OPTIMIZATION PROBLEMS IN TELECOMMUNICATIONS AND THE INTERNET

By

CARLOS A.S. OLIVEIRA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2004

To my wife Janaina.

ACKNOWLEDGMENTS

The following people deserve my sincere acknowledgments:

- My advisor, Dr. Panos Pardalos;
- Dr. Mauricio Resende, from AT&T Research Labs, who was responsible for introducing me to this University;
- My colleages in the graduate school of the Industrial and Systems Engineering Department;
- My family, and especially my parents;
- My wife.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

OPTIMIZATION PROBLEMS IN TELECOMMUNICATIONS AND THE INTERNET

By

Carlos A.S. Oliveira

August 2004

Chair: Panos M. Pardalos
Major Department: Industrial and Systems Engineering

Optimization problems occur in diverse areas of telecommunications. Some problems have become classical examples of application for techniques in operations research, such as the theory of network flows. Other opportunities for applications in telecommunications arise frequently, given the dynamic nature of the field. Every new technique presents different challenges that can be answered using appropriate optimization techniques.

In this dissertation, problems occurring in telecommunications are discussed, with emphasis for applications in the Internet. First, a study of problems occurring in multicast routing is presented. Here, the objective is to allow the deployment of multicast services with minimum cost. A description of the problem is provided, and variations that occur frequently in some of these applications are discussed.

Complexity results are presented for multicast problems, showing that it is NP-hard to approximate these problems effectively. Despite this, we also describe algorithms that give some guarantee of approximation.

A second problem in multicast networks studyed in this dissertation is the multicast routing problem. Its objective is to find a minimum cost route linking source to destinations, with additional quality of service constraints. A heuristic based on a Steiner tree algorithm is proposed, and used to construct solutions for the routing problem. This construction heuristic is also used as the basis to develop a restarting method, based on the greedy randomized adaptive search procedure (GRASP).

The last part of the dissertation is concerned with problems in wireless networks. Such networks have numerous applications due to its highly dynamic nature. Algorithms to compute near optimal solutions for the minimum backbone problem are proposed, which perform in practice much better than other methods. A distributed version of the algorithm is also provided.

CHAPTER 1
INTRODUCTION

Computer networks are a relatively new communication medium that has quickly become essential for most organizations. In this dissertation, we present some optimization problems occurring in computer and telecommunications networks. Performing optimization on such networks is important for several reasons, including cost and speed of communication. We concentrate on two types of networks that have recently received much attention. The first type is *multicast systems*, which are used to reliably share information with a (possibly large) group of clients. The second type of networks considered in this dissertation is *wireless ad hoc systems*, an important type of networks with several applications.

We are mostly concerned about computational issues arising in the optimization of problems occurring on telecommunications networks. Thus, although we present mathematical programming aspects for each of these problems, the main objective will be to derive efficient algorithms, with or without guarantee of approximation.

The topics discussed in the dissertation are divided as follows. In Chapter 2, a survey of research on the area of multicast systems is presented. The review is used as a starting point for the topics that will be discussed later in the dissertation related to multicast networks.

Chapter 3 introduces the problem that will be studied in the next chapters, the streaming cache placement problem (SCPP). Variants of this basic problem are introduced, and all variants are proved to be $\mathcal{NP}$-hard.

Chapter 4 is dedicated to the study of approximability properties of the different versions of the SCPP. It is shown that in general the SCPP cannot have a polynomial time approximation scheme (PTAS). This demonstrates that the SCPP is a very hard problem not only to solve exactly, but also to approximate. We also show that for the directed flow version it is not possible to approximate the problem by less than $\log\log|D|$, where D is the set of destinations.

In Chapter 5, algorithms for different versions of the SCPP are proposed. Both approximation algorithms, as well as heuristics are discussed. Initially, some algorithms with performance guarantee are proposed. However, due to complexity results, these algorithms in general do not give good results for problems found in practice. Heuristic algorithms are then studied, and two main strategies for construction heuristics are discussed. Results of computational results with these methods are presented and compared.

Another problem in multicast networks is discussed in Chapter 6. The routing problem in multicast networks asks for an optimal route, i.e., a minimum cost tree connecting the source node to destinations. The routing problem for multicast networks is known to be NP-hard. We propose new heuristics, and use these heuristics to implement a greedy adaptive search procedure (GRASP).

In the last part of the dissertation, wireless network systems are discussed. In particular, ad hoc systems (also known as MANETs) are studied. Chapter 7 is dedicated to the problem of determining a minimum backbone for such ad hoc networks. A new algorithm for this problem is given, and the advantages of this algorithm are addressed. A distributed version of the algorithm is also proposed.

