon})}$ *and computes a tour whose latency is at most* $(1+\epsilon)$ OPT.

Proof: If δ is the constant in the rounding procedure then we apply Theorem 4 so that the approximation factor for the rounded instance is $1+\epsilon-\delta$. Because of Lemma 4 the approximation factor for the tour we compute in the original instance will be $(1+\epsilon)$.

The running time of the algorithm is dominated by the size of the dynamic programming look-up table. We prove the time bound by induction on the depth of the quadtree. The first level (leaves) contains squares with at most one point in them. The tour we are computing is $(c\log n/\epsilon, f\log n/eps)$-light (with c, f being the constants in Theorem 4), so a leaf square is entered and left by the tour $O(\log n/\epsilon)$ times through a pair of portals. This involves enumerating all choices for (a) a multiset of $O(\log n/\epsilon)$ portals on the four sides of the square and (b) the order in which the portals in (a) are crossed by the (m,r)-light tour. It is easy to see that the number of choices in (a) is at most $m^{O(r)} = O(\log n/\epsilon)^{O(\log n/\epsilon)}$ and the number of choices in (b) is $r^{O(r)} = O(\log n/\epsilon)^{O(\log n/\epsilon)}$ for a total of $2^{O(\log n \log\log n/\epsilon)}$ choices. In addition we need to "guess" the length of the tour portion after each crossing ($O(n^{O(1)}/\epsilon)$ possibilities since all distances between points form a set of polynomially many values), and how many points are on that tour portion (n possibilities) for each one of the $O(\log n/\epsilon)$ possible crossings of the tour through the square boundaries. Notice that since we guessed the number of nodes visited after each portal crossing, we know exactly the tour portion that contains the point in the current square. The total number of guesses is thus no more than

$$
\begin{aligned}
& \# \text{ portal arrangements} \times (\text{length} \times \# \text{ points})^{\#\text{ crossings}} \\
= \; & O\left(2^{\frac{\log n \log\log n}{\epsilon}}\right) \times \left(O\left(\frac{n^{O(1)}}{\epsilon}\right) \times n\right)^{O(\frac{\log n}{\epsilon})} \\
= \; & n^{O(\frac{\log n}{\epsilon})}
\end{aligned}
$$

The analysis is the same for the inductive step. Assume that we have calculated all possible subtours for all squares in depth $> i$. Let S be a square at depth i and S_1, S_2, S_3, S_4 be its four children in the quadtree. We guess the same numbers as before for each S_j, $j = 1, \ldots, 4$ ($n^{O(\frac{\log n}{\epsilon})}$ choices) and we look them up in the look-up table constructed thus far. If the choices we find there are consistent with our current guess, we store this guess in the look-up table and continue. Otherwise the algorithm will reject it and will go on to the next possible guess. So the total extra amount of work for each square in the quadtree is $n^{O(\frac{\log n}{\epsilon})}$ and the total running time of the algorithm is $O(n \log n) \times n^{O(\frac{\log n}{\epsilon})} = n^{O(\frac{\log n}{\epsilon})}$ (recall that the quadtree contains only $O(n \log n)$ nodes).

□

Derandomization: The algorithm described is a *randomized* one. But it can be easily derandomized: random shifts a and b take discrete values between 1 and L. By trying all possible $O(n^2)$ values for the pair (a, b) and running the algorithm for each one, we are bound to try one of the "good" values (i.e. one that will give the tour guaranteed by Theorem 4). This procedure will increase the running time by a factor $O(n^2)$.

4.2.4 Euclidean spaces of higher dimension

Since the results in [7] generalize to Euclidean spaces of any *constant* dimension d in a straight-forward manner, the plane algorithm for the MLT generalizes to higher dimensions as well. We will give just the changes needed in order for the proofs in the plane case to go through here.

The grid we place in the d-dimensional *cube* of side length L that surrounds the instance to transform it into a well-rounded one has granularity $\Theta(\epsilon L/(n^2\sqrt{d}))$. Then all internode distances are bound by $O(\sqrt{d}n^2/\epsilon)$.

Instead of randomly shifted quadtrees we use an obvious extension, the 2^d-ary trees. Instead of squares, we are dealing with d-dimensional cubes, so their boundaries are $(d-1)$-dimensional cubes. The m portals placed on these cubes form an orthogonal *lattice* with granularity $W/m^{\frac{1}{d-1}}$, where W is the side length of the cube we place the portals on. The Structure theorem now becomes

Theorem 6: (Structure theorem for Euclidean space of dimension d**)** *There exist constants c, f such that the following is true for every integer $n > 0$ and every $\epsilon > 0$. For every well-rounded instance in Euclidean space of dimension d with n nodes, a randomly-shifted dissection has with probability at least $1/2$ an associated tour that is $(c \cdot (O(\sqrt{d}\log n/\epsilon))^d, f \cdot (O(\sqrt{d}/\epsilon))^d \log n)$-light, and whose latency is at most $(1+\epsilon)OPT$, where OPT denotes the latency of the minimum latency tour. The tour crosses each portal at most $O(\log n/\epsilon)$ times.*

The dynamic programming algorithm will run in time $O((\frac{\log n}{\epsilon})^{(\sqrt{d}/\epsilon)^{O(d)}})n^{O(\frac{\log n}{\epsilon})}$.

4.2.5 Extension to other norms

Like Arora's algorithm for the Euclidean TSP, our algorithm for the MLT generalizes to any *Minkowski* norm in $\mathbb{R}^d$. Any symmetric body C that is symmetric around the origin can be used to define a Minkowski norm: the length of $x \in \mathbb{R}^d$ is defined to be $\frac{||x||^2}{||y||^2}$, where y is the intersection of the surface of C with the line connecting x to the origin. This definition generalizes the l_p norm for $p \geq 1$ (in the case of l_p norm C is the l_p-unit ball centered at the origin).

Distances in any Minkowski norm are within a constant factor of the distances under the l_2 norm. Hence the algorithm described for the Euclidean case works for *any* Minkowski norm as well (with the necessary adjustments of the constants in the Structure theorem 6 and the running time calculations).

4.3 Planar graphs

Our techniques apply also to the MLT problem on *weighted planar graphs* (planar graphs with nonnegative edge lengths). The distance between two nodes is defined as the length of the shortest path in the graph that connects them.

Arora et al. [8] have extended the ideas from [7] to devise a PTAS for the TSP on planar graphs. The first step in their algorithm is the extraction of a *spanner* of the input graph. Informally, a spanner is subgraph of the input graph that approximates within a factor $1 + \epsilon$ the distances in the original graph but with a total edge length that is much smaller (just $O(1/\epsilon)$ times the cost of the minimum spanning tree in the input graph). They use a construction by Althofer et al. [6] that runs in polynomial time.

Let G be the input graph and G' its spanner. In order to apply dynamic programming we need the notion of a separator (like the $\frac{1}{3} : \frac{2}{3}$-partition in the tree case or the quadtree in the Euclidean case) and a *hierarchical decomposition* of the instance. The separator in this case is a *Jordan curve* that cuts a "hole" in the graph, thus separating it into an *exterior* (i.e. the hole) and an *exterior*. By defining the *weight* of a planar graph in an appropriate way, Arora et al. show that in polynomial time they can compute such a curve with two essential properties: (a) the interior and exterior weights are a constant fraction of the total weight, and (b) the interior is connected to the exterior only via a set of $k = O(\log n/\epsilon^2)$ vertices.

After defining the hierarchical decomposition of the graph, they proceed to the proof of a patching lemma similar to lemma 3, i.e. they prove that there is a near optimal tour that crosses the connecting vertices at most a constant number of times.

Now it becomes apparent that a QPTAS similar to the Euclidean one works for the planar graph, too. The role of the square boundaries is played by the Jordan curves and the role of portals is played by the k connecting vertices. Our "guesses" are exactly the same as in the Euclidean case, so we are able to prove the following

Theorem 7: *The algorithm for weighted planar graphs runs in time* $n^{\frac{\log n}{\epsilon}}$ *and computes a tour whose latency is at most* $(1+\epsilon)$ OPT.

Proof: Same as the proof of Theorem 5. □

For a complete description of the planar graph TSP PTAS the reader is referred to [8].

4.4 General metrics

In this section we will use our main idea to derive an algorithm for the MLT for general metrics. The only requirement for the distances between points in the given instance is to satisfy the metric conditions (2.3). For this general case we give a simple *polynomial time* 11.656-approximation algorithm that uses an approximation algorithm for the k-Minimum Spanning Tree (k-MST) as a subroutine. The k-MST problem is defined as the problem of finding the minimum cost tree that spans *exactly* k vertices of a given weighted graph. In the second part of this thesis we give the formal definition of this problem and we describe a polynomial time algorithm that achieves an approximation factor of $2+\delta$ for any constant $\delta > 0$.

Our algorithm has the same flavor as the approximation algorithms in [16, 25]. We note that [25] give an improved 7.18-approximation algorithm, which is more complicated.

The general approach is motivated by the observation in Chapter 3, and we use the numbers $n_1, n_2, \ldots, n_k$ defined there, where $k = O(\log n/\epsilon)$. We will choose $\epsilon = \sqrt{2}$.

The algorithm is as follows. Assume for simplicity that we know the last segment in the tour, which contains $\lceil 1/\sqrt{2} \rceil = 2$ vertices (i.e., 1 edge). Compute the rest of the tour as follows. Let P be the starting node. For $i = 1, \ldots, k$, find, using the algorithm for k-MST, a tour L_i that starts at P and visits n_i nodes that are not visited by $L_1, \ldots, L_{i-1}$. Of the two possible directions of each tour, pick the one that minimizes the latency. Output the concatenation of these tours (using shortcuts to avoid multiple visits to already visited points).

Theorem 8: *The algorithm just described runs in polynomial time and achieves an approximation factor of* 11.656.

Proof: We analyze this algorithm as follows. Let $\mathcal{T}$ be the optimum tour. Denote by T_i the length of the i-th segment of $\mathcal{T}$ (which contains n_i nodes). Then, as in (3.2), a lower bound on OPT, the latency of the optimum tour, is

$$\text{OPT} \geq \sum_{i=1}^{k} n_{>i} \cdot T_i, \tag{4.3}$$

where $n_{>i} = n_{i+1} + n_{i+2} + \cdots + n_k$ is the number of nodes appearing in the last $k-i$ segments.

Let d_i be the length of the tour L_i computed by our algorithm. We claim that

Claim 1:

$$d_i \leq (2+\delta) \times 2(T_1 + \ldots + T_i) \textit{ for } i = 1, 2, \ldots, k-1 \tag{4.4}$$

Proof: (of Claim 1) The reason for the "$(2+\delta)$" is that it is the approximation ratio of k-MST algorithm. From now on we will treat this factor as equal to 2 (by making δ very small, e.g. $\delta = 10^{-6}$). The rest of the expression is explained as follows. Consider the closed tour starting from P, including the first i segments of $\mathcal{T}$, and returning to P. Its length is at most $2(T_1 + T_2 + \cdots + T_i)$, and it includes at least $(n_1 + n_2 + \ldots + n_i)$ nodes. At least n_i of these nodes are not in $T_1, \ldots, T_{i-1}$ (since these first $i-1$ segments contain in total only $n_1 + n_2 + \cdots + n_{i-1}$ nodes). Thus $2(T_1 + T_2 + \cdots + T_i)$ is an upper bound on the length of shortest tour that starts and finishes at P and includes at least n_i nodes not on $T_1, \ldots, T_{i-1}$. This finishes the justification for claim 1. □

Now notice that the latency of the n_i new nodes visited during tour L_i is upper bounded by $\frac{1}{2}n_i d_i + n_i \sum_{l=1}^{i-1} d_l$. The second term is an upper bound for the total latency incurred due to the $i-1$ segments of $\mathcal{T}$ preceding the i-th segment. The first term is an upper bound for the latency due to the i-th segment itself, since the latency in the forward direction plus the latency in the backward direction is equal to $n_i d_i$, and we traverse tour L_i in the direction that minimizes the latency of the n_i new points visited. So the total latency of our solution —ignoring the last segment — is upper bounded by

$$\begin{aligned} A &= \sum_{i=1}^{k-1} \left[n_i \left(\frac{1}{2} d_i + \sum_{l=1}^{i-1} d_l \right) \right] \\ &= \frac{1}{2} \sum_{i=1}^{k-1} n_i d_i + \sum_{i=1}^{k-1} \sum_{l=1}^{i-1} n_i d_l \\ &\leq \frac{1}{2} \sum_{i=1}^{k-1} n_i d_i + \sum_{i=1}^{k-1} d_i n_{>i} \end{aligned}$$

From Chapter 3, $n_i \leq (1+\epsilon) n_{i+1}$ and $n_{>i} \leq \frac{n_i}{\epsilon}$, so

$$\begin{aligned} A &\leq \frac{1+\epsilon}{2}\sum_{i=1}^{k-1} n_{i+1}d_i + \frac{1+\epsilon}{\epsilon}\sum_{i=1}^{k-1} n_{i+1}d_i \\ &= \frac{(1+\epsilon)(2+\epsilon)}{2\epsilon}\sum_{i=1}^{k-1} n_{i+1}d_i \end{aligned}$$

Observing that $n_{>i} = \sum_{j=i+1}^{k} n_j$ and ignoring the last segment (which was computed optimally) we also have

$$\begin{aligned} \sum_{l=1}^{k-1} n_{l+1}d_l &\overset{(4.4)}{\leq} 4\sum_{l=1}^{k-1} n_{l+1}\sum_{i=1}^{l} T_i \\ &= 4\sum_{l=1}^{k-1} n_{>l}T_l \\ &\leq 4 \times OPT \end{aligned}$$

and thus

$$A \leq 2\frac{(1+\epsilon)(2+\epsilon)}{\epsilon}\ OPT.$$

The factor $\frac{(1+\epsilon)(2+\epsilon)}{\epsilon}$ is minimized for $\epsilon = \sqrt{2}$ and its value is $3 + 2\sqrt{2} = 5.828$. Thus $A \leq 11.565\ OPT$. □

Chapter 5

Extending the MLT problem

It turns out that our main idea from Chapter 3 is general enough to apply to a broader category of problems. In this chapter we deal with the following *weighted* version of MLT:

WEIGHTED MINIMUM LATENCY TOUR (WMLT)

Input : A set of n points (one of them designated as the starting point p_1), a symmetric distance matrix $[d_{ij}]$ and an integral weight $w_i \geq 0$ for each node $i = 1, 2, \ldots, n$.

Output : A tour $p_1 \to p_2 \to \ldots \to p_n$ that visits *all* points and minimizes the total *weighted* latency

$$\sum_{i=2}^{n} \left[w_{p_i} \sum_{j=1}^{i-1} d_{p_j, p_{j+1}} \right] \tag{5.1}$$

Obviously this is an extension to the MLT problem: the contribution of each node to the objective function is the length of the tour from the start to the first visit at the node, *weighted by the weight of the node w_i*. When $w_i = 1$, $\forall i$ we get the MLT problem as defined in Chapter 2. Notice that again we can express the objective function in a form similar to formula (2.2), namely

$$\sum_{i=1}^{n-1} \left[\sum_{j=i+1}^{n} w_{p_j} \right] d_{p_i, p_{i+1}} \tag{5.2}$$

Again we are going to study instances where d defines a metric (i.e. it satisfies conditions (2.3)). In this case whenever a weight w_i is 0, point i can be moved to the end of a tour without increase to the value of the objective function, and it is enough to solve the problem for the subset of points with non-zero weights. Hence from now on the w_i's will be integers greater than 0.

5.1 Extending the main idea

The difference between objective function 5.2 and the MLT objective function 2.2 is that in the latter point i contributes a unit weight while in the former it contributes with weight w_i. Thus our main idea extends in many cases naturally to the more general setting of WMLT.

Let $\mathcal{T} = p_1 \to p_2 \to \ldots \to p_n$ be an optimal tour with total weighted latency OPT. Let $W = \sum_{i=1}^{n} w_{p_i}$ be the total point weight and $\epsilon > 0$ any parameter. We will assume that the following is true for this instance of WMLT:

Assumption 1: *$\mathcal{T}$ can be broken into k segments, so that in segment i it visits n_i points with total weight W_i given by the following expressions:*

$$\begin{aligned} W_1 &= \left\lceil \frac{\epsilon\ W}{1+\epsilon} \right\rceil \\ W_i &= \left\lceil \frac{W_{i-1}}{1+\epsilon} \right\rceil,\ i = 2, 3, \ldots, k \end{aligned} \tag{5.3}$$

Obviously $k = O(\frac{\log W}{\epsilon})$. To simplify the calculations we will assume that ceilings are not needed in the expressions above. This assumption will not affect our results. Let the length of the i^{th} segment be T_i. If we let $W_{>i}$ denote the total weight of nodes visited in segments numbered $i+1$ and later, then a simple calculation shows that (and this was the reason for our choice of W_i's)

$$W_{>i} = \sum_{j>i} W_j \leq \frac{W_i}{\epsilon}, \text{ for every } i = 1 \ldots k-1 \tag{5.4}$$

As in the case of the MLT, we are going to show that if one replaces each segment by the minimum-cost traveling salesman path on the same subset of nodes, while maintaining the starting and ending points, the new total weighted latency is at most $(1+\epsilon)$OPT. The lower bound on the OPT is similar to the lower bound in the MLT case:

$$\mathrm{OPT} \geq \sum_{j=1}^{m-1} W_{>j} \cdot T_j \tag{5.5}$$

By replacing each segment T_i with the optimum salesman path T_i' we increase the weighted latency of points within the segment by at most $W_i T_i'$. Hence the total weighted latency of the new tour $\mathcal{T}'$ can be upper bounded as follows:

$$\begin{aligned}
\mathrm{cost}(\mathcal{T}') &\leq \sum_{i=1}^{k} W_i T_i' + \sum_{i=1}^{k} W_{>i} T_i' \\
&\leq \epsilon \cdot \sum_{i=1}^{k} W_{>i} T_i + \sum_{i=1}^{k} W_{>i} T_i \\
&\overset{(5.5)}{\leq} (1+\epsilon)\ \mathrm{OPT}
\end{aligned}$$

Thus we have shown that under Assumption 1, the analogue of Theorem 1 can be proven:

Theorem 9: *Let OPT be the total latency of the WMLT under Assumption 1. There exists a tour that is a concatenation of $O(\frac{\log W}{\epsilon})$ optimal salesman paths, and whose total weighted latency is at most $(1+\epsilon)OPT$.*

Even if, instead of the optimum salesman path in each segment, we use a $(1+\gamma)$-approximate salesman path, then the latency of the tour of Theorem 9 is $(1+\gamma\cdot\epsilon+\gamma+\epsilon)$OPT.