Finally, in Chapter 8 general conclusions are given about the work presented in the dissertation. Future work in the area is presented, and some concluding remarks about this area of research are given.

CHAPTER 2
A SURVEY OF COMBINATORIAL OPTIMIZATION PROBLEMS IN MULTICAST ROUTING

In multicasting routing, the main objective is to send data from one or more source to multiple destinations, while at the same time minimizing the usage of resources. Examples of resources which can be minimized include bandwidth, time and connection costs. In this chapter we survey applications of combinatorial optimization to multicast routing. We discuss the most important problems considered in this area, as well as their models. Algorithms for each of the main problems are also presented.

2.1 Introduction

A basic application of computer networks consists of sending information to a selective, usually large, number of clients of some specific data. Common examples of such applications are multimedia distribution systems (Pasquale et al., 1998), video-conferencing (Eriksson, 1994), software delivery (Han and Shahmehri, 2000), group-ware (Chockler et al., 1996), and game communities (Park and Park, 1997). Multicast is a technique used to facilitate this type of information exchange, by routing data from one or more sources to a potentially large number of destinations (Deering and Cheriton, 1990). This is done in such a way that overall utilization of resources in the underlying network is minimized in some sense.

To handle multicast routing, many proposals of multicast technologies have been done in the last decade. Examples are the MBONE (Eriksson, 1994), MOSPF (Moy, 1994a), PIM (Deering et al., 1996), core-based trees (Ballardie

et al., 1993) and shared tree technologies (Chiang et al., 1998; Wei and Estrin, 1994). Each proposed technology requires the solution of (usually hard) combinatorial problems. With the proliferation of services that require multicast delivery, the associated routing methods became an important source of problems for the combinatorial optimization community. Many objectives can be devised when designing protocols, routing strategies, and overall networks that can be optimized using techniques from combinatorial optimization.

In this chapter we discuss some of the combinatorial optimization problems arising in the area of multicast routing. These are very interesting in their own, but sometimes are closely related to other well known problems. Thus, the cross-fertilization of ideas from combinatorial optimization and multicast networks can be beneficial to the development of improved algorithms and general techniques. Our objective is to review some of the more interesting problems and give examples and references of the existing algorithms. We also discuss some problems recently appearing in the area of multicast networks and how they are modeled and solved in the literature.

2.1.1 Multicast Routing

The idea of sending information for a large number of users is common in systems that employ broadcasting. Radio and TV are two standard examples of broadcasting systems which are widely used. On the other hand, networks were initially designed to be used as a communication means among a relatively small number of participants.

The TCP/IP protocol stack, which is the main technology underlying the Internet, uses routing protocols for delivery of packets for single destinations. Most of these protocols are based on the calculation of shortest paths. A good example of a widely used routing protocol is the OSPF (Moy, 1994b;

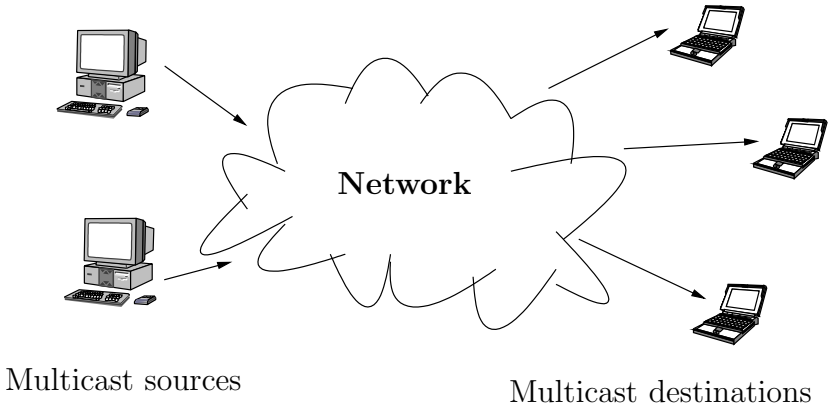

Figure 2–1: Conceptual organization of a multicast group.

Thomas II, 1998) (Open Shortest Path First), which is used to compute routing tables for routers inside a subnetwork. In OSPF, each router in the network is responsible for maintaining a table of paths for reachable destinations. This table can be created using the Dijkstra's algorithm (Dijkstra, 1959) to calculate shortest paths from the current node to all other destinations in the current sub-network. This process can be done deterministically in polynomial time, using at most $\mathcal{O}(n^3)$ iterations, where n is the number of nodes involved.

However, with the Internet and the increased use of large networks, the necessity appeared for services targeting larger audiences. This phenomenon became more important due to the development of new technologies such as virtual conference (Sabri and Prasada, 1985), video on demand, group-ware (Ellis et al., 1991), etc. This series of developments gave momentum for the creation of *multicast routing protocols*. In multicast routing, data can be sent from one or more source nodes to a set of destination nodes (see Figure 2–1). It is required that all destinations be satisfied by a stream of data.

Dalal and Metcalfe (1978) were the first to give non-trivial algorithms for routing of packets in a multicast network. From then on, many proposals have been made to create technology supporting multicast routing, such as

by Deering (1988), Eriksson (1994), and Wall (1980). Some examples of multicast protocols are PIM – Protocol Independent Multicast (Deering et al., 1996), DVMRP – Distance-Vector Multicast Routing Protocol (Deering and Cheriton, 1990; Waitzman et al., 1988), MOSPF – Multicast OSPF (Moy, 1994a), and CBT – Core Based Trees (Ballardie et al., 1993). See Levine and Garcia-Luna-Aceves (1998) for a detailed comparison of diverse technologies.

2.1.2 Basic Definitions

A *multicast group* is a set of nodes in a network that need to share the same piece of information. A multicast group can have one or more source nodes, and more than one destination. Note that even when there is more than one source, the same information is shared among all nodes in the group.

A multicast group can be static or dynamic. *Static groups* cannot be changed after its creation. Starting with Wall (1980), the problem of routing information in static groups is frequently modeled as a type of Steiner tree problem. On the other hand, *dynamic groups* can have members added or removed at any time (Waxman, 1988). Clearly the task of maintaining routes for dynamic groups is complicated by the fact that it is not known in advance which nodes can be added or removed.

Multicast groups can be also classified according to the relative number of users, as described by Deering and Cheriton (1990). In *sparse groups*, the number of participants is small compared to the number of nodes in the network. In the other situation, in which most of the nodes in the network are engaged in multicast communication, the groups involved are called *pervasive groups* (Waitzman et al., 1988).

For more information about multicast networks in general, one can consult the surveys by A.J. Frank (1985), and Paul and Raghavan (2002). A good

introduction to multicasting in IP networks is given in the Internet Draft by Semeria and Maufer (1996) (available online). Other interesting related literature include Du and Pardalos (1993a); Pardalos and Du (1998); Wan et al. (1998); Pardalos et al. (2000, 1993); Pardalos and Khoury (1996, 1995).

2.1.3 Applications of Multicast Routing

Applications of multicast routing have a wide spectrum, from business to government and entertainment. One of the first applications of multicast routing was in audio broadcasting. In fact, the first real use of the Internet MBONE (Multimedia Backbone, created in 1992) was to broadcast audio from IETF (Internet Engineering Task Force) meetings over the Internet (Eriksson, 1994).

Another important application of multicast routing is video conference (Yum et al., 1995), since this is a resource-intensive kind of application, where a group of users is targeted. It has requirements, such as real-time image exchanging and allowing interaction between geographically separated users, also found in other types of multimedia applications. Being closely related to the area of remote collaboration, video conferencing has received great attention during the last decade. Among others, Pasquale et al. (1998) give a detailed discussion about utilization of multicast routing to deliver multimedia content over large networks, such as the Internet. Jia et al. (1997) and Kompella et al. (1996) also proposed algorithms for multicast routing applied to real-time video distribution and video-conferencing problems.

Many other interesting uses of multicast routing have been done during the last decade, with examples such as video on demand, software distribution, Internet radio and TV stations, etc.

2.1.4 Chapter Organization

The remainder of this chapter is organized as follows. In Section 2.2 we give a common ground for the description of optimization problems in multicast routing. We start by giving the terminology used throughout the chapter, mainly from graph theory. Then, we discuss some of the common problems appearing in this area. In Section 2.3 we discuss delay constrained Steiner tree problems. These are the most studied problems in multicast routing, from the optimization point of view, being used in diverse algorithms. Thus, we discuss many of the versions of this problem considered in the literature. In Section 2.4 we review some other optimization problems related to multicast routing. They are the multicast packing problem, the multicast network dimensioning problem, and the point-to-point connection problem. Finally, in Section 2.5 we give some concluding remarks about the subject.

2.2 Basic Problems in Multicast Routing

In this section we discuss the basic problems occurring in multicast networks. We start by an introduction to terminology used. In the sequence we discuss some basic problems which are addressed in the multicast routing literature.

2.2.1 Graph Theory Terminology

Graphs in this chapter are considered to be undirected and without loops. In our applications, the nodes in a graph represent hosts, and edges represent network links. We use $N(v)$ to denote the set of neighbors of a node $v \in V$. Also, we denote by $\delta(V)$ the number of such neighbors.