5.1.1 Relation to weighted Vehicle Routing

The approximation-preserving reduction from the MLT to the Weighted Vehicle Routing with per-mile costs works for the WMLT, too. For completeness, we give the reduction that uses an oracle for the routing problem in order to solve the WMLT ($k, W, W_i, W_{>i}$ have the same meaning as in the previous section): "Let p_0 denote the starting node of the WMLT.

For every sequence of $k = O(\log W/\epsilon)$ nodes $p_1, p_2, \ldots, p_k$, use the Vehicle Routing Oracle to construct a solution to the instance in which there are k vehicles, and the capacity of the ith vehicle is W_i, its per-mile cost is $W_{>i}$, and its starting and end depots are p_i and p_{i+1} respectively. At the end, output the lowest cost solution found (over the choice of all sequences of k points)."

Clearly, if the vehicle routing oracle computes a ρ-approximation in polynomial time, our reduction will lead to a $\rho(1+\epsilon)$-approximate solution for WMLT in $n^{O(\log W/\epsilon)}$ time. This follows from Theorem 9, since it suffices to go over all possible sequences $p_1, p_2, \ldots, p_k$, and this increases the running time by a factor of at most $O(n^k) = n^{O(\frac{\log W}{\epsilon})}$.

5.1.2 Approximation algorithms for the Weighted MLT

First we point out the differences between our rounding procedures for the MLT and the rounding for the weighted MLT. In this case it suffices to assume wlog that the minimum nonzero internode distance is 1 and maximum internode distance is $O(n \cdot W/\epsilon)$, where $W = \sum_{i=1}^{n} w_i$ and $\epsilon > 0$ any parameter. The reason is that if L denotes the diameter of the space, then L is a lower bound on the weighted minimum latency. Again we merge all pairs of nodes with internode distance at most $\frac{\epsilon L}{n \cdot W}$. This affects the latency of the optimum tour by at most $\epsilon L \leq \epsilon \mathrm{OPT}$. Furthermore, the ratio of the maximum internode distance and the minimum nonzero internode distance is $O(nW/\epsilon)$. Since ϵ is constant, we will often think of the maximum internode distance as $O(nW)$. Note that in the weighted case the internode distance (and therefore the running time of our algorithms) depend directly on W. If W is, for example, exponential on the size of the input, our algorithms will run in exponential time. We give the running times of algorithms described for the MLT, when they are applied in the case of WMLT. Their correctness is ensured by the extension of our main ideas in Section 5.1.

5.1.3 The tree case

Theorem 10: *Algorithm 1 runs in time* $(nW)^{O(\log W/\epsilon)}$ *and computes a tour whose weighted latency is at most* $(1+\epsilon)$ OPT.

Proof: The proof is the same as for Theorem 3 and uses Theorem 9. $\square$

5.1.4 Euclidean spaces

Theorem 11: *The algorithm for the MLT in Euclidean spaces of constant dimension* d *runs in time* $O((\frac{\log W}{\epsilon})^{(\sqrt{d}/\epsilon)^{O(d)}})(nW)^{O(\frac{\log W}{\epsilon})}$ *and computes a tour whose weighted latency is at most* $(1+\epsilon)$ OPT.

Proof: See Section 4.2.4 and adapt Structure theorem 6 to work with Theorem 9. $\square$

This result extends also to any Minkowski norm (cf. Section 4.2.5).

5.1.5 Planar graphs

Theorem 12: *The algorithm for weighted planar graphs runs in time* $(nW)^{\frac{\log W}{\epsilon}}$ *and computes a tour whose weighted latency is at most* $(1+\epsilon)$ OPT.

Proof: See Section 4.3. $\square$

Chapter 6

The k-Minimum Spanning Tree problem

In Section 4.4 it was pointed out that the currently best polynomial time approximation algorithm for the general metric MLT due to Goemans & Kleinberg [25] uses an algorithm that calculates a k-Minimum Spanning Tree (k-MST) as a subroutine. In the second part of this thesis we study the k-MST problem. In particular, we give an improved approximation algorithm for this problem that achieves an approximation factor of $2 + \delta$ for any constant $\delta > 0$.

6.1 Definition of the k-MST problem

One of the most well studied problems in the area of algorithms is to find a Minimum Spanning Tree (MST) of a (weighted) graph. Its definition is simple: we are given a graph G (with edge weights in the weighted case) and we want to find a subgraph that is a tree of minimum total edge weight and spans all n vertices of G (i.e. a minimum spanning tree). The k-MST is a natural generalization of the MST: it is a subtree of G with minimum total edge weight that spans *exactly* k vertices of G. Notice that the number k is a new parameter of the problem that is given as an input. Obviously for $k = n$ the k-MST is a MST for G.

k-MINIMUM SPANNING TREE (k-MST)

Input : An undirected graph $G = (V, E)$ with non-negative edge costs and an integer k.

Output : A tree in G of minimum total edge cost that spans exactly k vertices of G.

A problem similar to the k-MST is k-TSP: find a tour of minimum cost that visits exactly k vertices; we call such a tour a k-tour.

k-TRAVELING SALESMAN PROBLEM (k-TSP)

Input : An undirected graph $G = (V, E)$ with non-negative edge costs and an integer k.

Output : A tour in G of minimum total edge cost that visits exactly k vertices of G.

6.1.1 Previous work

The problem is known to be **NP**-complete [40, 22, 44]. A sequence of results reduced the approximation factor for this problem from an initial $O(\sqrt{k})$ [40] to $O(\log^2 k)$ [13] and $O(\log k)$ [39] to constant factor [17]. A better constant approximation factor of 3 was achieved by Garg [23], and a modification of this algorithm in [12] achieved the best known factor of 2.5. For the special case of finding the k-MST of points on the Euclidean plane, a PTAS is known to exist ([7] and independently, [34]).

Since the traveling salesman problem (TSP) reduces to k-TSP, we do not expect to have a bounded approximation guarantee for general instances of k-TSP. However, if the edge costs satisfy the triangle inequality, Garg's ideas also give a 3-approximation algorithm for the k-TSP.

6.1.2 Main result

This thesis presents, for any $\epsilon > 0$, an $n^{O(1/\epsilon)}$ time algorithm that computes a $2+\epsilon$ approximation to the k-MST problem. We use a simple modification to Garg's 3-approximation algorithm, which uses (as does the earlier [17]) an LP relaxation and the primal-dual framework.

To explain our improvement, we first recall how Garg obtained an approximation ratio of 3. He uses the obvious LP relaxation for the problem. The integrality gap for this relaxation (i.e., the worst-case ratio of the integer optimum and the fractional optimum) is unbounded. However, he uses an additional lowerbound for the optimum cost (in addition to the LP value), the diameter D of the optimum k-tree. (The diameter can be easily "guessed" by the algorithm by trying all possible $\binom{n}{2}$ internode distances.) Another source of improvement in Garg's paper as compared to the earlier [17] is a modification of the 'traditional' primal-dual schema (as described in [26]) to incorporate a final phase of 'potential' reduction that allows him to prove tighter upper and lower bounds. Roughly speaking, the factor 3 ultimately derives from two sources: a (familiar) factor 2 due to the traditional primal-dual analysis, and a factor 1 from the diameter lowerbound.

Our improvement derives from changing the contribution of the diameter lowerbound from OPT to $\epsilon \cdot OPT$ (OPT is the length of a minimum k-tree). We achieve this by allowing the algorithm to "guess" not just 2 vertices in the optimum k-tree (which are needed to guess the diameter), but as many as $O(\frac{1}{\epsilon})$ vertices. This set of vertices have the property that every vertex in the optimum tree has distance at most $\epsilon \cdot OPT$ to this set. Note that the $n^{O(1/\epsilon)}$ running time is due to the necessity of having to try all $\binom{n}{O(1/\epsilon)}$ choices for this set of vertices.

6.1.3 Connection to MLT

We note that the improved $2 + \epsilon$ approximation factor has immediate implications for the problem of finding the *minimum latency tour* in a metric space. The first constant factor approximation algorithm for this problem was given in [16], while [25] give the currently best factor of $3.59 \times \alpha$, where α is the approximation factor for the minimum k-tour. Therefore our result improves the approximation factor for the minimum latency problem from 10.77 to 7.18, albeit with a higher running time.

Chapter 7

The approximation algorithms for k-MST and k-TSP

Our algorithm is a modification of Garg's algorithm [23] that achieved an approximation factor of 3. It is based on the primal-dual method which has seen a wide variety of applications in the area of approximation algorithms since its introduction in the field by [2]. The reader can find excellent surveys on this method by M.Goemans and D.Williamson in [28] and by V.V.Vazirani [43].

7.1 Preliminaries

Like Garg [23], we are going to deal with the rooted version of the k-MST problem. Namely, in addition to graph G, the weights of the edges and k, we are also given a vertex $r \in V$ to be included in the solution. Notice that the general problem reduces to this problem by simply solving the rooted version of the problem for all $|V|$ possible roots r and choosing the solution with the minimum cost. Obviously this increases the running time by a factor of $O(|V|)$. Unlike Garg though, we are going to assume that we have some more information about the optimal solution besides its root. As with the root, this information is not available to us, so we will have to enumerate all possibilities and pick the best of the solutions produced.

Hence our improvement is based again on "limited guessing", a theme we encountered in the MLT problem. The k-MST algorithm will guess a set of $O(1/\epsilon)$ vertices from the optimum k-tree that, in a way, are going to indicate how "spread" this tree is in G. If $|V| = n$, it is apparent that the possibilities for these 'special' vertices are at most $n^{O(1/\epsilon)}$ and the running time of the algorithm will increase by the same factor.

7.1.1 Special vertices

We are motivated into "guessing" $O(1/\epsilon)$ special vertices of the optimum k-tree by the following simple lemma:

Lemma 5: *For every $\epsilon > 0$ the following is true in any weighted tree of total edge length L: There is a set of $O(1/\epsilon)$ vertices such that all other vertices in the tree are at distance at most $O(\epsilon L)$ from one of the vertices in the set.*

Proof: First find all edges that are longer than ϵL. There are at most $\frac{1}{\epsilon}$ such edges. The $O(1/\epsilon)$ vertices incident to these edges are included in the set. Now pick the rest of the vertices as follows:

First label all unpicked vertices in the tree as *unmarked*. Repeat the following procedure until all vertices in the tree are either *picked* or *marked*:

1. Label as *marked* all *unmarked* vertices that are at a distance at most ϵL from some picked vertex.

2. If there is no *unmarked* vertex left, then terminate. Otherwise, pick an unmarked vertex whose distance from one of the picked vertices is between ϵL and $2\epsilon L$ (there is at least one such vertex, since we have already picked the vertices adjacent to edges of length at least ϵL). Mark this vertex and go to the previous step.

It is obvious that the vertices picked by the procedure above satisfy the lemma requirements.

□

Lemma 5 will hold also for the optimum k-tree of total edge length OPT. Let $W \subseteq V$ be such a set of special vertices for the k-MST guaranteed by the lemma. Our algorithm will "guess" W. By "guessing" we mean that the algorithm tries all possibilities (in $n^{O(1/\epsilon)}$ time) and runs the primal-dual procedure described below for each of them. It is guaranteed to ultimately guess the correct W (in which case the primal-dual procedure will return a tree of cost $(2+\epsilon)\ OPT$, as we shall see).

7.1.2 The LP formulation

The constant factor approximation algorithms of [17], [23] are based on the relaxation of an integer program for the rooted k-MST . Let r be the root of the k-tree and let $\delta(S)$ denote the set of edges with one end-point in $S \subseteq V$ and the other outside S. We associate a binary variable x_e with the edge $e \in E$: $x_e = 1$ implies that edge e belongs to the k-tree, otherwise $x_e = 0$. We also associate a variable x_v with the vertex $v \in V$ and $x_v = 1$ implies that the k-tree spans v, otherwise $x_v = 0$. Thus an obvious constraint is $\sum_{v \in V} x_v = k$. Another obvious set of constraints that should be included are those ensuring that all picked vertices (i.e. v's with $x_v = 1$) are connected to the root. In other words, for every set $S \subseteq V - r$ that *does not* contain the root r, the number of edges with $x_e = 1$ that have exactly one end-point in S must be at least as large as x_v for any $v \in S$. Thus for each set $S \subseteq V - r$ and vertex $v \in S$ constraint $\sum_{e \in \delta(S)} x_e \geq x_v$ must hold. Given a cost c_e for each edge e, we seek to minimize the total cost of the picked edges, i.e. we want to minimize $\sum_{e \in E} c_e x_e$. Hence the integer program for the rooted k-MST used by previous researchers will be the following:

$$
\begin{array}{rcll}
\text{minimize} & & \sum_{e\in E} x_e c_e & \text{subject to} \\
\sum_{e\in\delta(S)} x_e & \geq & x_v & (\forall (v,S) : v \in S \subseteq V - r) \\
\sum_{v\in V} x_v & = & k & () \\
x_v & \in & \{0,1\} & (\forall v \in V) \\
x_e & \in & \{0,1\} & (\forall e \in E)
\end{array}
$$

To simplify notation, from now on whenever we refer to a set S it is understood that S does not contain the root r.

As mentioned, our algorithm will "guess" a set $W \subseteq V$ of $O(1/\epsilon)$ vertices and require that the solution should pick them. This means that $x_v = 1$ for all $v \in W$ in the above LP. Let $w = |W|$. Our new integer program takes W into account:

$$
\begin{array}{rcll}
\text{minimize} & & \sum_{e\in E} x_e c_e & \text{subject to} \\
\sum_{e\in\delta(S)} x_e & \geq & x_v & (\forall (v,S) : v \in V \setminus W, v \in S) \\
\sum_{e\in\delta(S)} x_e & \geq & 1 & (\forall (v,S) : v \in S \cap W) \\
\sum_{v\in V\setminus W} x_v & = & k - w & \\
x_v & \in & \{0,1\} & (\forall v \in V \setminus W) \\
x_e & \in & \{0,1\} & (\forall e \in E)
\end{array}
$$

Note that the variables x_v for all $v \in W$ do not participate in this new integer program, which we call IP. Since the k-MST problem is NP-complete, there is not much hope that we can solve this integer program. Instead, we will work with its linear relaxation. Specifically, we will relax variables x_v, x_e to be real numbers between 0 and 1. So the 0/1 constraints on these variables are replaced by $0 \leq x_e, x_v \leq 1$. In fact if we impose that $x_v \leq 1$ there cannot be edges with $x_e > 1$ in an optimum solution, because then we could reduce them to $x_e = 1$ and obtain another feasible solution with smaller cost. Thus we will use the following fractional relaxation for the k-MST problem in which the guessed set of vertices W has to be included in the solution:

$$
\begin{array}{rcll}
\text{minimize} & & \sum_{e\in E} x_e c_e & \text{subject to} \\
\sum_{e\in\delta(S)} x_e & \geq & x_v & (\forall (v,S) : v \in V \setminus W, v \in S) \\
\sum_{e\in\delta(S)} x_e & \geq & 1 & (\forall (v,S) : v \in S \cap W) \\
\sum_{v\in V\setminus W} x_v & = & k - w & () \\
x_v & \leq & 1 & (\forall v \in V \setminus W) \\
x_v & \geq & 0 & (\forall v \in V \setminus W) \\
x_e & \geq & 0 & (\forall e \in E)
\end{array}
$$

We will call this linear program LP. Its dual has a variable $y_{v,S}$ for all pairs (v, S) such that $v \in S \subseteq V - r$ (notice that v can be any vertex, even a vertex in W), a non-negative variable p_v for vertex $v \in V \setminus W$ and a free variable p:

$$
\text{maximize} \quad (k-w)p - \sum_{v\in V\setminus W} p_v + \sum_{v\in W} \sum_{S:v\in S} y_{v,S} \quad \text{subject to}
$$

$$
\begin{array}{rcll}
\sum_{S:v\in S} y_{v,S} + p_v & \geq & p & (\forall v \in V \setminus W) \\
\sum_{S:e\in\delta(S)} \sum_{v\in S} y_{v,S} & \leq & c_e & (\forall e \in E) \\
p_v & \geq & 0 & (\forall v \in V \setminus W) \\
y_{v,S} & \geq & 0 & (\forall v, S : v \in S) \\
p & \text{free} & &
\end{array}
$$

We will call this linear program $DUAL$.

For notational ease we introduce some new (dummy) variables: α_v for each vertex v and z_S for each subset S. These are shorthands for the following quantities:

$$
\alpha_v = \sum_{S:v\in S} y_{v,s} \tag{7.1}
$$

$$
z_S = \sum_{v\in S} y_{v,S} \tag{7.2}
$$

That is, α_v is equal to the sum of dual variables $y_{v,S}$ vertex v "feels" because of the sets S that contain it, and z_S is the sum of all dual variables $y_{v,S}$ for vertices v that are contained in set S.

The following lemma is a simple modification of Claim 2.1 in [23]:

Lemma 6: *In the optimum solution to the dual, p has a value between the $(k-w)^{th}$ and the $(k-w+1)^{th}$ smallest values of α_v for $v \in V \setminus W$. The optimum value of the dual program is the sum of the $k-w$ smallest α_v's for $v \in V \setminus W$ and the α_v's of the vertices in W.*

Proof: We will deal with the a_v's of vertices $v \in V \setminus W$. Notice that if $\alpha_v \geq p$ then $p_v = 0$ else $p_v = p - \alpha_v$. If p is greater than the $(k-w+1)^{th}$ α_v then we can decrease p and all positive p_v's by some small ϵ, thus increasing the objective function by at least ϵ and contradicting the optimality of the solution. On the other hand if p is strictly smaller than the $(k-w)^{th}$ smallest a_v's then for at most $k-w-1$ vertices $p - a_v \geq 0$; for the rest $p - a_v < 0$. Hence there is some $\epsilon > 0$ such that $\epsilon < \min_{v:p-a_v<0}\{|p-a_v|\}$. But then we can increase p and the $k-w-1$ p_v's that correspond to v's with $p - a_v \geq 0$ by ϵ without losing feasibility for the new solution. The change of the objective function will be an increase of $(k-w)\epsilon$ due to the increase of p by ϵ and a decrease of at most $(k-w-1)\epsilon$ because at most $k-w-1$ p_v's were increased, so there is a net increase of the solution by ϵ. This contradicts the optimality of the solution.