With each edge $(i, j) \in E$ we can associate functions representing characteristics of the network links. The most widely used functions are capacity $c(i, j)$, cost $w(i, j)$ and delay $d(i, j)$, for $i, j \in V$. For each edge $(i, j) \in E$, the

associated capacity $c(i, j)$ represents the maximum amount of data that can be sent between nodes i and j. In multicasting applications this is generally given by an integer multiple of some unity of transmission capacity, so we can say that $c(i, j) \in Z_+$, for all $(i, j) \in E$.

The function $w(i, j)$ is used to model any costs incurred by the use of the network link between nodes i and j. This include leasing costs, maintenance costs, etc.

Some applications, such as multimedia delivery, are sensitive to transmission delays and require that the total time between delivery and arrival of a data package be restricted to some particular maximum value (Ferrari and Verma, 1990). The delay function $d(i, j)$ is used to model this kind of constraint. The delay $d(i, j)$ represents the time needed to transmit information between nodes i and j. As a typical example, video-on-demand applications may have specific requirements concerning the transmission time. Each packet i can be marked with the maximum delay d_i that can be tolerated for its transmission. In this case the routers must consider only paths where the total delay is at the most d_i.

A path in a graph G is a sequence of nodes $v_{i_1}, \ldots, v_{i_j}$, where $(v_{i_k}, v_{i_{k+1}})$ is in E, for all $k \in \{1, \ldots, j-1\}$. In a routing problem we want to find paths from a source s to a set D of destinations, satisfying some requirements. The cost $w(P)$ of a path P is defined as the sum of the costs of all edges $(v_{i_k}, v_{i_{k+1}})$ in P. A path P between nodes u and v is called a *minimum path* if there is no path P' in G such that $w(P') < w(P)$. The *path delay* $d(P)$ is defined as the delay incurred when routing data between nodes v_1 and v_k through path $P = (v_1, \ldots, v_k)$. In other words, $d(P) = \sum_{i=1}^{k-1} d(v_i, v_{i+1})$.

In this chapter, we use interchangeably the words edge and link to relate to the same object. The word link is used when it is more appropriate in

the application context. For more information of graph theoretical aspects of multicast networks, see Berry (1990).

2.2.2 Optimization Goals

Different objectives can be considered when optimizing a multicast routing problem, such as, for example, path delay, total cost of the tree, and maximum congestion. We discuss some of these objectives.

Quality of service is an important consideration with network service, and it is mostly related to the time needed for data delivery. Depending on the quality of service requirements of an application, one of the possible goals is to minimize path delay. The best example of application that needs this quality of service is video-conference. The path delay is an additive delay function, corresponding to the sum of delays incurred from source to destination, for all destinations. It is interesting to note that this problem is solvable in polynomial time, since the paths from source to destination are considered separately. Shortest path algorithms such as, for example, the Dijkstra's algorithm (Dijkstra, 1959), can be used to achieve this objective.

A second objective is to minimize the total cost of the routing tree. This is again an additive metric, where we look for the minimum sum of costs for edges used in the routing tree. In this case, however, the optimization objective is considerably harder to achieve, since it can be shown to be equivalent to the minimum Steiner tree, a classical $\mathcal{NP}$-hard problem (Garey and Johnson, 1979).

Another example of optimization goal is to minimize the maximum network congestion. The *congestion* on a link is defined as the difference between capacity and usage. The higher the congestion, the more difficult it is to handle failures in some other links of the network. Also, higher congestion makes

it harder to include new elements in an existing multicast group, and therefore is an undesirable situation in dynamic multicast. Thus, in a well designed network it is interesting to keep congestion at a minimum.

2.2.3 Basic Multicast Routing Algorithms

The most basic way of sending information to a multicast group is using *flooding*. With this technique, a node sends packets through all its adjacent links. If a node v receives a packet p from node u for which it is not the destination, then v first checks if p was received before. If this is true, the packet does not need to be sent again. Otherwise, the v just re-sends the packet to all other adjacent nodes (excluding u). The formal statement of this strategy is shown in Algorithm 1. It is clear that after at most n such steps (where n is the number of nodes in the network), the package must have reached all nodes, including the destinations. Thus, the algorithm is correct. The number of messages sent by each node is at most n. The number of messages received by v is at most $n\delta_G(v)$.

```
Receive packet p from node u
if destination(p) = v then
    Packet-Received
else
    if packet was not previously processed then
        Sent packet p to all nodes in N(v) \ {u}
    end
end
```

Algorithm 1: Flooding algorithm for node v in a multicast network

This method of packet routing is simple, but very inefficient. The first reason is that it uses more bandwidth than required, since many nodes which are not in the path to the destination will end up by receiving the packet. Second, each node in the network must keep a list of all packets which it sent,

in order to avoid loops. This makes the use of flooding prohibitive for all but very small networks. Another problem, which is more difficult to solve, is how to guarantee that a packet will be delivered, since the network can be disconnected due to some link failure, for example.