Thus p has a value between the $(k-w)^{th}$ and $(k-w+1)^{th}$ smallest a_v's. In this case only the vertices with the $k-w$ smallest a_v's have a positive value for p_v. For these vertices $a_v = p - p_v$, while for the other vertices in $V \setminus W$ $p_v = 0$. Now it is obvious that the optimum value is the sum of the $k-w$ smallest α_v's for $v \in V \setminus W$ plus the α_v's of the vertices in W. □

Definition 6: *A* potential assignment *is any function $\pi : V \to \mathbb{R}^+$ such that $\pi(r) = 0$, where r is the (given) root of the k-tree.*

That is, a potential assignment is just the assignment of positive real values to the vertices in V with the root r taking the value 0. In what follows, we are going to use a special class of such assignments called *feasible potential assignments.*

Definition 7: *A potential assignment π is* feasible *if there exists an assignment of non-negative values to $y_{v,S}$ for all $v, S : v \in S$ such that for any vertex $v \in V$, $\pi(v) \leq \sum_{S:v\in S} y_{v,S}$ and for any edge $e \in E$, $\sum_{S:e\in\delta(S)} \sum_{v\in S} y_{v,S} \leq c_e$.*

Note that the values α_v that follow from any solution of the dual program define a feasible potential assignment: $\pi(v) := a_v$ and $\pi(r) := 0$. Hence the dual program can now be interpreted as finding a feasible potential assignment that maximizes the sum of the a_v's for $v \in W$ plus the $k - w$ smallest a_v's of the other vertices.

Any feasible potential assignment gives a lower bound for the cost of any tree rooted at r:

Lemma 7: *Let π be a feasible potential assignment and T be any tree that is rooted at r. Then T has cost at least $\sum_{v\in T} \pi(v)$.*

Proof: Let $[y_{v,S}]_{v,S}$ be the nonnegative vector that witnesses the feasibility of π. Then we have

$$
\begin{aligned}
\sum_{e\in T} c_e &\geq \sum_{e\in T} \sum_{S:e\in\delta(S)} \sum_{v\in S} y_{v,S} \\
&\geq \sum_{e\in T} \sum_{S:e\in\delta(S)} \sum_{v\in T\cap S} y_{v,S} \\
&= \sum_{v\in T} \sum_{S\ni v} \sum_{e\in T\cap\delta(S)} y_{v,S} \\
&\geq \sum_{v\in T} \sum_{S\ni v} y_{v,S} \\
&\geq \sum_{v\in T} \pi(v)
\end{aligned}
$$

where the first inequality is due to feasibility, the second follows from dropping some terms, the third from some rearrangement and the penultimate line follows from the observation that any set S that intersects the tree must have a tree edge leaving it. □

7.2 The algorithm

Our algorithm can be broken down into three parts:

1. Preprocessing.

2. Application of the primal-dual method.

3. Modification of the tree produced in Part 2 to produce a tree with *exactly* k vertices.

The last two parts are almost identical with the corresponding parts in Garg's algorithm. But in our case the algorithm is helped by the preprocessing stage in order to achieve a better approximation factor.

7.2.1 Preprocessing

The algorithm we describe needs some more information than that given by the original instance of the k-MST problem we are trying to solve. Namely, it is going to need the following:

- the root r of the k-MST
- the cost OPT of the k-MST
- the set W of special vertices that is guaranteed to exist by Lemma 5

Since this information is not available the algorithm needs to "guess" it. By "guess" we mean exhaustive enumeration of all possibilities, solving the problem (if it is solvable) for each one of them, and finally picking the cheapest solution as the algorithm output. In order to estimate the running time of our algorithm we need to calculate how many possibilities there are for this extra information. Obviously there are $|V| = n$ possibilities for r. For OPT we will need to round the instance as we did, for example, in the MLT case for trees in order to avoid superpolynomial (on the size of the input) edge costs. A **rounding procedure** similar to the one described in Section 4.1.2 will work:

1. Let $\delta > 0$ be any constant. **Guess** the length of the longest path (diameter) D in the k-MST .

2. Merge (by contracting edges) all pairs of nodes with internode distance at most $\delta D/k$.

3. Round each edge weight to its closest multiple of $\delta D/k$.

4. Divide all edge weights by $\delta D/k$.

Note that for Step 1 we need another piece of information not given to us: the diameter of the optimal solution D. So the algorithm will try all n possibilities for the vertex that is the farthest from r. In exactly the same way as in Lemma 2 we can prove

Lemma 8: *Let OPT be the cost of the k-MST in the given instance and D, δ are as in the above procedure. Then the k-MST on the rounded instance and OPT differ by at most $O(\delta\ OPT)$. Moreover, each internode distance is between* 1 *and* $O(k/\delta) = O(n)$.

Proof: Steps 2 and 3 can affect the cost of the optimal k-tree by at most $k \times \frac{\delta D}{k} \leq \delta OPT$, and the ratio of maximum to minimum internode distance is at most $\frac{k}{\delta} = O(n)$ since δ is a constant. By rescaling we get an instance of internode distances between 1 and $O(n)$. □

Thus, after the rounding, OPT lies between 1 and $O(kn)$. Finally, there are $\binom{n}{O(1/\epsilon)} = n^{O(\frac{1}{\epsilon})}$ possibilities for W. Overall, the running time of the algorithm will increase by a factor of $n^{O(\frac{1}{\epsilon})}$ due to "guessing".

The preprocessing part of the algorithm is given in Figure 7.1.

Input : Graph $G = (V, E)$, integer k.

1. Guess the root r and the diameter D of the k-MST.
2. Round the instance. Let $G' = (V', E')$ be the graph of the new instance.
3. Guess the cost OPT of the k-MST in G' and the set of special vertices W.
4. Remove from G' all the vertices (and the incident edges) whose distance from all vertices in W is greater than $\epsilon \cdot \mathrm{OPT}$ (these vertices cannot be spanned by the k-MST due to the special property of W). Let $G'' = (V'', E'')$ be the new (pruned) graph.

Output : Graph G'', root r, set W, OPT.

Figure 7.1: Preprocessing

7.2.2 Using the Primal-Dual method

In the heart of our algorithm (as in Garg's algorithm [23] and the first constant approximation factor algorithm by Blum et al. [17]) there is an application of the Primal-Dual method. Hence, before we proceed into giving the details of the algorithm, it is instructive to give an overview of this method.

Linear programs LP and $DUAL$ are bound together by the duality properties of linear programming. It is well known that the optimal solution of a linear program has the same value as the optimal solution of its dual. These two optimal solutions must obey the **complementary slackness conditions**. If (x_v, x_e) is the optimal solution of LP and $(y_{v,S}, p_v, p)$ is the dual optimal solution of $DUAL$ then the complementary slackness conditions translate into the following:

Primal complementary slackness conditions

- For each $v \in V \setminus W$: either $x_v = 0$ or $\sum_{S:v \in S} y_{v,S} + p_v = p$.
- For each $e \in E$: either $x_e = 0$ or $\sum_{S:e \in \delta(S)} \sum_{v \in S} y_{v,S} = c_e$.

Dual complementary slackness conditions

- For each (v, S) with $v \in V \setminus W, v \in S$: either $y_{v,S} = 0$ or $\sum_{e \in \delta(S)} x_e = x_v$.
- For each (v, S) with $v \in S \cap W$: either $y_{v,S} = 0$ or $\sum_{e \in \delta(S)} x_e = 1$.
- For each $v \in V \setminus W$: either $p_v = 0$ or $x_v = 1$.

The relation between optimality of the solutions and the complementary slackness conditions goes also the other way: if one finds a pair of (primal, dual) feasible solutions that satisfy the complementary slackness conditions, then these solutions are optimal.

Unfortunately solving LP (or its dual $DUAL$) produces fractional (i.e. non-integral) solutions (as expected, since the k-MST problem is NP-complete and an integral optimal solution to LP would imply P=NP). The main idea behind the Primal-Dual method is to *relax* the dual complementary slackness conditions and then try to construct a pair of feasible (primal, dual) solutions, that satisfy the original primal conditions and the relaxed dual conditions, and the primal solution is an integral one. The relaxed dual slackness conditions will guarantee that although the integral primal solution we constructed is not necessarily optimal (since it may not satisfy the original dual complementary slackness conditions), it is guaranteed to be within a factor away from the optimal. As an example, let's assume that the dual slackness conditions for the k-MST defined above are relaxed (by modifying the second condition) as follows:

Relaxed dual complementary slackness conditions

- For each (v,S) with $v \in V \setminus W, v \in S$: either $y_{v,S}=0$ or $\sum_{e\in\delta(S)} x_e = x_v$.
- For each (v,S) with $v \in S \cap W$: either $y_{v,S}=0$ or $\sum_{e\in\delta(S)} x_e \leq 2$.
- For each $v \in V \setminus W$: either $p_v = 0$ or $x_v = 1$.

If we can construct a pair (x^*, y^*) of an integral feasible primal solution $x^* = (x^*_v, x^*_e)$ and a dual feasible solution $y^* = (y^*_{v,S}, p^*_v, p^*)$ that satisfy the primal slackness conditions and the relaxed dual slackness conditions, then it is easy to see that this primal solution cannot be more than 2 times the value of the optimal solution of LP OPT_{LP}:

$$\begin{aligned}
\sum_{e\in E} c_e x^*_e &= \sum_{e\in E}\left(\sum_{S:e\in\delta(S)}\sum_{v\in S} y^*_{v,S}\right) x^*_e \\
&= \sum_{v\in V\setminus W}\sum_{S\ni v}\left(\sum_{e\in\delta(S)} x^*_e\right) y^*_{v,S} + \sum_{v\in W}\sum_{S\ni v}\left(\sum_{e\in\delta(S)} x^*_e\right) y^*_{v,S} \\
&= \sum_{v\in V\setminus W}\sum_{S\ni v} x^*_v y^*_{v,S} + \sum_{v\in W}\sum_{S\ni v}\left(\sum_{e\in\delta(S)} x^*_e\right) y^*_{v,S} \\
&\leq \sum_{v\in V\setminus W\wedge x^*_v>0}\ \sum_{S\ni v} y^*_{v,S} + 2\cdot\sum_{v\in W}\sum_{S\ni v} y^*_{v,S} \\
&\leq 2\cdot\left((k-w)p^* - \sum_{v\in V\setminus W} p^*_v + \sum_{v\in W}\sum_{S\ni v} y^*_{v,S}\right) \\
&= 2\cdot \mathrm{OPT}_{DUAL} = 2\cdot\mathrm{OPT}_{LP}
\end{aligned}$$

Unfortunately we cannot construct such a pair (x^*, y^*) that will satisfy these relaxed complementary slackness conditions. But we can construct a pair that satisfies them “on average” where the exact definition of “on average” will become apparent later.

The construction of such a pair is iterative. The algorithm starts with an infeasible primal solution and a feasible dual solution; these are the trivial solutions where all primal and dual variables are 0. These initial solutions satisfy the complementary slackness conditions (relaxed or otherwise) trivially. The infeasibility of the primal solution leads to a new dual feasible solution with better (i.e. increased) objective value. In its turn, this better dual solution leads to a new *integral* primal solution with fewer unsatisfied constraints. This "game" between the infeasible primal solution and the feasible dual will continue in the same fashion until the primal solution becomes feasible: the current primal solution is used to determine an improved dual, and the improved dual will determine a "less infeasible" primal. Throughout the running of this procedure two invariants are maintained: the primal solution is always integral and the relaxed complementary slackness conditions will hold. This phase of the algorithm will be called the **grow phase** (because during this phase we "grow" a primal solution).

The integral solution produced by the grow phase of the algorithm is a feasible one, but it may not be a solution to the original problem or it may be pruned down into a cheaper solution for the problem. For example, the problem at hand may ask for a tree as a solution, while the grow phase returns a forest. So the algorithm must go through a **delete phase** that finally produces a solution to the original solution. Note that the delete phase must be designed so that the approximation factor guarantee that worked for the solution produced by the grow phase works also for the final solution.

As in Garg's algorithm, we will use the procedure TREEGROW below for progressively larger values of C (C is a parameter we will define shortly that takes values from 1 to OPT) until the returned tree has at least k nodes (note that the Primal-Dual method we implement does not guarantee that the tree it returns has *exactly* k vertices). Thus the tree needs to be pruned to contain exactly k nodes. The pruning procedure is another source of suboptimality and the reason for using the set of special vertices W. It is described in Section 7.2.3.

Procedure TREEGROW

This procedure is given a number $C \in \mathbb{R}^+$ and the output of the *Preprocessing* part of the algorithm: a rounded graph G, a root r, a set of special vertices W and the value OPT of the optimal k-MST in G. It generates a tree T_C rooted at r that contains W and possibly some other nodes. (The number of nodes is determined by C, as will become clear.) The procedure also generates a feasible potential assignment π_C (for the purposes of our analysis, and not as an output).

Following the general framework of the Primal-Dual method, the algorithm will run in two phases. During the **grow phase** the algorithm grows clusters of vertices while simultaneously it builds a tree for each cluster. Since the first phase may produce a forest instead of a single tree, we need a **delete phase** during which extra edges are pruned.

Grow phase: Initially, we assign each vertex $v \in V \setminus W$ a credit C. The root gets 0 credit. Vertices in W have infinite credit (or credit at least $OPT + 1$). At any moment the algorithm maintains a set of clusters, which we will call *components*. The vertices in each component are connected by a subtree of G, and altogether they form a forest F. At each step the components are either *active* or *inactive* depending on whether they can grow any further or not. The component that contains the root is always inactive.

First we give an informal description of the grow phase. As an example we will use the graph in Figure 7.2. To simplify the exposition we will assume for the moment that the algorithm runs in continuous, rather than discrete time. Initially the forest consists of singletons. The vertices will use their credit to grow a spherical component around them in an effort to reach r. A sphere of radius δ will cost δ amount of credit to the vertex. This growth of components starts simultaneously for all vertices (except for r), it is continuous, and the growth rate is the same for all components (Figure 7.3). As the algorithm proceeds, two components may grow enough to meet each other; for example, the first such meeting will happen when the components growing from the two nearest neighbors in G at the middle of the edge connecting them (if more than one pair of components meet at the same moment, we break ties arbitrarily.) When this happen the two vertices act together: they stop growing their previous components and use their remaining credit to grow a common component around the old ones. Again, in order to grow this component by a "width" of δ, one of the vertices in this component must expend a δ amount of its credit. Every time two components meet, they meet along an edge. Then this edge will be added to the set of picked edges to connect the two trees of picked edges in these components into a tree that spans the vertices in the new component. So it is obvious that the set of picked (or *tight* as we will call them) edges is the set of edges in forest F maintained by the algorithm (Figure 7.4). At some point a component grows enough to meet the root component (that is always inactive and doesn't grow). Then the vertices in this component are connected with r in a tree and do not need to spend any more of their credit for growth. Another event that may take place is that the vertices in a component spend all their credit while growing it. Then this component cannot grow any more and becomes *inactive*. Hence a component is *active* iff throughout its growing phase it always contains a vertex with positive credit. The grow phase of the algorithm ends when all components become inactive, i.e. when all vertices are either connected to the root tree or have no more credit. Note that at the end all vertices in W are connected to the root component, because they always have enough credit to pay until their component is connected to the root component. In our example, there are three inactive components, one of them being the root component (Figure 7.5).

Formally, at each step, the algorithm picks, for every active component S in the current forest, one vertex, say v, whose credit is positive. This selects a dual variable $y_{v,S}$ for S. Having selected a dual variable for each component, the algorithm raises all of them simultaneously and at the same rate, and decreases the credit of the picked vertices at the same rate. (Thus the credit of the vertices pays for the dual increases.) Initially all dual variables are equal to 0. The algorithm starts by raising dual variables $y_{v,\{v\}}$, for all $v \neq r$. At some point two components meet along an edge e. Then this edge becomes tight (i.e. constraint $\sum_{S:e\in\delta(S)} \sum_{v\in S} y_{v,S} \leq c_e$ in $DUAL$ becomes an equality) and we pick it (i.e. we make $x_e = 1$ in LP). Whenever two components S_1, S_2 meet, the algorithm stops raising the variables y_{v_1,S_1}, y_{v_2,S_2} for $v_1 \in S_1, v_2 \in S_2$. Instead it picks a vertex $v_3 \in S_1 \cup S_2$ with positive credit and starts raising variable $y_{v_3,S_1\cup S_2}$ using v_3's credit to pay for the raise. If v_3's credit is exhausted, the algorithm finds another vertex v_4 with positive credit, stops

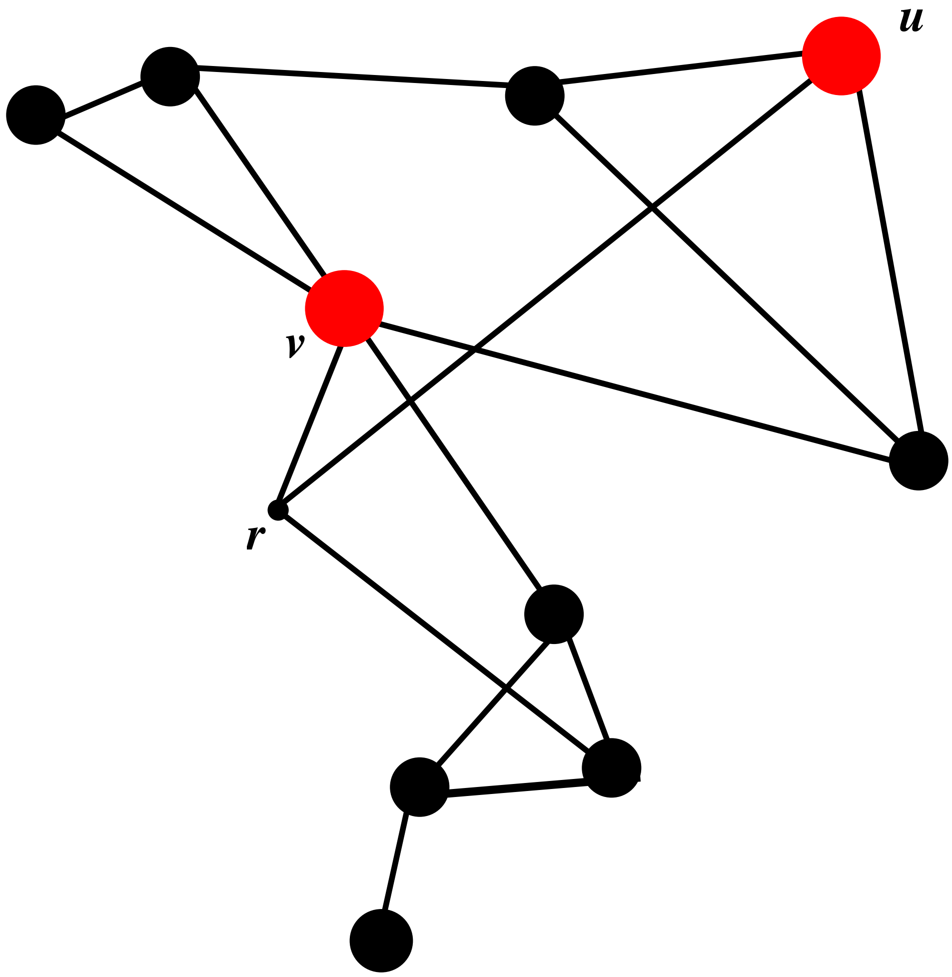


Figure 7.2: The graph G with root vertex r and $W = \{v, u\}$.

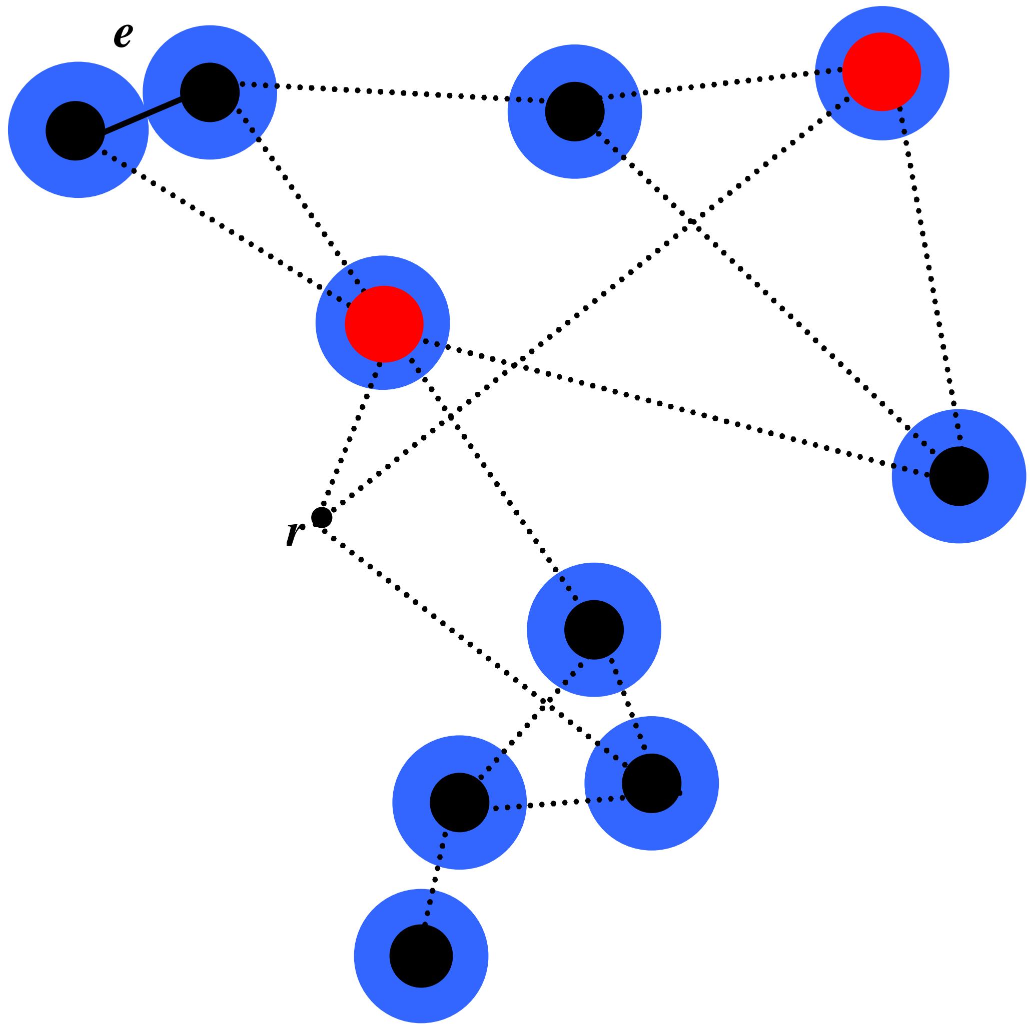


Figure 7.3: The components grow and edge e becomes tight.

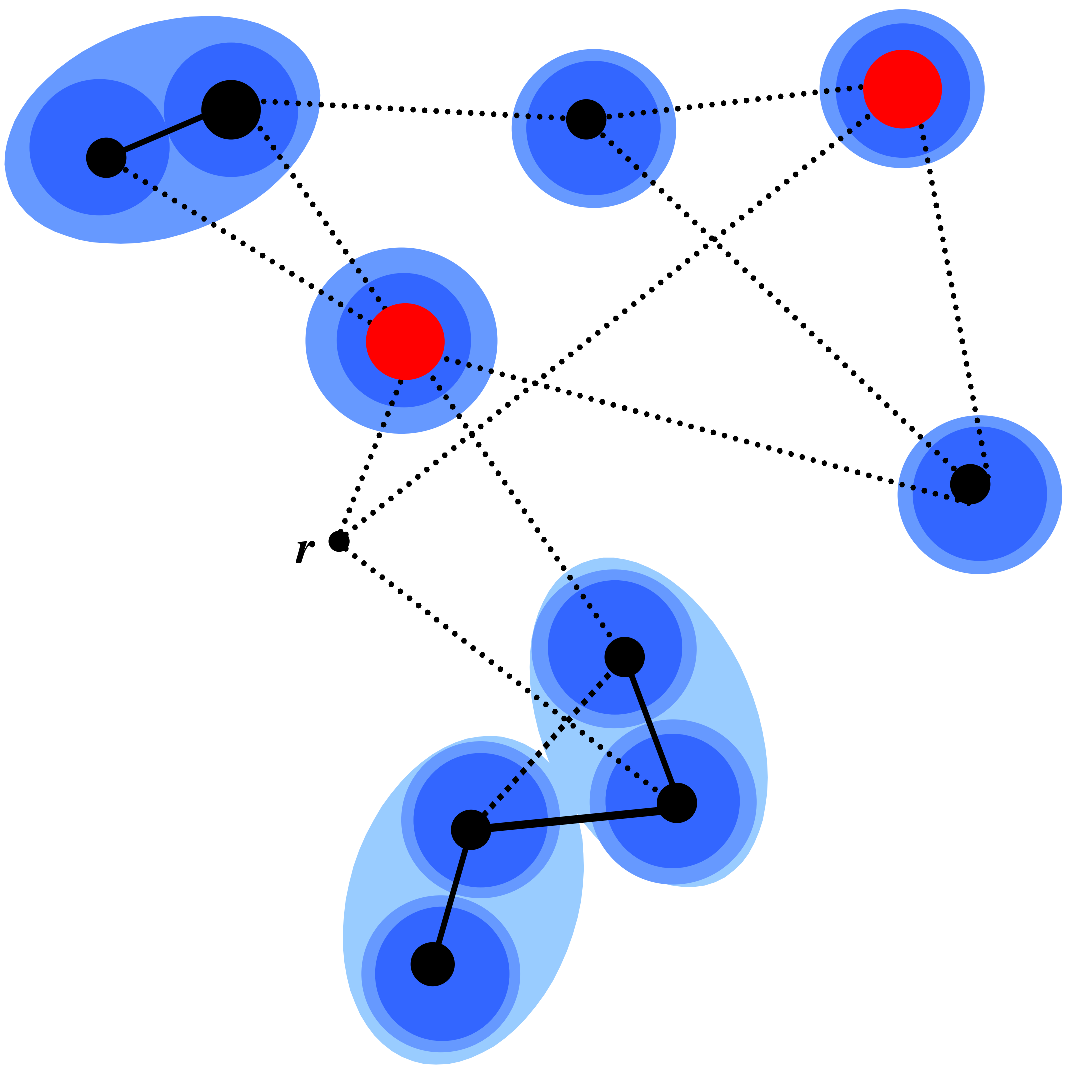


Figure 7.4: Components continue to grow. Solid lines denote tight edges. Tight edges in each component form a tree.

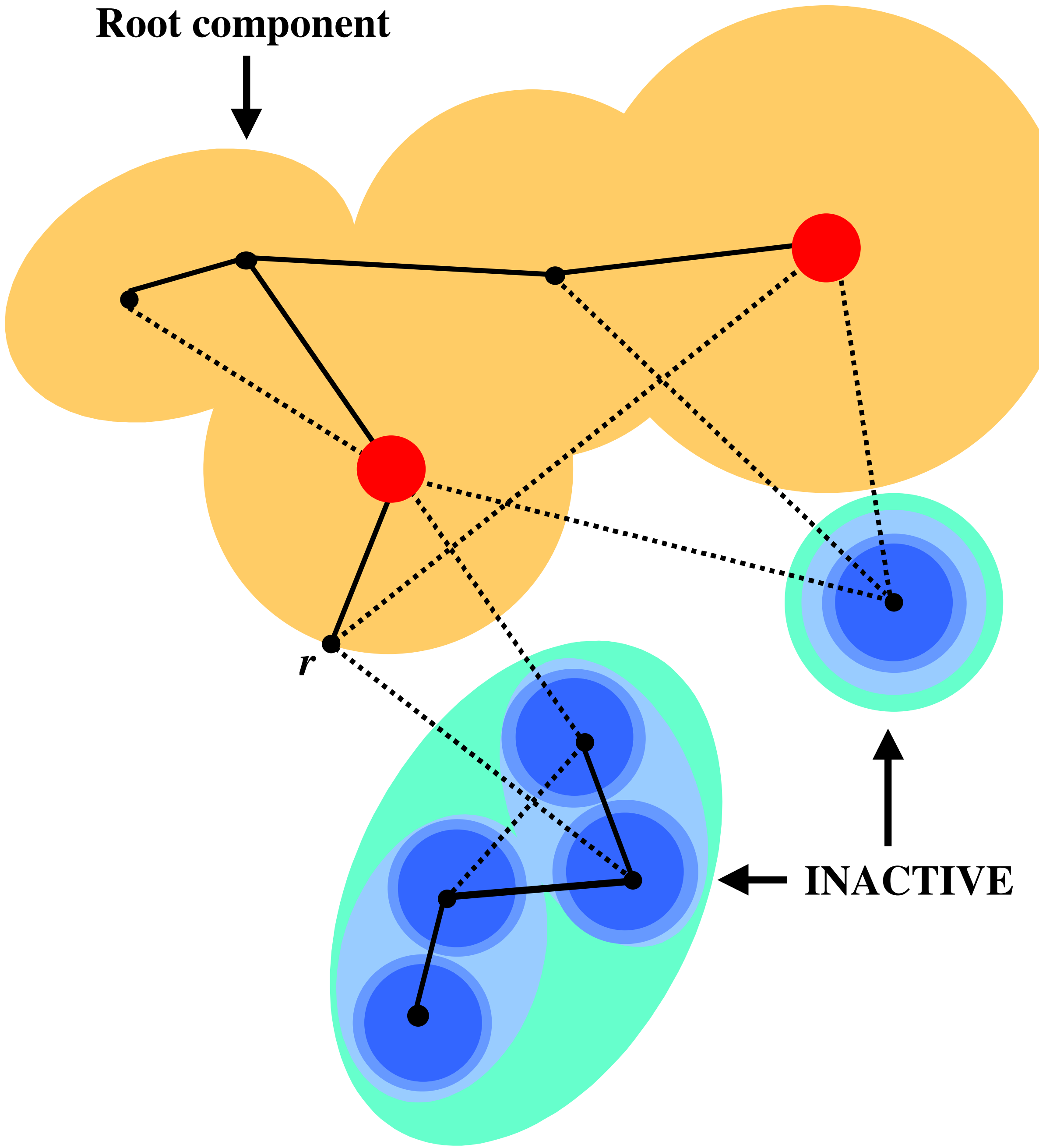


Figure 7.5: The end of the grow phase of the algorithm.

raising $y_{v_3,S_1\cup S_2}$, and starts raising variable $y_{v_4,S_1\cup S_2}$. If it cannot find any vertex with positive credit in the component, the component becomes inactive.

The dual increase rule can be also viewed as increasing z_S at a uniform rate for all active components. (See (7.2) for definition of z_S.) This is because the algorithm increases $y_{v,S}$ for only one vertex $v \in S$ at any step. During this process, primal variables x_v and dual variables p, p_v can be set so that the second (relaxed) complementary slackness conditions holds "on average" (as explained later in the analysis). Also notice that although our description was based on continuous time, the algorithm can be easily run in discrete steps, where each step is an 'event' (two components meet, or a component becomes inactive) and the time until the next event (i.e. by how much the components will have to grow) can be calculated in polynomial time. Since the number of events is linear in $n = |V|$, this phase of TREEGROW will run in polynomial time.

Delete Phase: At the end, we delete all inactive components that are not on a path from an active component to the root (i.e. inactive components that are leaf subtrees of the root component). This process is done in a bottom-up fashion: first we prune all the leaf inactive components (except the root component, if it is a leaf), then we are left with a new (pruned) tree, we again prune the inactive leaves and so on, until we are left with a tree where all leaves are active components with the possible exception of the root component. The exact reasons for the delete rules will become apparent in the analysis of the algorithm, but the intuition is the following: An inactive leaf component is connected to the root component via an active component that has grown enough to touch it. But then the credit in the active component has been used not only to connect this component to the root, but to connect the inactive leaf component to the root as well. This turns out to be an overkill because the same credit has paid for (potentially many) more tight edges than its worth. By deleting the inactive leaf components we will be able later to argue that there is an equivalence between the credit spent and the cost of the tree.

Output of TREEGROW The procedure will output the tree T_C of the (pruned) component containing the root.

It is not necessary that T_C will span exactly k vertices. It is obvious that the bigger the initial credit C is, the bigger the root component will be. In the third part of the algorithm we will compute a tree that spans *exactly* k vertices.

Constructing a feasible potential assignment

TREEGROW outputs tree T_C. As an aside we can use it to construct a potential assignment π_C whose feasibility is witnessed by the dual values $y_{v,S}$ constructed by the algorithm. Although this assignment is not required as an output, it will help in the analysis of the algorithm. For each vertex v whose final credit is 0 (note: $v \notin W$), $\pi_C(v)$ is C. This implies that vertices in inactive components (other than the root component) will get a potential assignment C. Note that any vertex whose final credit is positive (the vertices in W included) must be in the root component, and before it joined the root component it was always in an active component. For such vertices we do the following: We start decreasing the initial credit of all such vertices uniformly and run TREEGROW with the new initial credit. We continue the decrease until we notice that one of these vertices, say u, would now finish the algorithm with zero final credit. Then $\pi_C(u)$ is set to the (new) initial credit of u and we don't decrease its initial credit anymore. We continue in this manner until we ensure that all vertices finish the algorithm with zero final credit.

By construction, if the algorithm is started with initial credits given by π_C, then its behavior is the same as when all the vertices not in W got an initial credit C. Moreover, when TREEGROW ends, all vertices *not* in inactive components will have zero final credit. Thus we have the following lemma:

Lemma 9: *The potential assignment π_C is feasible.*

Proof: The proof follows from two observations. First, the potential assignment (i.e., initial credit values) are being used to exactly pay for the increase of the dual variables $y_{v,S}$, so $\pi_C(v) = \sum_{S:v\in S} y_{v,S}$. Second, the edges are added to the forest as soon as they become tight, so during the run of TREEGROW we always maintain $\sum_{S:e\in\delta(S)} \sum_{v\in S} y_{v,S} \le c_e$ for all $e \in E$. Hence π_C complies with Definition 6. □

Let $k_C = |T_C|$. The next lemma follows from the construction of π_C:

Lemma 10: *$W \subseteq T_C$ and T_C contains the $k_C - |W|$ vertices in $V \setminus W$ with the smallest potentials. All vertices outside T_C have potential C.*

Proof: The vertices that did not get included in T_C were among the ones who expended all their credit C on growing their dual values before the end of the algorithm. Hence their initial credit was not changed at the end, so their potential is C. Vertices in $T_C \setminus W$ have potential at most C, so they are the vertices with the $k_C - |W|$ smallest potentials. □

Lower & upper bounds

Before we continue with the third (and last) part of the algorithm, we relate the cost of T_C produced by TREEGROW with OPT. Since π_C is feasible, Lemma 7 implies that OPT is lower bounded by the sum of potentials of the nodes in W plus the $k - |W|$ smallest potentials in *any* feasible potential assignment. Hence we have the following:

Corollary 1: *If W is contained in the optimum k-tree, then OPT is at least the sum of the potentials of vertices in W and the sum of the $k - w$ smallest potentials in $V \setminus W$, for any feasible potential assignment.*

Corollary 1 can be combined with Lemma 10 to prove

Lemma 11:

$$\sum_{v\in T_C} \pi_C(v) \le OPT.$$

For an upper bound we follow the standard techniques of upper bounds in primal-dual schemes (cf. [26], [17]):

Lemma 12:

$$\sum_{e \in T_C} c_e \leq 2 \sum_{v \in T_C} \pi_C(v).$$

Proof: By the way the potentials were defined, we have

$$\sum_{v \in T_C} \pi_C(v) = \sum_{S \subseteq T_C} \sum_{v \in S} y_{v,S} = \sum_{S \subseteq T_C} z_S$$

where S in the above expressions denotes components formed during the running of the algorithm. We wish to upperbound

$$\begin{aligned} \textstyle\sum_{e \in T_C} c_e &= \textstyle\sum_{e \in T_C} \sum_{S: e \in \delta(S)} z_S \\ &= \textstyle\sum_{S \subseteq T_C} z_S |T_C \cap \delta(S)| \end{aligned}$$

So the Lemma will be proved if we could show

$$\sum_{S \subseteq T_C} z_S |T_C \cap \delta(S)| \leq 2 \times \sum_{S \subseteq T_C} z_S \tag{7.3}$$

We prove (7.3) by induction for every step of the algorithm. Initially this is true since all the z_S's are zero.