The *reverse path-forwarding algorithm* is a method, proposed by Dalal and Metcalfe (1978), used to reduce the network usage associated with the flooding technique. The idea is that, for each node v and source node s in the network, v will determine in a distributed way what is the edge $e = (u, v)$, for some $u \in V$, which is in the shortest path from s to v. This edge is called the *parent link*. The parent link can be determined in different ways, and a very simple method is: select $e = (u, v)$ to be the parent link for source s if this was the first edge from which a packet from s was received. With this information, a node can selectively drop incoming packets, based on its source. If a packet p is received from a link which is not considered to be in the shortest path between the source node and the current node, then p is discarded. Otherwise, the node broadcasts p to all other adjacent links, just as in the flooding algorithm. The parent link can also be updated depending on the information received from other nodes. Other algorithms can be used to enhance this basic scheme as discussed, e.g., by Semeria and Maufer (1996).

2.2.4 General Techniques for Creation of Multicast Routes

During the last decades a number of basic techniques were proposed for the construction of multicast routes. Diot et al. (1997) identified some of the main techniques used in the literature. They describe these techniques as being divided into source based routing, center based tree algorithms, and Steiner tree based algorithms.

In *source based routing*, a routing tree rooted at the source node is created for each multicast group. This technique is used, for example, in the DVMRP and PIM protocols. Some implementations of source based routing make use of the reverse path-forwarding algorithm, discussed in the previous sub-session (Dalal and Metcalfe, 1978). Sriram et al. (1998) observed that this technique does a poor job in routing small multicast groups, since it tries to optimize the routing tree without considering other potential users not in the current group.

Among the source based routing algorithms, the *Steiner tree based methods* focus on minimization of tree cost. This is probably the most used approach, since it can leverage the large number of existing algorithms for the Steiner tree problem. There are many examples of this technique (such as in Bharath-Kumar and Jaffe (1983); Wall (1982); Waxman (1988); Wi and Choi (1995)), which will be discussed on Section 2.3.

In contrast to source based routing, *center based tree algorithms* create routing trees with a specified *root node*. This root node is computed to have some special properties, such as, for example, being closest to all other nodes. This method is well suited to the construction of shared trees, since the root node can have properties interesting to all multicast groups. For example, if the root node is the topological center of a set of nodes, then this is the node which is closest to all members of the involved multicast groups. In the case of the topological center, the problem of finding the root node becomes $\mathcal{NP}$-hard, but there are other versions of the problem which are easier to solve. An important example of use of this idea occurs in the CBT (core-based tree) algorithm (Ballardie et al., 1993).

A recent method proposed for distributing data in multicast groups is called *ring based routing* (Baldi et al., 1997; Ofek and Yener, 1997). The idea

is to have a ring linking nodes in a group, to minimize costs and improve reliability. Note for example that trees can be broken by just one link failure; on the other hand, rings are 2-connected structures, which offer a more reliable interconnection.

2.2.5 Shortest Path Problems with Delay Constraints

Given a graph $G(V, E)$, a source node s and a destination node t, with $s, t \in V$, the shortest path problem consists of finding a path from s to t with minimum cost. The solution of shortest path problems is required in most implementations of routing algorithms. This problem can be solved in polynomial time using standard algorithms (Dijkstra, 1959; Bellman, 1958; Ford, 1956).

However, other versions of the shortest problem are harder, and cannot be solved exactly in polynomial time. An example of this occurs when we add delay constraints to the basic problem. The delay constraints require that the sum of the delays from source to each destination be less than some threshold. In this case, the shortest path problem becomes $\mathcal{NP}$-hard (Garey and Johnson, 1979) and therefore, some heuristic algorithms must be used in order to find efficient implementations (e.g. Salama et al. (1997b)). For example, Sun and Langendoerfer (1995) and Deering and Cheriton (1990) have proposed good heuristics for this problem.

Some algorithms for shortest path construction are less useful than others, due to properties of their distributed implementations. According to Cheng et al. (1989), a disadvantage of the distributed Bellman-Ford algorithm for shortest path computation is that is difficult to recover from link failures, from the bouncing effect (Sloman and Andriopoulos, 1985) caused by loops, and from termination problems caused by disconnected segments. Thus, a

chief requirement for shortest path algorithms used in multicast routing is to have a scalable distributed implementation. The problems associated with distributed requirements for shortest path algorithms are discussed by Cheng et al. (1989), who proposed a distributed algorithm to overcome such limitations.

2.2.6 Delay Constrained Minimum Spanning Tree Problem

In the *minimum spanning tree* (MST) problem, given a graph $G(V, E)$, we need to find a minimum cost tree connecting all nodes in V. This problem can be solved in polynomial time by Kruskal's algorithm (Kruskal, 1956) or Prim's algorithm (Prim, 1957). However, similarly to the shortest path problem, the MST problem becomes $\mathcal{NP}$-hard when delay constraints are applied to the resulting paths in the routing tree. This fact can be easily shown, since the minimum spanning tree problem is a generalization of the minimum cost path problem.

Salama et al. (1997a) discuss the *delay constrained minimum spanning tree problem*. They propose a simple heuristic, which resembles Prim's algorithm, to give an approximate solution to the problem. The proposed method can be described as follows. In its first phase, the algorithm tries to incorporate links, ordered according to increasing cost, but without creating cycles. At each step, the algorithm must also insure that the current (partial) solution satisfy the delay constraints. If this is not true, then a relaxation step is carried on, which consists of the following procedure. If a node can be linked by an alternative path, while reducing the delay, then the new path is selected. If, after this relaxation step, there is still no path with a suitable delay for some node, then the algorithm fails and returns just a partial answer.

Other examples of algorithms for computing delay constrained spanning trees include the work of Chow (1991). In his paper, an algorithm for the problem of combining different routes into one single routing tree is proposed. For more information about delay constrained routing, see Salama et al. (1997c), where a comparison of diverse algorithms for this problem is performed.

2.2.7 Center-Based Trees and the Topological Center Problem

In the context of generation of multicast routing trees, some routing technologies, such as PIM and CBT, use the technique known as *center-based trees* (Salama et al., 1996), which was initially developed in Wall (1982). This method can be classified as a center-based routing technique, as described in Section 2.2.4. In this approach the first step is to find the node v which is the *topological center* of the set of senders and receivers. The topological center of a graph $G(V, E)$ is defined as the node $v \in V$ which is closest to any other node in the network, i.e., the node v which minimizes $\max_{u \in V} d(v, u)$. Then, a routing tree rooted at v is constructed and used throughout the multicast session.

The basic reasoning behind the algorithm is that the topological center is a better starting point for the routing tree, since it is expected to change less than other parts of the tree. This scheme departs from the idea of rooting the tree at the sender, and therefore can be extended to be used by more than one multicast group at the same time.

The topological center is, however, a $\mathcal{NP}$-hard problem (Ballardie et al., 1993). Thus, other related approaches try to find root nodes that are not exactly the topological center, but which can be thought of as a good approximation. Along these lines we have algorithms using *core points* (Ballardie et al., 1993) and also *rendez-vous points* (Deering et al., 1994).

It is interesting to note that, for simplicity, most of the papers which try to create routing trees using center-based techniques simply disregard the $\mathcal{NP}$-complete problem and try to find other approximations. It is not completely understood how good these approximations can be for practical instances. However, Calvert et al. (1995) gave an informative comparison of the different methods of choosing the center for a routing tree, based on several experiments.

2.3 Steiner Tree Problems and Multicast Routing

In this section we discuss different versions of the Steiner tree problem, and how they can be useful to solve problems arising in multicast routing. Some of the algorithm for the Steiner tree are also presented.

2.3.1 The Steiner Tree Problem on Graphs

Steiner tree problems are very useful in representing solutions to multicast routing problems. They are employed mostly when there is just one active multicast group and the minimum cost tree is wanted. In the Steiner tree problem, given a graph $G(V, E)$, and a set $R \subseteq V$ of required nodes, we want to find a minimum cost tree connecting all nodes in R. The nodes in $V \setminus R$ can be used if needed, and are called "Steiner" points. This is a classical $\mathcal{NP}$-hard problem (Garey and Johnson, 1979), and has a vast literature on its own (Bauer and Varma, 1997; Du et al., 2001; Du and Pardalos, 1993b; Hwang and Richards, 1992; Hwang et al., 1992; Kou et al., 1981; Takahashi and Matsuyama, 1980; Winter, 1987; Winter and Smith, 1992). Thus, in this subsection we give only some of the most used results. For additional information about the Steiner problem, one can consult the surveys Winter (1987); Hwang and Richards (1992); Hwang et al. (1992).