At any stage of the algorithm, let H be a tree whose vertices are the components formed till this stage, and whose edges are the edges of T_C that are missing from these components (i.e. edges that have not become tight yet but will connect them at the end). Let N_a, N_i denote the sets of active and inactive components at this stage respectively. Recall that the algorithm raises all z_S's for current S's at the same rate. When each z_S for the active components rises by ϵ, the algorithm increases the LHS of (7.3) by $\epsilon \sum_{v \in N_a} \deg_v$ and the RHS by $2\epsilon |N_a|$ ($\deg_v$ is the degree of component v in H). It suffices to show that the average degree of the components in N_a is at most 2.

Owing to the **Delete** phase, the leaves in H cannot be inactive components, except possibly the component containing r. Thus all components $S \in N_i$ except possibly one have degree at least 2, and so the sum of the degrees of components in N_i is at least $2|N_i| - 1$. The sum of the degrees of components in tree H is $2|H| - 2$, so the sum of the degrees of components in N_a is at most $2|H| - 2|N_i| - 1 = 2|N_a| - 1$. Hence the average degree of components in N_a is at most 2 (and the relaxed dual complementary slackness condition holds "on average").

□

All the above hold for any value of C. We will run TREEGROW for C ranging from 0 to OPT. The increase step will depend on the 'tightest' edge of those that are not tight with the current value of C. That is, we increase C by the minimum amount that will make one more edge tight. There is some value C_0 that is a threshold in the following sense: if we give initial credit $C_0 - \delta$ (where δ is infinitesimally small), the tree $T_{C_0-\delta}$ has $k_1 < k$ vertices, but if we give initial credit C_0, the tree T_{C_0} has $k_2 \geq k$ vertices[1]. These two trees will be the output of the second part of the algorithm, together with the feasible potential assignments described above. The third (and last) part will use these two trees to output a tree with exactly k vertices.

7.2.3 Producing a tree that spans exactly k vertices

In the second part of the algorithm we call procedure TREEGROW for increasing values of initial credit C, ranging from 0 to OPT, where the increase step is determined as explained above0. From the description of the **Grow** phase it is apparent that the tree in the root component will be bigger for greater values of C. Obviously, for some initial credit C_0, trees $T_{C_0-\delta}, T_{C_0}$ (i.e. the trees produced when the initial credit is $C_0 - \delta$ and C_0 respectively, for

[1]It may be the case that for $C = 0$ the tree produced by TREEGROW has more than k vertices. In this case it is obvious that all the leaves are vertices in W (because any other vertex would become inactive as soon as the grow phase starts, and would be pruned in the delete phase if it were a leaf). But then there would be another set W that instead of one of these leaf vertices it contains its closest ancestor not in W, and produces the same tree with one vertex less. If we continue in this fashion we will eventually find a W for which the tree produced when $C = 0$ has *exactly* k vertices. Then the rest of our analysis applies.

infitesimally small δ^2), are such that $|T_{C_0-\delta}| = k_1$ and $|T_{C_0}| = k_2$, with $k_1 < k \leq k_2$. The third part of the algorithm will append some of the extra vertices in T_{C_0} to $T_{C_0-\delta}$ in order to produce a tree that spans exactly k vertices. Since it is the almost the same as Garg's algorithm [23] we will follow his exposition before pointing out where our improvement occurs.

Let $\hat{T}_{C_0-\delta}$, $\hat{T}_{C_0}$ be the root component trees for these initial credit values. Then we can prove that $\hat{T}_{C_0-\delta}$ and $\hat{T}_{C_0}$ have similar structure.

Lemma 13: *Let S be a set of vertices such that $z_S > 0$ when we run the algorithm with initial credit C_0. Then either all the vertices of S do not belong in the final root component or they occur contiguously in both root component trees $\hat{T}_{C_0-\delta}$ and $\hat{T}_{C_0}$.*

Proof: Initial credits $C_0 - \delta$, C_0 differ only by δ. Since δ is infinitesimally small, either S is a subcomponent of active components throughout the running of the algorithm for both initial credit values (and therefore it is connected to the root component), or it will become part of an inactive component not connected to the root component for both values. In the first case the vertices in S occur contiguously in $\hat{T}_{C_0-\delta}$ and $\hat{T}_{C_0}$ because S was a component at some point during the algorithm, and its vertices were connected by a subtree of the final root component. □

We will concentrate our attention to those S that participate in $\hat{T}_{C_0-\delta}$ and $\hat{T}_{C_0}$. Let S be such a set. Then the vertices of S appear contiguously in $\hat{T}_{C_0-\delta}$ and $\hat{T}_{C_0}$. Hence we can replace the edges induced by S in $\hat{T}_{C_0-\delta}$ with the edges induced by this set in $\hat{T}_{C_0}$ without increasing the cost of the tree (because the difference between the initial credit values was infinitesimal the only possibility for the algorithm to pick different edges in the two cases is the case of a tie, i.e. when two or more edges between two subcomponents of S become tight simultaneously). Hence if we start from $\hat{T}_{C_0-\delta}$ and apply the above modification on

[2] Although we assume that δ is infinitesimally small, our analysis will hold for δ equal to the difference between C_0 and the previous value of initial credit used to run TREEGROW.

active components S in the order they were created by the algorithm, we will eventually get $\hat{T}_{C_0}$. In this way we have created a sequence of trees, and each element of this sequence differs from the next by a pair of edges (short inductive proof: The difference between two consecutive elements of the sequence is due to the modification of an active component S. But this component was created when two other components $S_1,\ S_2$ were connected. These two are earlier components, so they have already been modified. Hence the only difference between the two consecutive trees is the edge that connects S_1 and S_2 to create S.) Note that all the above hold for the root component trees created by the Grow phase in TREEGROW. Since $|T_{C_0-\delta}| < k$ and $|T_{C_0}| \geq k$, there must be two trees in the sequence $\hat{T}_-,\ \hat{T}_+$ that differ only by an edge, and after the Delete phase the new trees $T_-,\ T_+$ span $|T_-| < k$ and $|T_+| \geq k$ vertices respectively.

In what follows we are going to work with $\hat{T}_-$ and $\hat{T}_+$. Let $e_- \in \hat{T}_-, e_+ \in \hat{T}_+$ be the pair of edges that trees $\hat{T}_-, \hat{T}_+$ differ in and let S_1, S_2 be the two components of S between which these edges run with S_1 being the component closer to the root. Figure 7.6 summarizes the structure of $\hat{T}_-$ and $\hat{T}_+$. Also let $k_1 = |T_-|, k_2 = |T_+|$ and $k_1 < k \leq k_2$. We call the extra vertices of T_+ that do not belong to T_- *new vertices*. The remaining vertices of T_+ (which also belong to T_-) will be the *old vertices*. Let s be the number of new vertices, where $s \geq k_2 - k_1$. The similarity between the root component trees $\hat{T}_-, \hat{T}_+$ implies the following lemma:

Lemma 14: *In the tree T_+ all new vertices occur contiguously while the old vertices form at most two contiguous sets.*

Proof: The new vertices belong to a component that has enough initial credit in it to attach itself to the root component when the initial credit is C_0, but it was an inactive leaf that was deleted in the Delete phase when the initial credit was $C_0 - \delta$. So the new vertices occur contiguously. There are two possible ways for the new vertices component to attach itself to T_+ and they are both depicted in Figure 7.7.

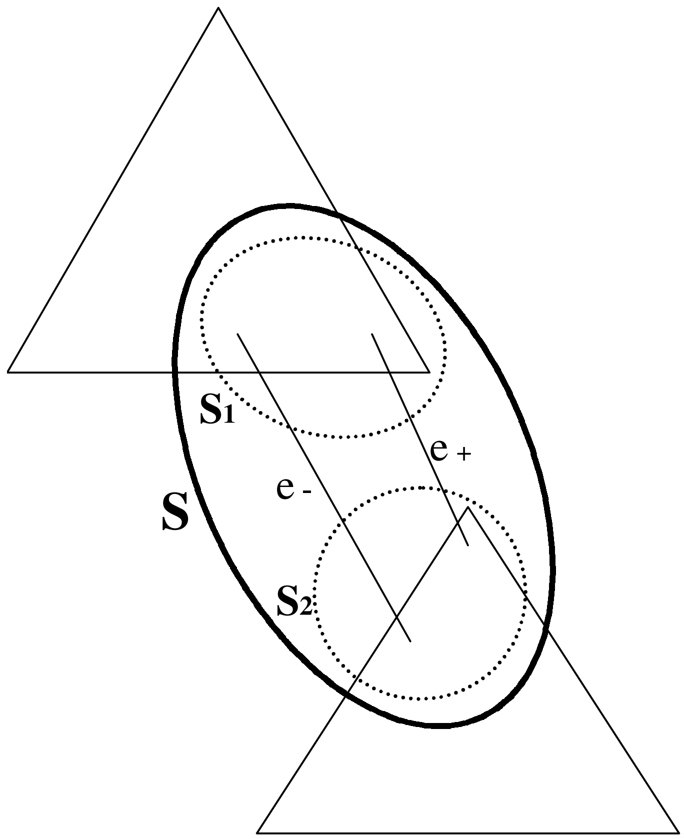
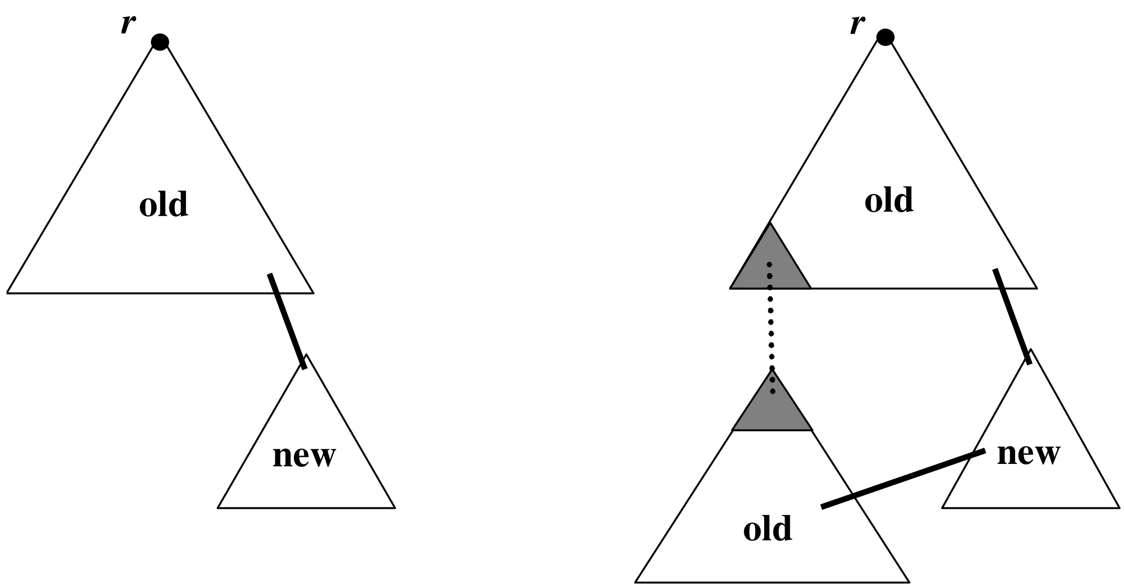


Figure 7.6: The trees $\hat{T}_-, \hat{T}_+$.

$\square$

All new vertices belong to component S, since the component that contains them also contains edge e_+. If X_1, X_2 are the two contiguous sets of old vertices in T_+, with $r \in X_1$, then the two subcomponents S_1, S_2 of S contain all the new vertices and S_1 is contiguous with X_1 and S_2 is contiguous with X_2. Notice that X_2 may be empty (case (a) in Figure 7.7). This is a case covered by our analysis, so we will assume that $X_2 \neq \emptyset$. Also note that $W \subseteq X_1 \cup X_2$. Since $|X_1 \cup X_2| = k_2 - s$ all we have to do is to pick $k - k_2 + s$ of the new vertices. First we pick new vertices from S_1. If they are not enough we will continue picking



(a) (b)

Figure 7.7: (a) The new vertices component is a leaf component in T_+. (b) In tree T_+ the new vertices component replaces an edge (dotted line) in connecting a component of old vertices to the root. Notice that some of the old vertices that belong to T_- (gray areas) may not belong to T_+, because by replacing the edge with the new component, they become inactive *leaves* and are deleted during the Delete phase.

new vertices from S_2. Wlog we will assume that we pick all new vertices in S_1 and the rest are picked from S_2. The vertices in $X_1 \cup S_1$ occur contiguously in T_+. Let T_1 be the tree induced over these vertices by T_+. The new vertices from S_2 will be picked so that they occur contiguously with X_2 in T_+. Let T_2 be the tree induced over these vertices by T_+. The solution produced by the algorithm is the trees T_1, T_2 connected by the edge of minimum cost that connects them. We will prove that one can pick the new vertices so that the cost of T_1, T_2 is at most twice the sum of the potentials given to their vertices by the potential assignment π_{C_0} defined in the previous section. We will prove that T_1, T_2 also contain the vertices with the $k - |W|$ smallest potentials given by π_{C_0}, so the sum of potentials is also a lower bound of the optimal solution, by Corollary 1. Thus $\text{cost}(T_1) + \text{cost}(T_2) \leq 2 \cdot \text{OPT}$. Because of the special properties of W the edge we will use to connect T_1 to T_2 cannot cost more that $\epsilon \cdot \text{OPT}$. This is exactly the point of our improvement over Garg's algorithm. The overall approximation factor of the algorithm is $2 + \epsilon$.

Now we describe the algorithm for picking the new vertices. We would like to pick a certain number of new vertices from S_2. Recall that the new vertices of S_1 have already been picked[3]. Let T be the tree induced by T_+ on the vertices in $X_2 \cup S_2$, and M the set S_2. The new vertices are going to be picked from M. At each step of the algorithm, M or T will shrink until T contains exactly the number of new vertices needed. The iterative algorithm is outlined in Figure 7.8. The algorithm uses the notion of *inactive leaves* to refer to inactive components created during TREEGROW that are (either as a whole or a contiguous part of them) leaf components of T.

The algorithm can be shown to maintain four invariants that are essential for its correctness and its cost analysis.

Invariant 1: *All vertices in T_2 picked by the algorithm occur contiguously.*

Proof: It suffices to prove that every time M is pruned, the pruned (and thus unpicked)

[3] If S_1 contains more that enough new vertices, then what we say for S_2 apply to S_1.

Input : Tree T_1. Sets X_2, S_2.

1. Let $T := T_+ \cap \{X_2 \cup S_2\}$ and $M := S_2$.
2. **While** T doesn't contain the number of vertices needed **do**:
   - If C is an inactive leaf component of T with $C \subseteq M$ (Figure 7.9(a)) then
     - If $M \setminus C$ contains enough new vertices then move to next iteration with $T := T \setminus C, M := M \setminus C$
     - If $M \setminus C$ doesn't contain enough new vertices then move to next iteration with $T := T, M := C$.
   - If T doesn't have an inactive leaf in M, let C_1, C_2 be the two components forming M, with C_1 being the component connected to $T \setminus M$ (Figure 7.9 (b)).
     - If C_1 contains enough new vertices then move to next iteration with $T := T \setminus C_2, M := C_1$.
     - If C_1 doesn't contain enough new vertices then move to next iteration with $T := T, M := C_2$.
3. Connect $T = T_2$ to T_1 with the cheapest edge. Let $\hat{T}$ be the resulting tree spanning exactly k vertices.

Output : Tree $\hat{T}$.

Figure 7.8: Producing a tree spanning exactly k vertices

vertices are always a leaf subtree of T. This is clearly the case when T does not have an inactive leaf that is a subset of M; in this case the only vertices that may be pruned are the vertices of component C_2, which is clearly a leaf subtree of the current T (Figure 7.9 (b)).

In case T has an inactive leaf C with $C \subseteq M$, the only vertices that may be pruned are the vertices of C. The only complication arises when $T \setminus M = \emptyset$ and C contains *all* old vertices of T. Then it may be the case that the extra vertices picked from $M \setminus C$ are not contiguous with the old vertices in C. But C cannot contain all old vertices of T because it is an inactive leaf and it would have been deleted from T_- during the Delete phase (and thus T wouldn't contain any old vertices, a contradiction). $\square$

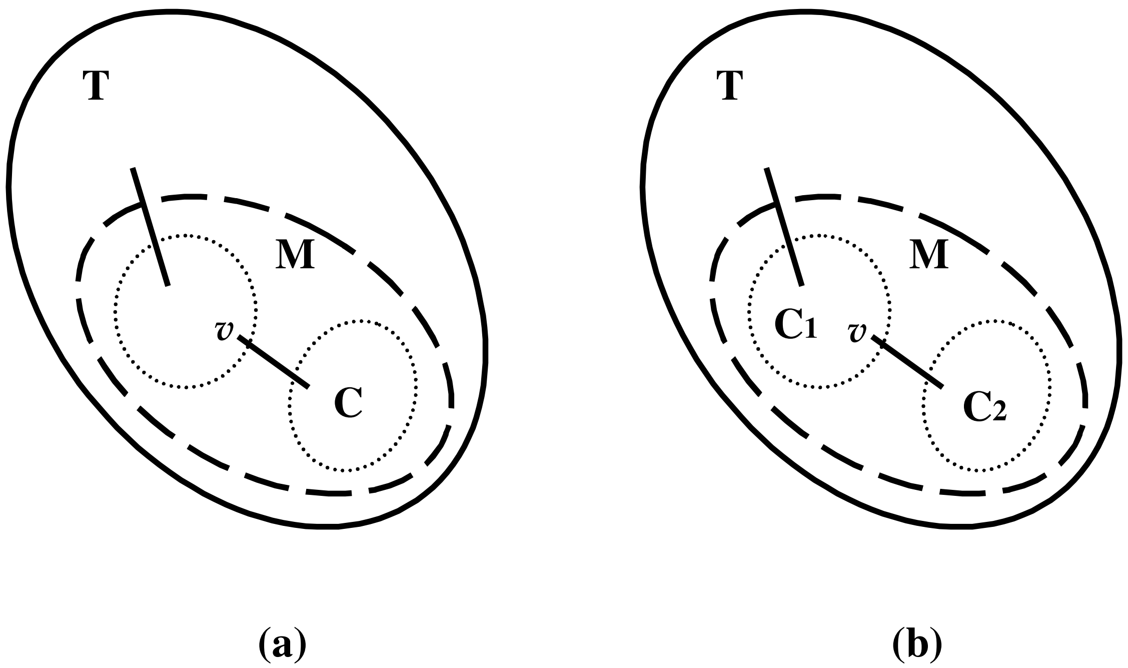


Figure 7.9: (a) C is an inactive leaf of T. (b) M is formed by components C_1, C_2 that are not inactive leaves of T at the current iteration of the algorithm.

Invariant 1 ensures the correctness of the algorithm. The second invariant ensures that M is always a **component** of T in the following sense: M is either a component as defined in the previous section, or it is the intersection of such a component with T.

Invariant 2: *Set M is a component.*

Proof: The proof is inductive. Initially $M = S_2$, so the invariant holds. If T does not contain an inactive leaf $C \subseteq M$, the new M is going to be either C_1 or C_2 which are components. If T contains an inactive leaf $C \subseteq M$, then the new M is either C (which is a component) or $M \setminus C$, which is the intersection of component M with the new tree $T \setminus C$

(and therefore a component, according to the new definition). □

As defined in the previous paragraph, the components of T at any stage of the algorithm are either original components formed by TREEGROW or the intersection of such components with T. We would like to use the same line of analysis we used in Section 7.2.2 to prove that T continues to have bounded cost, just like tree $\hat{T}_{-}$. To this end, we define an extension of the active components in TREEGROW for the new notion of a component:

Definition 8: *A component C of T* pays for itself *if* $\sum_{S\subseteq C}\sum_{v\in S} y_{v,S} \leq \sum_{v\in C} \pi_{C_0}(v)$.

Invariant 3: *The components that cannot pay for themselves are supersets of M.*

Proof: Again the proof is inductive. Notice that all original components that intersect X_2 were active (or their intersection would had been deleted from X_2 during the Delete phase), so they can pay for themselves. The rest of the components have to intersect S_2. If someone of them was active throughout the run of the algorithm it can pay for itself. If it was inactive, then it has to contain the whole S_2, otherwise its intersection with S_2 would had been deleted during the delete phase and it wouldn't be a part of T_{+}. Hence initially the components that cannot pay for themselves contain $M = S_2$.

For the inductive step, notice that at every step the same vertices are pruned from both M and the components that couldn't pay for themselves, so the restriction of the components that couldn't pay for themselves to the new T still contain the new M. The only case where a *new* component that cannot pay for itself is created by the algorithm is when the new T is $T \setminus C_2$ and the new M is C_1. Then component C_1 may not be able to pay for itself anymore, but in this case C_1 is itself the new M and the invariant holds.

□

The fourth (and last) invariant we maintain is the following:

Invariant 4: *At every step every inactive leaf of T intersects both the current M and all the other inactive leaves of T.*

Proof: Initially every inactive leaf of T has to contain the vertex incident to e_+, otherwise it would had been deleted during the Delete phase. This vertex also belongs to $M = S_2$, so the invariant holds.

When T is modified to $T \setminus C$ or $T \setminus C_2$, the rest of M (that is $M \setminus C, C_1$ respectively) may become a new inactive leaf or part of new inactive leaves. All of these have to contain vertex v in Figure 7.9 (otherwise they would had been deleted), and all of the older inactive leaves that are still present have to contain v (for the same reason). Since v also belongs to the new M in both cases the invariant holds.

The pruning of M can also result into new inactive leafs for T. If M was contained in an inactive leaf before the pruning, it it still contained in the new (pruned) inactive leaf since we prune the same vertices from both of them. The other case that may arise is the case of M containing an inactive leaf before the pruning, and this leaf is disjoint from the new (pruned) M. This may happen either when the new M is C, for some inactive leaf C, or when the new M is C_2. In the first case the inactive leaf has to be a leaf in $M \setminus C$ for the unpruned M. But, from the inductive step, this inactive leaf was an inactive leaf in the previous step of the algorithm, together with C, so they had to intersect each other, a contradiction. In the second case the inactive leaf has to be a subset of C_1. This leads to a contradiction, because in this case we have assumed that there are not inactive leaves that are subsets of the unpruned M.

$\square$

Upper & lower bounds for the final solution

Let $T = T_2$ and M be the final tree and set produced by the algorithm. We want to prove an analog of Lemma 12 for the trees T_1, T_2 we produced.

Lemma 15: *The total cost of T_1, T_2 is at most the sum of the potentials of their vertices as given by π_{C_0}.*

Proof: The proof is essentially the same as the proof of Lemma 12. It is inductive on the steps of TREEGROW and it distinguishes two cases: the steps before component S was formed by the merge of S_1, S_2, and the steps after S was formed.

The complication in the first case is the new notion of component; recall that T is always a collection of components in the original sense or *intersections* of original components with its vertices. Note that if all active components of T can pay for themselves and T has only one inactive leaf, then the proof of Lemma 12 carries through. But indeed T has only one inactive leaf, because all its inactive leaves would have to intersect each other, by invariant 4. Also any active component that cannot pay for itself contains M, by invariant 3. M is itself a component, by invariant 2, so during the Grow phase of TREEGROW can be contained only in one active component. Therefore there is only one active component that cannot pay for itself. Moreover, T cannot have both such an active component and an inactive leaf because they will intersect (the active component would contain M, by invariant 3, and the inactive leaf would have to intersect M, by invariant 4). So $T = T_2$ contains either at most one active component that cannot pay for itself or at most one inactive leaf. On the other hand, T_1 has at most two inactive leaves: the root component and (possibly) the component that contains the endpoint of e_+. Hence in the current step of TREEGROW $T_1 \cup T_2$ has either at most three inactive leaves or at most two inactive leaves and an active component that cannot pay for its growth. The result in both cases is the same: the analysis of Lemma 12 still carries through because $T_1 \cup T_2$ induces a forest of two trees on the current components.

In the second case, S has been formed and $T_1 \cup T_2$ together with e_+ induce a tree on the current components. This tree has at most one inactive leaf (the root component) and the analysis is exactly the same as in Lemma 12 (note that the only subcomponents of S that do not participate in the analysis for $T_1 \cup T_2$ are inactive leaves that didn't participate in the analysis for the whole T_+ in Lemma 12 as well).

□

Lemma 15 didn't account for the cost of the edge we use to connect T_2 to T_1. The crucial observation is that W is always either a subset of the old vertices of T_1 or e_- has length less than $\epsilon \cdot \text{OPT}$. Thus the cheapest edge that connects T_2 to T_1 cannot be more costly than $\epsilon \cdot \text{OPT}$. Hence we have proven

Lemma 16: *The tree $\hat{T}$ produced by the algorithm spans exactly k vertices and its cost is upper-bounded by*

$$cost(\hat{T}) \leq 2 \cdot \sum_{v \in \hat{T}} \pi_{C_0}(v) + \epsilon \cdot OPT \tag{7.4}$$

Also the tree the algorithm produced contains all the old vertices of T_-, so it contains all vertices with potential assignment less than C_0, together with all the vertices in W. The rest of the vertices have a potential assignment of C_0 by π_{C_0}. Hence $\hat{T}$ contains W together with the vertices that have the $k - |W|$ smallest potentials under feasible potential assignment π_{C_0}. Corollary 1 implies

Lemma 17: $\sum_{v \in \hat{T}} \pi_{C_0}(v) \leq OPT$.

Lemmata 16 and 17 are enough to show the main result of this chapter:

Theorem 13: *The algorithm described above produces a tree $\hat{T}$ that spans exactly k vertices of graph G, its cost is*

$$cost(\hat{T}) \leq (2 + \epsilon) \cdot OPT$$

and its running time is $n^{O(\frac{1}{\epsilon})}$.

Proof: Note that all the steps of the algorithm after the guessing can be implemented in polynomial time. Guessing multiplies this running time by a factor $n^{O(\frac{1}{\epsilon})}$, as explained in Section 7.2.1. □

7.3 The metric k-TSP problem

The metric k-TSP problem was defined in Section 6.1 as follows: given a graph $G = (V, E)$ with non-negative edge costs that satisfy the triangle inequality, we wish to find a cycle of minimum cost that visits exactly k vertices; we call such a cycle a k-tour. Again wlog we will consider the rooted version of this problem.

For the case of k-MST we used the fact that the final potential assignment π_{C_0} was *feasible*, namely any tree T rooted at r and including the preselected vertices in W, had cost at least as much as the sum of the potentials of its vertices. This means that the sum of the $k - |W|$ smallest potentials plus the potentials of the vertices in W were a lower bound for the cost of the edges in the optimal k-tree. For tours, any feasible assignment has an even stronger property:

Lemma 18: *For any tour P and any feasible potential assignment, the cost of P is at least twice the sum of the potentials of the vertices in P.*

Proof: Notice that for a cycle, every component we grow pays for *two* edges of the cycle (since a component that is entered by the cycle must be left by it as well).

□

Hence, if T denotes the k-tree constructed by the algorithm, OPT_{tour} the cost of the minimum k-tour and OPT_{tree} the cost of the minimum k-tree, we have:

$$2 \sum_{v \in T} \pi_C(v) \leq OPT_{\text{tour}} \tag{7.5}$$

since T contains the vertices with the $k - |W|$ smallest potentials together with the vertices of W.

Now we construct a tour out of T, by duplicating its edges and short-circuiting an Eulerian walk on it, incurring an approximation factor of 2. Hence, if C is the tour constructed in this way, we have

$$
\begin{aligned}
\mathrm{cost}(C) &\leq 2 \cdot \mathrm{cost}(T) \\
&\leq 2 \cdot (2 \sum_{v \in T} \pi_{C_0}(v) + \epsilon \cdot OPT_{\text{tree}}) \\
&\leq 2 \cdot (OPT_{\text{tour}} + \epsilon \cdot OPT_{\text{tour}}) \\
&= (2 + \epsilon') \cdot OPT_{\text{tour}}
\end{aligned}
$$

for any $\epsilon' > 0$. The running time is essentially the running time of the algorithm for k-MST , i.e. $n^{O(\frac{1}{\epsilon'})}$.

Chapter 8

Approximation of the LFU policy for Web Caching

The last part of this thesis is dedicated to the study of the Least Frequently Used (LFU) replacement policy for Web caching. Under certain assumptions we will give both theoretical bounds on various parameters for this policy, as well as practical implementations of LFU that achieve almost as good (and in some cases even better) results as the theoretical bounds.

8.1 Web and caching

The past decade has witnessed a tremendous progress of the Internet, a global network that connects millions of users around the globe. This World Wide Web (WWW) provides not only the means for communication between people but the deployment of a wide range of services to end-users as well. Services from distant learning to advertising to electronic commerce (e-commerce) have become part of everyday life. The huge amount of data that need to be transmitted through the Internet together with the exponential growth of Web users (clients) lead to significant congestion in the network, leading to long access delays, absence of Quality-of-Service (QoS), and low penetration of services to the electronic customer base. The problems of congestion in the backbone network has elevated the bandwidth available for data transfer to a critical commodity in short supply.

One of the methods used to avoid congestion in the network and decrease the access latency experienced by the users is caching. Caching has been used for a long time in all computing systems. In processors caches store data that are being accessed repeatedly due to temporal locality phenomena as well as neighboring data to exploit spatial locality. A similar locality phenomenon has also been observed in the accesses of groups of users over the Internet [20] [14]. It has been observed that objects (sites) on the Web are been accessed in a non-uniform uniform fashion; more precisely, it seems that accesses over the Web can be modeled using a Zipf or Zipf-like distribution [20] [18]. Thus the potential and improved performance of Web caching has led to the development and deployment of systems that implement caching mechanisms [3] [19].

Given a calculation or estimation of the probabilities of access for Web objects (sites or pages), one can view the Web as a system composed of servers, clients and the Internet which is the network interconnecting them. This environment is a dynamic system where clients inject requests to the servers and receive responses. In order to develop caching strategies, i.e. location and sizing of the caches as well as caching policies, we need to analyze the behavior of the system.

8.2 Zipf's distribution and the network model

We analyze a simple environment, as the one shown in Figure 8.1. In this environment, an enterprise network (or LAN) is connected to the Internet through a gateway, which also serves as a cache (for example, in a typical environment, the gateway could be a firewall). Users (clients) connect to Web servers through the gateway. So, user requests arrive to the gateway-cache and they are either forwarded to the Internet, or served through the cache, if the data are already cache-resident.

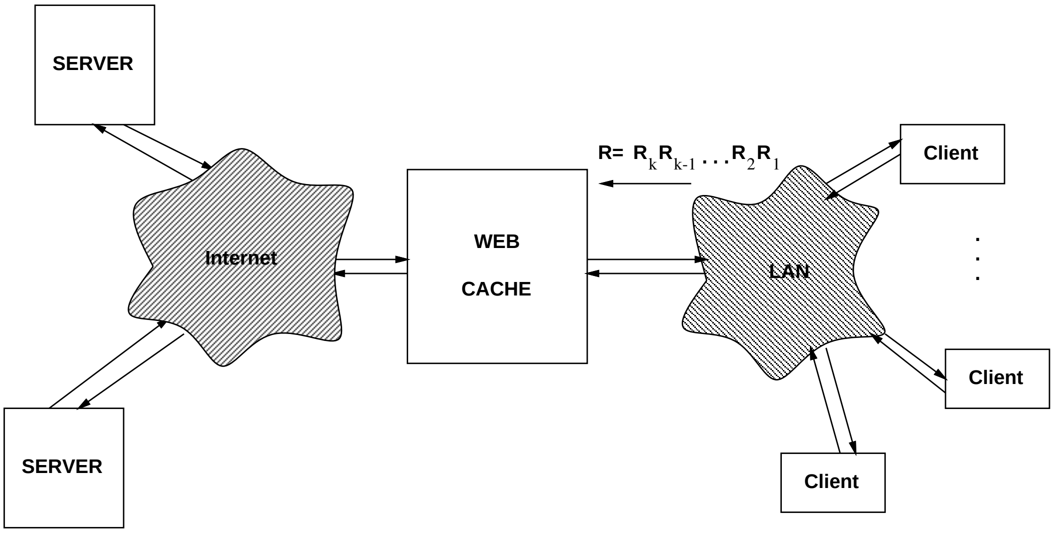


Figure 8.1: A simple caching environment

We assume that the set of all available objects, denoted $O = \{O_1, O_2, \ldots, O_N\}$, has size $|O| = N$. Also, we assume that client requests follow Zipf's distribution. Specifically, we assume that the stream of client requests, R, is a series of *independent* trials drawn from a Zipf distribution over the set of N possible objects (e.g., web pages or sites). This means that the next request in R will be for the i-th most popular of the N items with probability

$$P_N(i) = \frac{\Omega}{i} \tag{8.1}$$

where

$$\Omega = \frac{1}{H_N} \approx \frac{1}{\ln N}$$

H_N is the N-th harmonic number, which we approximate with $\ln N$. Zipf's function for $N = 10^i$, where $2 \leq i \leq 5$ is shown in Figure 8.2. Breslau et al. [18] studied the hypothesis of Zipf's law for web requests suggested by [20] and also [24] [4]. Their experimental results, based on actual Web traffic traces, indicated that web requests follow a Zipf-like distribution given by

$$P_N(i) = \frac{\Omega}{i^\alpha}$$

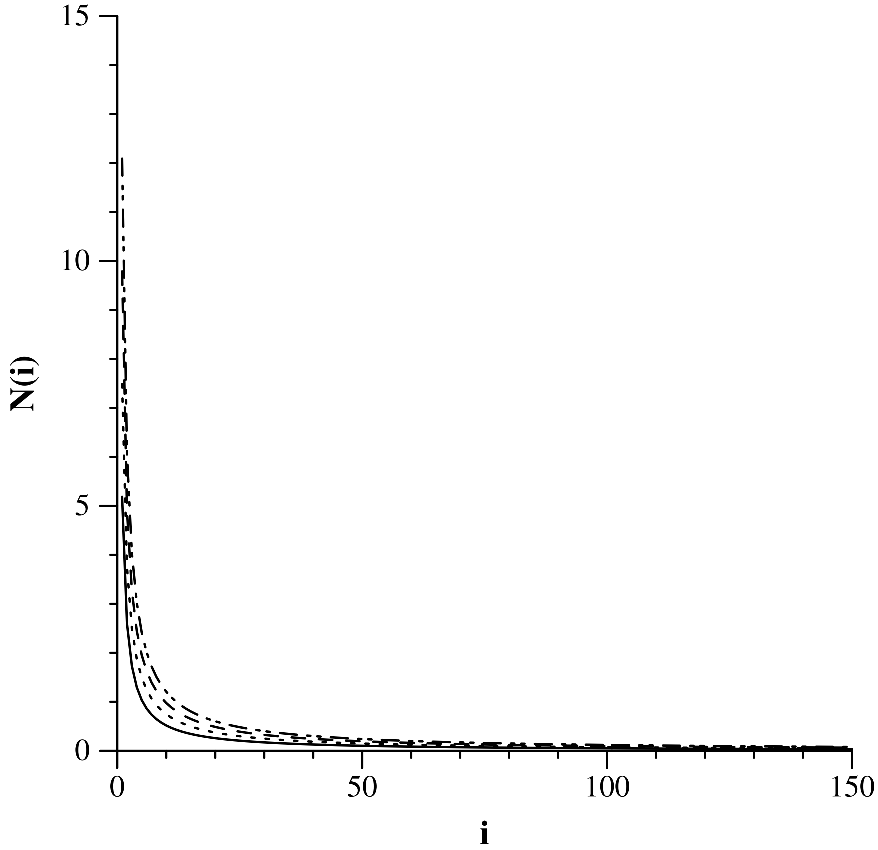


Figure 8.2: Zipf's function

where

$$\Omega = \left(\sum_{i=1}^{N} \frac{1}{i^{\alpha}} \right)^{-1} \approx \frac{1-\alpha}{N^{1-\alpha}}.$$

The parameter α depends on the trace source but always $0 < \alpha \leq 1$ (notice that for $\alpha = 1$ we get the true Zipf's distribution described above). In order to simplify the exposition and the calculations we will assume that $\alpha = 1$. The modifications for $\alpha \neq 1$ will be obvious. Furthermore, we assume that the system is *closed*, i.e., that N, the total number of objects, and their nature do not change (no objects "die" and no new ones are "born"). This assumption is realistic for time intervals of the order of weeks or months, when we observe no dramatic changes in the population of requested objects.