One of the most well known heuristics for the Steiner tree problem was proposed by Kou et al. (1981), and frequently refereed to as the KMB heuristic.

There is practical interest in this heuristic, since it has a performance guarantee of at most twice the size of the optimum Steiner tree. The steps of the KMB heuristic are shown in Algorithm 2.

Construct a complete graph $K(R, E)$ where the set of nodes is R.
Let the distance $d(i, j)$, $i, j \in R$ be the shortest path from i to j in G.
Find a minimum spanning tree T of K.
Replace each edge (i, j) in T by the complete path from i to j in G.
Let the resulting graph be T'
Compute a minimum spanning tree $\hat{T}$ of T'.
repeat
  $r \leftarrow false$
  if *there is a leaf* $w \in \hat{T}$ *which is not in* R **then**
    Remove w from $\hat{T}$
    $r \leftarrow true$
  end
until not r

Algorithm 2: Minimum spanning tree heuristic for Steiner tree.

Theorem 1 (Kou *et al.* (Kou et al., 1981)) *Algorithm 2 has a performance guarantee of* $2 - 2/p$, *where* $p = |R|$.

Wall (1980) made a comprehensive study of how the KMB heuristic performs in problems occurring in real networks. For example, Doar and Leslie (1993) report that this heuristic can give much better results than the claimed guarantee, usually achieving 5% of the optimal for a large number of realistic instances.

Another basic heuristic for Steiner tree was proposed by Takahashi and Matsuyama (1980). This heuristic works in a way similar to the Dijkstra's and Prim's algorithms. The operation of the heuristic consists of increasing the initial solution tree using shortest paths. Thus, it is classified as part of the broad class of *path-distance heuristics*. Initially, the tree is composed of the source node only. Then, at each step, the heuristic searches for a still

unconnected destination d that is closest to the current tree T, and adds to T the shortest path leading to d. The algorithm stops when all required nodes have been added to the solution tree.

The Steiner tree technique for multicast routing consists of using the Steiner problem as a model for the construction of a multicast tree. In general, it is considered that there is just one source node for the multicast group. The set of required nodes is defined as the union of source and destinations. This technique is one of the most studied for multicast tree construction, with many algorithms available (Bauer and Varma, 1995; Chow, 1991; Chen et al., 1993; Kompella et al., 1992, 1993b,a; Hong et al., 1998; Kompella et al., 1996; Ramanathan, 1996). In the remaining of this and the next sections we discuss the versions of this problem which are most useful, as well as algorithms proposed for them.

In one of the first uses of the Steiner tree problem for creating multicast trees, Bharath-Kumar and Jaffe (1983) studied algorithms to optimize the cost and delay of a routing tree at the same time. Also, Waxman (1988) discusses heuristics for cost minimization using Steiner tree, taking in consideration the dynamics of inclusion and exclusion of members in a multicast group.

It is also important to note some of the limitations of the Steiner problem as a model for multicast routing. It has been pointed out by Sriram et al. (1998) that Steiner tree techniques work best in situations where a virtual connection must be established. However, in the most general case of packet networks, like the Internet, it does not make much sense to minimize the cost of a routing tree, since each packet can take a very different route. In this case, it is more important to have distributed algorithms with low overhead. Despite this, Steiner trees are still useful as a starting point for more sophisticated algorithms.

2.3.2 Steiner Tree Problems with Delay Constraints

The simplest way of applying the Steiner tree problem in multicast networks requires that the costs of edges in the tree represent the communication costs incurred by the resulting multicast routes. In this case we can just apply a number of existing algorithms, such as the ones discussed in the previous section, for the Steiner tree problem. However, most applications have additional requirements in terms of the maximum delay for delivering of the information.

That is the reason why the most well studied version of the Steiner tree problem applied to multicast routing is the delay constrained version (Im et al., 1997; Kompella et al., 1992, 1993b,a; Jia, 1998; Sriram et al., 1998). We give in this section some examples of methods used to give approximate solutions to this problem.

One of the strategies used to solve the delay constrained Steiner tree problem is to adapt existing heuristics, by adding delay constraints. The heuristic proposed by Kompella et al. (1993b), for example, uses methods that are similar to the KMB algorithm (Kou et al., 1981). The resulting heuristic is composed of three stages. The first stage consists of finding a closure graph of constrained shortest paths between all members of a multicast group. The *closure graph* of G is a complete graph which has the set of nodes $V(G)$ and, for each pair of nodes $u, v \in V$, an edge representing the cost of the shortest path between u and v.