As in [42], one can calculate the number of accesses of the k most popular objects O_1, O_2, $\ldots$, O_k as follows. If a number of accesses N_A is directed to the set of the N objects and N_A is large enough, then, in general, object O_i, $1 \leq i \leq N$, will be accessed $P_i \times N_A$ times, based on Zipf's law. So, the total number of accesses to the k most popular objects is:

$$\sum_{i=1}^{k} N_A \times P_i = N_A \times \sum_{i=1}^{k} P_i = N_A \times \frac{H_k}{H_N} \tag{8.2}$$

This implies that, if we have a "hot" cache that serves the requests, which stores *only* the k most popular objects, then the cache hit rate is:

$$h = \frac{N_A \frac{H_k}{H_N}}{N_A} = \frac{H_k}{H_N} \tag{8.3}$$

Based on the above, we can calculate k, the number of objects in the cache, which can achieve a given hit-ratio h, from (8.3):

$$H_k = h \times H_N \Rightarrow \ln k = h \times H_N \Rightarrow k = e^{h \times H_N} \tag{8.4}$$

The calculations indicate that the given cache hit ratio h will be observed (measured) under the following two conditions:

1. Zipf's law holds for the set of accesses and objects measured;

2. the time interval during which measurements are made is large enough.

Note that, one can certainly develop scenarios where the accesses are in such an order that the cache hit rate becomes significantly lower than the expected h for short time intervals (e.g., when many consecutive accesses are targeted to the least popular objects). However, the cache hit ratio is the expected h for long enough intervals, i.e. long enough request streams. In what follows, we calculate an upper bound for the minimum length of the request stream, so that the measured object order according to their observed popularities is reliable (with high confidence) and the Zipf distribution estimated up at that point converges to the final one.

All the above calculations and the calculations in other works (e.g. [18]) assume that the request stream R is a stream of *independent* trials from Zipf's distribution. In this work we study this particular assumption of statistical independence in two directions: first we examine its impact on caching policy design and then we test its validity.

8.3 Consequences of the statistical independence assumption

8.3.1 Determining the Zipf distribution of the request stream

The statistical independence assumption we (among others) have made is very strong. In fact it is strong enough to allow us to give exact *theoretical* bounds for several parameters of the system. The question we answer in this section is the following: assuming that the request stream follows the *same* Zipf's distribution for a long enough time, how far in the past (in terms of past requests) do we need to look in order to determine this distribution with a very high probability (over the possible request streams)? It turns out that we need to look only on a *polynomial* (over the set of sites/pages) number of past requests. While this is a theoretical bound and it is still too large for all practical purposes, it points towards a new direction on implementing the LFU caching policy. We will study the LFU policy in detail in the next section.

Considering that the request stream R is a series of *independent* trials drawn from a Zipf distribution over the set S of N possible objects, at any point in R, the next request will be the i-th most popular of the N objects with probability $P(i) = \frac{a}{i}$, where $a = \frac{1}{H_N} \approx \frac{1}{\ln N}$. For the purposes of our analysis, we consider the environment *closed*, i.e. that the set S of the N objects does not change (none of the objects "dies" or changes, and no new objects are born).

In order to perform our analysis, we introduce the concept of a *past*, P, of a stream request R: P is a prefix of R. We define as $n_P(i)$ the number of appearances in P of the

i-th most popular object (in R). It follows that the expected value of $n_P(i)$ is:

$$E[n_P(i)] = \frac{|P|}{i \ln N} \tag{8.5}$$

where $|P|$ is the length of P. For simplicity, we denote this value as $E(i)$ in the following.

Given the concept of a *past* in a request stream, the problem we solve is the following: given a random R, how long should the past P be, so that the access frequencies in P render reliable measurements of object popularities that reflect exactly the distribution of the N objects for the *entire* R with very high probability?

The answer to this question provides information about the convergence of P to the real (final) Zipf distribution. We can provide theoretical upper bounds by taking advantage of the *knowledge* of the distribution in R and the assumption of *independence* between the requests in R.

In order to quantify the concept of confidence described above, we introduce the metric of *difference*, $D(i)$, for every object O_i:

$$D(i) = E(i) - E(i+1) \stackrel{(8.5)}{=} \frac{|P|}{i(i+1)} \tag{8.6}$$

Using $D(i)$, we characterize a past P as a *good past*, as follows.

Definition 9 (Good past): *A past P of a random stream of requests R is a* **good past** *of R if the following condition is met:*

> *$n_P(i)$, the number of appearances of O_i in P, is within distance $\frac{D(i)}{2}$ of its expected value, i.e. the following holds:*
>
> $$|n_P(i) - E(i)| \leq \frac{D(i)}{2}, \quad i = 1, 2, \ldots, N \tag{8.7}$$

If condition (8.7) holds for all objects, then the objects have exactly the same popularity ordering in P as they have in R. This can be easily deduced:

$$\left.\begin{array}{rcl} -\frac{D(i)}{2} & \leq & n_P(i) - E(i) \\ -\frac{D(i+1)}{2} & \leq & E(i+1) - n_P(i+1) \end{array}\right\} \stackrel{(+)}{\Rightarrow}$$

$$\Rightarrow n_P(i) - n_P(i+1) \geq D(i) - \frac{D(i)+D(i+1)}{2} \overset{(8.6)}{>} 0$$

for all i.

Effectively, the definitions of *difference* and *good past* allow us to specify a confidence radius around the expected value of each O_i in such a way so that, if the number of appearances falls into their confidence intervals, the objects retain their ordering in R (which is the same as the ordering of the $E(i)$'s), because the confidence intervals do *not* intersect. Based on the above, our problem becomes: how long should P be, so that it is a good past with very high probability?

Theorem 14: *For any $\epsilon > 0$, a past P of R of length $2N^2(N+1)^2 \ln^2 N \ln \frac{2N}{\epsilon}$ is a good past with probability at least $1-\epsilon$.*

Proof: In our analysis we use the following Chernoff bound [5]:

Lemma 19: *Let $X_1, X_2, \ldots, X_n$ be mutually independent random variables such that*

$$Pr[X_i = 1] = p$$

$$Pr[X_i = 0] = 1 - p$$

for some $p \in [0,1]$. Let $X = X_1 + X_2 + \ldots + X_n$ and $E[X] = pn$. Then

$$Pr[|X - pn| > \theta] \leq 2e^{-\frac{2\theta^2}{n}} \tag{8.8}$$

for any $\theta > 0$.

We define the following sequence of random variables for each O_i:

$$w_j(i) = \begin{cases} 1, & \text{if } j\text{-th request of } R \text{ is } O_i \\ 0, & \text{otherwise} \end{cases}, \; j = 1, \ldots, W$$

Then given that $w_j(i)$'s are mutually independent for all j, $n_W(i) = \sum_{j=1}^{W} w_j(i)$ and $Pr[w_j(i) = 1] \approx \frac{1}{i \ln N}$, due to Zipf's function. Thus, we can apply Lemma 8.8 with $p = \frac{1}{i \ln N}$

and $\theta = \frac{D(i)}{2}$, obtaining

$$\begin{aligned} Pr[|n_P(i) - E(i)| > \frac{D(i)}{2}] &< 2e^{-\frac{P}{2i^2(i+1)^2 \ln N}} \\ &\leq 2e^{-\frac{P}{2N^2(N+1)^2 \ln^2 N}} \end{aligned} \tag{8.9}$$

Note that

$$\begin{aligned} Pr[P \text{ is } \textit{not} \text{ good past}] = & \quad Pr[\text{ (8.7) not true for } O_1 \\ & \vee \text{ (8.7) not true for } O_2 \vee \ldots] \\ \leq & \sum_{i=1}^{N} Pr[\text{ (8.7) not true for } O_i] \\ \stackrel{(8.9)}{<} & \quad 2Ne^{-\frac{P}{2N^2(N+1)^2 \ln^2 N}} \end{aligned} \tag{8.10}$$

If our 'confidence' parameter is ϵ with $0 < \epsilon \leq 1$, then it must be true that

$$Pr[P \text{ is } \textit{not} \text{ a good past}] < \epsilon$$

But then from (8.10) we get that we must pick P so that

$$P \geq 2N^2(N+1)^2 \ln^2 N \ln \frac{2N}{\epsilon}$$

□

Notice that this bound is relatively large in terms of N, especially under the assumption that during this period the system is closed. The size of the bound is due to the very strong condition we want to satisfy (condition (8.7)) in order to obtain a good past.

The case of Zipf-like distributions: The above calculations can be easily modified for the case of a Zipf-like distribution with $\alpha < 1$. Then the upper bound of Theorem 14 becomes $O(N^{6-2\alpha} \ln \frac{2N}{\epsilon})$.

8.3.2 Practical LFU implementation for Web caching

The LFU (for Least Frequently Used) replacement policy for Web caches has been shown to perform optimally both theoretically [42] and experimentally [18] for large enough caches.

The LFU (or Perfect-LFU) policy replaces the least frequently used item in the cache with the new item requested by the user(s), when the new item is not already cached. The 'least frequently used' refers to *all* past requests, from the beginning of caching[1]. Obviously the Perfect-LFU policy is not practical. In order to implement it, one needs to keep statistics for *all* web sites accessed during the *whole* past and then use them to determine their popularity. Here we show that under the statistical independence assumption we can approximate the hit rate of Perfect-LFU by using much fewer resources. Instead of examining *all* past requests in order to determine the popularity of each object, we take into account only the latest few requests to determine the ordering of the objects according to their popularity. Specifically, we introduce the concept of a *time window*, W, a time interval of the recent past, and implement an LFU policy, called *Window-LFU*, which replaces objects based on access measurements only in the window W. We prove analytically that, under certain assumptions, in order to achieve the same cache hit rate as Perfect-LFU, we need to pick a window size that depends polynomially on the cache size but only sublinearly (or even logarithmically in the case of $\alpha = 1$) on the number of available objects on the Web. Considering that the number of objects on which we need to keep statistics cannot be larger than the window size, it becomes clear that a small window size leads to a small number of objects on which statistics are collected. In this fashion, we overcome the most significant obstacle to the implementation of LFU policies: the impractically large amount of resources necessary for the calculation and storage of statistics on accessed objects.

In any caching scheme, a cache stores the items that have been accessed in some recent past, which we refer to as *time window* W (or simply *window*). We denote as $|W|$ the length of the window, measured in number of requests. The window W always contains the last $|W|$ requests, which are denoted as $W_1, W_2, \ldots, W_{|W|}$; for example, in Figure 8.1, window W contains requests $(W_1, W_2, \ldots, W_{|W|}) = (R_k, \ldots, R_{k-|W|+1})$. The existent analytical

[1]The architecture community when referring to LFU usually refers to replacing the least frequently used item *in the future*, whereas in our case we are looking at the *past* requests.

results have been drawn for $|W| = |R|$, where R contains all requests received by the cache since the beginning of its operation [42].

Considering the definition of W, as the $|W|$ most recent requests in R, we define $n_W(i)$ as the number of appearances of the i-th most popular object, object O_i, in W; the definition of the i-th most popular object is based on the number of requests in R. As before, the expected value of $n_W(i)$ is easily calculated:

$$E[n_W(i)] = \frac{|W|}{i \ln N}$$

Again, we denote this value as $E(i)$.

The goal of our analysis is to estimate the length of W, so that, if the cache measures access frequencies using the information in the last $|W|$ accesses, then the achieved cache hit rate approximates the one achieved with Perfect-LFU. We formalize this, through the following definition:

Definition 10 (Good estimator): *Let C be the number of objects that are kept in the cache. Then the window W will be a* **good estimator** *of the C most popular objects in R, if two conditions are met:*

- *the number of appearances of the C most popular objects is greater than $E(C+1)$;*
- *the number of appearances of the remaining $N - C$ objects is smaller than $E(C+1)$, i.e. the remaining objects do not interfere with the ordering of the C most popular ones.*

The definition indicates that, while we ensure a separation between the C most popular objects and the $(N-C)$ less frequent, the conditions of the definition are too weak to ensure the correct ordering of the objects according to their access frequencies in the complete history[2]. However, the critical observation is that for the implementation of Perfect-LFU,

[2]This is a much weaker condition than condition (8.7), and this results to much better bounds.

it is sufficient to have the C most popular objects in the cache, without a need for knowledge of the specific order of their frequencies (popularities) in the window. The weakness of the conditions in the definition are the key of the improvements we achieve.

If both of the conditions are met, then we designate the window as *good*. In this case, our replacement algorithm, Window-LFU, will provide exactly the same performance (hit rate) as Perfect-LFU. So, the goal of our analysis is to choose W in such a way, so that it will ensure the "goodness" of the window with very high probability. Then our hit-rate will be very close to the one achieved with Perfect-LFU.

In the analysis, we use the following Chernoff bounds from [5]:

Lemma 20: *Let $X_1, X_2, \ldots, X_n$ be mutually independent random variables such that*

$$Pr[X_i = 1] = p$$

$$Pr[X_i = 0] = 1 - p$$

for some $p \in [0, 1]$.
Let $X = X_1 + X_2 + \ldots + X_n$ and $E[X] = pn$. Then

$$Pr[X > (1+\theta)pn] \leq e^{-\frac{\theta^2}{3}pn} \tag{8.11}$$

$$Pr[X - pn < -\alpha] < e^{-\alpha^2/2pn} \tag{8.12}$$

$$Pr[X - pn > \beta] < e^{-2pn/27} \tag{8.13}$$

$$Pr[X - pn > \gamma] < e^{\gamma - (\gamma+pn)\ln(1+\gamma/pn)} \tag{8.14}$$

where $0 < \theta \leq 1$, $\alpha > 0$, $\beta > 2pn/3$ and $\gamma > 0$.

We use these bounds, because they describe quantitatively the following simple fact: a series of independent trials is concentrated very heavily around its expected value. We use this fact to prove that, one does not need many trials, i.e. past requests, in order to get a very good estimate of the expected value, i.e. the frequency.

Theorem 15: *Let $\epsilon > 0$ be any parameter, and C the cache size. Under the assumptions of statistical independence and Zipf's distribution for the requests*

$$H_{W\text{-}LFU}(C) \geq (1-\epsilon) \cdot H_{P\text{-}LFU}(C)$$

where $H_{W\text{-}LFU}(C)$ is the hit rate of Window-LFU with window size

$$|W| = \max\{\Theta(C^3 \ln C \ln N \ln \frac{1}{\epsilon}), \Theta(C \ln^2 N \ln \frac{1}{\epsilon})\}$$

and $H_{P\text{-}LFU}(C)$ is the hit rate of Perfect-LFU.

Proof: Assume that the N objects are ordered according to their popularity in R (O_1 is the most popular, O_2 the second most popular, etc.). We define the following sequence of random variables for each O_i:

$$w_j(i) = \begin{cases} 1, & \text{if } W_j \text{ (the } j\text{-th request in } W\text{) is for } O_i \\ 0, & \text{otherwise} \end{cases}$$

for all $j = 1, 2, \ldots, |W|$. Then, by hypothesis, the $w_j(i)$'s are mutually independent, $n_W(i) = \sum_{j=1}^{|W|} w_j(i)$ and $Pr[w_j(i) = 1] \approx \frac{1}{i \ln N}$ from Zipf's distribution.

We distinguish the following cases:

Case 1: $1 \leq i \leq (C+1)$

From inequality (8.12) with $\alpha = E(i) - E(C+1)$ we obtain:

$$\begin{aligned} & Pr[n_w(i) < E(C+1)] = \\ & = Pr[n_w(i) - E(i) < -(E(i) - E(C+1))] \\ & < e^{-\frac{|W|(C+1-i)^2}{2i(C+1)^2 \ln N}} \\ & \leq e^{-\frac{|W|}{2C \ln N}} \end{aligned} \tag{8.15}$$

Case 2: $C+1 < i < 2(C+1)$

From inequality (8.11) with $\theta = \frac{i-C-1}{C+1}$ we obtain:

$$\begin{aligned} Pr[n_w(i) > E(C+1)] & < e^{-\frac{(i-C-1)^2}{3(C+1)^2} \frac{|W|}{i \ln N}} \\ & \leq e^{-\frac{|W|}{3C(C+1)^2 \ln N}} \end{aligned} \tag{8.16}$$

Case 3: $2(C+1) \leq i \leq 3(C+1)$

From inequality (8.13) with $\beta = E(C+1) - E(i)$ we obtain:

$$\begin{aligned} Pr[n_w(i) > E(C+1)] &< e^{-\frac{2|W|}{27i \ln N}} \\ &\leq e^{-\frac{|W|}{40.5(C+1) \ln N}} \end{aligned} \tag{8.17}$$

Case 4: $3(C+1) < i \leq N$

From inequality (8.14) with $\gamma = E(C+1) - E(i)$ we obtain:

$$\begin{aligned} Pr[n_w(i) > E(C+1)] &< e^{E(C+1) - E(i) - E(C+1) \ln \frac{i}{C+1}} \\ &\leq e^{-\frac{2|W|}{5(C+1) \ln N}} \end{aligned} \tag{8.18}$$

The probability that a window is not a good estimator is evaluated as:

$$\begin{aligned} & Pr[\text{the window is } \textit{not} \text{ a good estimator}] = \\ & = Pr[\text{Case 1 holds} \vee \ldots \vee \text{Case 4 holds, for some } i] \\ & \leq Ce^{-\frac{|W|}{2C \ln N}} + Ce^{-\frac{|W|}{3C(C+1)^2 \ln N}} + \\ & + (C+2)e^{-\frac{|W|}{40.5(C+1) \ln N}} + (N - 3(C+1))e^{-\frac{2|W|}{5(C+1) \ln N}} \end{aligned} \tag{8.19}$$

If we choose

$$|W| = \max\{\Theta(C^3 \ln C \ln N \ln \frac{1}{\epsilon}), \Theta(C \ln^2 N \ln \frac{1}{\epsilon})\}$$

then we can force the probability in Equation (8.19) to be smaller than any constant $\epsilon > 0$.

If we denote with $H_{\text{W-LFU}}(C, W)$ and $H_{\text{P-LFU}}(C)$ the cache hit rates for the Window-LFU and Perfect-LFU cases, respectively (with the window size $|W|$ as specified above), the following relations hold:

$$H_{\text{P-LFU}}(C) = \sum_{i=1}^{C} \frac{1}{i \ln N} \tag{8.20}$$

$$
\begin{aligned}
H_{\text{W-LFU}}(C,W) &= Pr[\text{next requested item } r \text{ is in cache}] \\
&\geq \sum_{i=1}^{C} Pr[r=i|W \text{ is good estimator}] \times Pr[W \text{ is good estimator}] \\
&= \sum_{i=1}^{C} \frac{1}{i \ln N} \times (1 - Pr[W \text{ is } \textit{not} \text{ a good estimator}]) \qquad (8.21) \\
&\overset{(8.19)}{\geq} (1-\epsilon) \sum_{i=1}^{C} \frac{1}{i \ln N} \\
&= (1-\epsilon) H_{\text{P-LFU}}(C)
\end{aligned}
$$

where $\epsilon > 0$ is the accuracy constant we have chosen. □

The case of Zipf-like distributions: The proof is essentially the same when the distribution is Zipf-like (for example the distributions described in [18]). In this case the bound of Theorem 15 is $|W| = \max\{\Theta(N^{1-\alpha} C^3 \ln C \ln \frac{1}{\epsilon}), \Theta(N^{2-2\alpha} C \ln \frac{1}{\epsilon})\}$. In practice α is a number between 0.6 and 0.9 (cf. [18]), so the window size depends *sublinearly* on N.

The importance of the theoretical results

From the analysis, it becomes clear that the required window size has a limited dependence on the total number of objects N that can be accessed. Thus, we succeed to reduce the effect of parameter N on our cache replacement policy, which is an advantage because N is not a parameter of the cache system itself, and we cannot control it. As the number of objects for which one keeps statistics cannot be larger than the window size, a smaller time window results in a smaller set of such objects. Unfortunately, Case 2 shows a dependency on the cache size C, which is impractical for big enough caches. However, the result is very strong (approximation of Perfect-LFU performance within any constant factor), which means that, in practice, smaller window sizes should perform quite well, e.g., $|W| = O(C \ln N)$ or $O(C^2 \ln N)$. This is supported by the results of simulations with traces of real traffic, as described below.

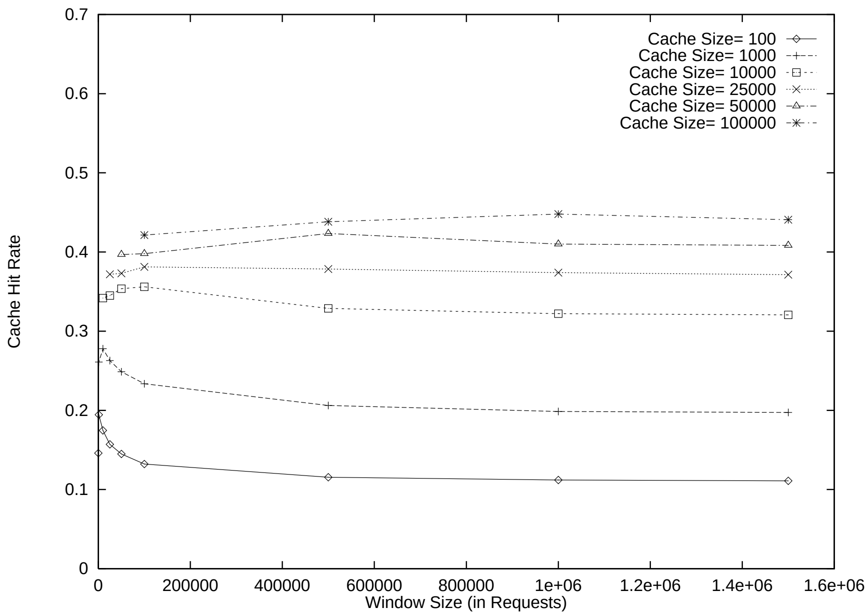


Figure 8.3: Cache hit rate for variable window sizes (long trace)

Simulation Results

We have performed several simulations of a cache employing the Window-LFU policy using traces from actual traffic patterns. Specifically, we have used four traces from NLANR [35]. The first three traces are short (they include the object requests of one day each) while the last one is longer, including the requests of a week. The traces are continuously updated; the latest of those we used are from the first week of June 2000.

Our simulator simulates a cache that uses Window-LFU replacement policy for variable window sizes W. When a replacement of an object is due, the object with the smallest frequency is replaced. If more than one objects in the cache have the same (smallest) frequency, then we replace the one which was used *least recently*; i.e., we use an LRU (Least Recently Used) rule in order to "break ties" among the least popular window objects.

Figure 8.4 shows the results of the simulations on short (daily) traces. The first was recorded the last week of January 2000 and contains $600,000$ requests for a total of $375,000$

different pages (objects). The other two traces were recorded during the first week of June 2000, with $800,000$ requests for $424,000$ objects and $462,000$ requests for $260,000$ objects respectively. Figure 8.3 shows the results of the simulations, using the long trace with a total of $751,000$ objects and 1.5 million requests. As the results show, the behavior of the cache is similar to the shorter traces.

The plots show the cache hit rate of a cache with Window-LFU as a function of the size of window W, and for various cache sizes C (measured in objects). As the results indicate, for all cache sizes, the effect of the window size is insignificant after a "threshold" value. This verifies our first result that, a small window size is sufficient to achieve the highest possible cache hit rate (per cache size). Interestingly, with small cache sizes and small window lengths, it appears that the cache hit rate improves. This seemingly surprising result can be explained easily: the source is the dependency among successive requests. In our analyses, we have assumed that the object requests R_1, R_2, $\ldots$, are independent. However, in real traces there is a dependency among them, which actually leads to higher locality, and thus improves the hit rate. As the length of the window increases, the cache hit rate, which is $h = \frac{Number\ of\ cache\ hits}{Number\ of\ Requests}$, decreases, because the "longer" history (due to the longer window size) tends to influence the replacement decision using popularities from a distant past, which do not apply to the recent past (due to the dependency of requests). In simpler terms, this means that, if an object was accessed heavily in the distant past, but is not accessed any more, Perfect-LFU will not replace this object from the cache, unless a new object is accessed at least as many times as the previous one; in the (possibly very long) meantime, the object will reside in the cache, although it is not accessed at all. So, with the longer window size, it takes longer for the cache to store the more recently accessed objects, which are more likely to be accessed in the near future due to the aforementioned locality. On the other hand, a small window will not allow accesses made in the distant past to be counted against the calculation of object frequencies. Thus, considering locality, the cache will store objects more likely to be accessed in the future.

To summarize, the simulations indicate that Window-LFU performs better than expected from the analytical results. This phenomenon is due to that, the assumption of request independence made for the analysis does not hold; there are dependencies in a real trace of requests, which actually render the Window-LFU more effective than the analysis indicates.

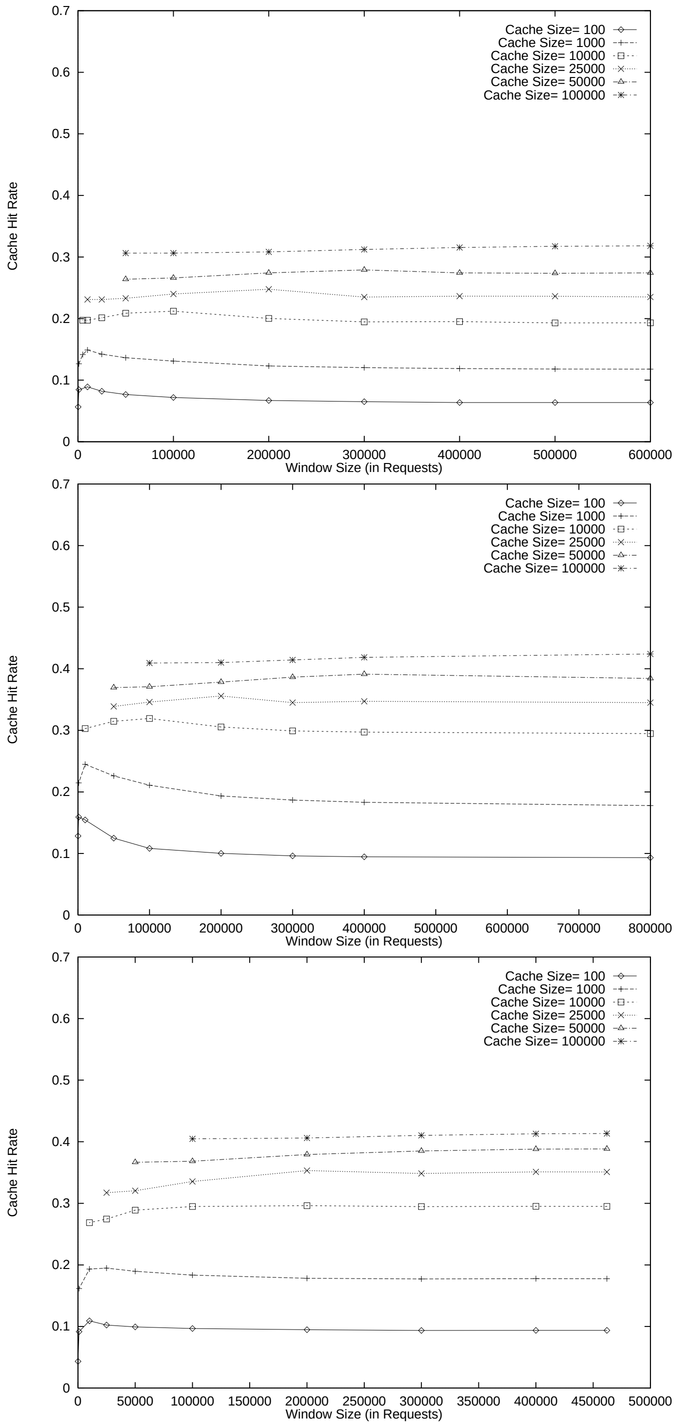


Figure 8.4: Cache hit rate for variable window sizes (short traces)

Bibliography

[1] F. Afrati, S. Cosmadakis, C. Papadimitriou, G. Papageorgiou, and N. Papakostantinou. The complexity of the traveling repairman problem. *Informatique Theorique et Applications*, **20**(1):79–87, 1986.

[2] A. Agrawal, P. Klein and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, **24**, pp 440–456, 1995.

[3] Akamai Technologies, Inc. http://www.akamai.com.

[4] V. Almeida, A. Bestavros, M. Crovella and A. de Oliveira. Characterizing reference locality in the WWW. *IEEE International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996

[5] N. Alon, J.H. Spencer, and P. Erdos. *The Probabilistic Method.* John Wiley and Sons, 1992.

[6] I. Althofer, G. Das, D. Dobkin, D. Joseph, L. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, **9**:1, 1993.

[7] S. Arora. Polynomial-time approximation schemes for Euclidean TSP and other geometric problems. *Journal of the ACM* **45(5)** pp 1-30, Sep. 1998. Preliminary version in *Proceedings of 37th IEEE Symp. on Foundations of Computer Science(FOCS)*, pp 2-12, 1996.

[8] S. Arora, M. Grigni, D. Karger, P. Klein, A. Woloszyn. A Polynomial-Time Approximation Scheme for Weighted Planar Graph TSP. *Proceedings of the* 9^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp 33–41, 1998.

[9] S. Arora and G. Karakostas. Approximation Schemes for Minimum Latency Problems. *Proceedings of 31st Annual ACM Symposium on Theory of Computing (STOC)*, pp 688–693, 1999.

[10] S. Arora and G. Karakostas. A $2+\epsilon$ approximation algorithm for the k-MST problem. *Proceedings of 11th Annual ACM-SIAM Symposium on Discrete Algorithms(SODA)*, pp 754–759, 2000.

[11] S. Arora, C. Lund, R. Motwani, M. Sudan, M. Szegedy. Proof Verification and the Hardness of Approximation for Problems. *Journal of the ACM* **45(3)** pp 501-555, May 1998.

[12] S. Arya, H. Ramesh. 2.5-factor approximation algorithm for the k-MST problem. In *Information Processing Letters*, 65(3), pp. 117–118, 1998.

[13] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. Improved approximation guarantees for minimum weight k-trees and prize-collecting salesmen. In *Proceedings,27th ACM Symp. on Theory of Computing*, pp 277–283, 1995.

[14] M. Baentsch, L. Baum, G. Molter, S. Rothkugel and P. Sturm. Enhancing the Web's infrastructure: From Caching to Replication. *IEEE Internet Computing*, pp. 18–27, March/April 1997.

[15] L. Bianco, A. Mingossi and S. Ricciardelli. The traveling salesman problem with comulative costs. *Networks*, vol. 23, no. 2, pp 81-91, Mar. 1993.

[16] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan and M. Sudan. The minimum latency problem. *Proceedings of 26th ACM Symp. on Theory Of Computing(STOC)*, pp 163–171, 1994.

[17] A. Blum, R. Ravi, and S. Vempala. A constant factor approximation for the k-MST problem. In *Proceedings,28th ACM Symp. on Theory of Computing*, pp 442–448, 1996.

[18] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99*, 1999.

[19] CacheFlow, Inc. http://www.cacheflow.com.

[20] C.R. Cunha, A. Bestavros, and M.E. Crovella. Characteristics of WWW Client-based Traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, July 1995.

[21] X. Deng and C. Papadimitriou. Exploring an unknown graph. *Proc. 31st IEEE Symp. on Foundations of Computer Science(FOCS)*, pp. 355-361, 1990.

[22] M. Fischetti, H.W. Hamacher, K. Jörnsten, F. Maffioli Weighted k-cardinality trees: complexity and polyhedral structure. In *Networks,* 24, pp. 11-21, 1994.

[23] N. Garg. A 3-approximation for the minimum tree spanning k vertices. *Proc. 37th IEEE Symp. on Foundations of Computer Science(FOCS)*, pp.302–309, 1996.

[24] S. Glassman. A caching relay for the World Wide Web. *First International Conference on the World Wide Web*, CERN, Geneva, Switzerland, May 1994.

[25] M. Goemans and J. Kleinberg. An improved approximation ratio for the minimum latency problem. *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms(SODA)*, pp 152-158, 1996.

[26] M. Goemans and D. Williamson. A general approximation technique for constrained forest problems. In *SIAM J. Comput.*, 24:296–317, 1995.

[27] M. Grötschel, L. Lovász, and A. Schrijver. Geometric Algorithms and Combinatorial Optimization. Springer Verlag, Berlin 1988.

[28] D. Hockbaum. *Approximation Algorithms for $\mathcal{NP}$-hard Problems.* PWS Publishing Company, 1997.

[29] G. Karakostas and D. Serpanos Practical LFU implementation for Web Caching. Princeton University Technical Report TR-622-00, 2000.

[30] E. Koutsoupias, C. Papadimitriou and M. Yannakakis. Searching a fixed graph. *Lecture Notes in Computer Science(LNCS)* 1099, pp.280–289, Springer Verlag, 1996.

[31] A. Lucena. Time-dependent traveling salesman problem - the deliveryman case. *Networks*, vol. 20, no. 6, pp 753-763, Oct. 1990.

[32] E. Mayer, H. Prömel and A. Steger (Eds.). *Lectures on Proof Verification and Approximation Algorithms.* Lecture Notes in Computer Science 1367, Springer, 1998.

[33] E. Minieka. The delivery man problem on a tree network. *Annals of Operations Research*, Vol. 18, no. 1-4, pp 261-266, Feb. 1989.

[34] J.S.B. Mitchell. Guillotine Subdivisions Approximate Polygonal Subdivisions: A Simple Polynomial-Time Approximation Scheme for Geometric TSP, k-MST, and Related Problems. SIAM Journal on Computing:**28**, 1999.

[35] National Laboratory for Applied Network Research. http://www.nlanr.net (traces at: ftp://ircache.nlanr.net/traces/.

[36] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[37] C. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and Systems Sciences*, 43:425–440, 1991.

[38] C. Papadimitriou and M. Yannakakis. The traveling salesman problem with distances one and two. *Mathematics of Operations Research*, **18**(1):1–11, Feb. 1993.

[39] S. Rajagopalan and V.V. Vazirani. Logarithmic approximation of minimum weight k-trees. Unpublished manuscript, 1995.

[40] R. Ravi, R. Sundaram, M. Marathe, D. Rosenkrantz, and S. Ravi. Spanning trees short and small. In *Proceedings, 5th ACM-SIAM Symp. on Discrete Algorithms*, pp 546–555, 1994.

[41] D.N. Serpanos, G. Karakostas, and W.H. Wolf. Effective Caching of Web Objects Using Zipf's Law. In *Proceedings of IEEE International Conference on Multimedia and Expo (ICME 2000)*, page (To appear), July, 30 - August, 2 2000.

[42] D.N. Serpanos and W.H. Wolf. Caching Web Objects Using Zipf's Law. In *Proceedings of SPIE, Vol. 3527, Photonics East, Technical Conference 3527: Multimedia Storage and Archiving Systems III, Boston, MA, USA, November 2-4, 1998, pp. (not available yet). Available at http://www.spie.org/web/meetings/programs/ pe98/confs/3527.html*, 1998.

[43] V.V. Vazirani *Approximation algorithms*, manuscript

[44] A. Zelikovsky and D. Lozevanu. Minimal and bounded trees. In *Tezele Cong. XVIII Acad. Romano-Americane, Kishinev,* pp. 25-26, 1993.