In the second stage, Kompella's algorithm finds a constrained spanning tree of the closure graph. To do this, the heuristic uses a greedy algorithm based on edge costs, to find a spanning tree with low cost. In the last stage, edges of the spanning tree found in the previous step are mapped back to the original paths in the graph. At the same time, loops are removed using

the shortest path algorithm on the expanded constrained spanning tree. The time complexity of the whole procedure is $\mathcal{O}(\Delta n^3)$, where Δ is the maximum delay allowed by the application. It should be noted, however, that even being very similar to the KMB heuristic, this algorithm does not have any proved approximation guarantee. This happens because the delay constraints make the problem much harder to approximate.

Sriram et al. (1998) proposed an algorithm for constructing delay-constrained multicast trees which is optimized for sparse, static groups. Their algorithm is divided into two phases. The first phase is distributed, and works by creating delay constrained paths from source to each destination. The paths are created using a unicast routing algorithm, so it can use information already available on the network. The second phase uses the computed paths to define a routing tree. Each path is added sequentially and cycles are removed as they appear. Basically, on iteration i, when a new path P_i is added to an existing tree T_{i-1}, each intersection $P_i \cap T_{i-1}$ of the path with the old tree is tested. This is necessary to determine if just the part of P_i which does not intersect can be used, while maintaining the same delay constraint. If this is possible, then the tree becomes T_i, after adding the non-intersecting part of the path. Otherwise, the algorithm must remove some parts of the old tree in order to avoid a cycle.

Another heuristic for the delay constrained Steiner tree problem is presented by Feng and Yum (1999). This heuristic uses the idea of constructing a minimum cost tree, as well as a minimum delay tree, and then combining the resulting solutions. Recall that a *shortest delay tree* can be computed using some algorithm for shortest paths, in polynomial time, with the delay being used as the cost function. Thus, the hard part of the algorithm consists of finding the minimum cost tree and then decide how to combine it with the

minimum delay tree. The algorithm used to compute the minimum cost, delay constrained tree is a modification of the Dijkstra's algorithm, which maintains each path within a specified delay constraint. To combine different trees, the algorithm employs a loop removal subroutine, which verifies if the resulting paths still satisfies the delay constraints. The resulting complexity of this algorithm is similar to the complexity of the Dijkstra's algorithm, and therefore is an improvement in terms of computation time.

Another possible method for designing good multicast routing trees is to start from algorithms for computing constrained minimum paths. This was the technique chosen by Kumar et al. (1999), who proposed two heuristics for the constrained minimum cost routing problem. In the first heuristic, which is called "dynamic center based heuristic", the idea is to find a center node to which all destinations will be linked, using constrained minimum paths. The center node c is calculated initially by finding the pair of nodes with highest minimum delay path, and taking c as the node in the middle of this path. Other destinations are linked using minimum delay paths with low cost. The second heuristic, called "best effort residual delay heuristic", follows a similar idea, but this time each node added to the current routing tree T has a residual delay bound. New destinations are then linked to the tree through paths which have low cost and delay smaller than the residual delay of the connecting node $v \in T$.

Not only delay constraints have being used with the multicast routing problem. Jiang (1992) discusses another version of the multicast Steiner tree problem, this time with link capacity constraints. His work is related to video-conferencing, where many users need to be source nodes during the establishment of the conference. One of the ideas used is that, as each user can become a source, then a distinct multicast tree must be created for each user. He

proposes some heuristics to solve this problem, with computational results for the heuristics.

As a last example, Zhu et al. (1995) proposed a heuristic for routing with delay constraints with complexity $\mathcal{O}(k|V|^3 \log V)$. The algorithm has two phases. In the first phase, a set of delay-bounded paths is constructed from source to each destination, to form a delay-bounded tree. Then, in the second phase the algorithm tries to optimize this tree, by reducing the total cost at each iteration. The algorithm is also shown useful to optimize other objective functions than total cost. For example, it can be used to minimize the maximum congestion in the network, after changes in the second phase to account for the new objective function. In the paper there are comparisons between the proposed heuristic and the heuristic for Steiner tree problem proposed by Kou et al. (1981). The results show that the heuristic achieves solutions very close to that given by the algorithm for Steiner tree.

Sparsity and Delay Minimization

Chung et al. (1997) proposed heuristics to the delay constrained minimum multicast routing, when considering the structure of sparse problems. The heuristic depends on the use of other algorithms to find approximate solutions to Steiner problem. The Steiner tree heuristic is used to return two solutions: in the second run, the cost function c is replaced by the delay function d. Thus, there are two solution which optimize different objective functions. The main idea of the proposed algorithm is trying to optimize the cost of the routing tree, as well as the maximum delay, at the same time. To do this, the algorithm uses a method proposed by Blokh and Gutin (1996), which is based on Lagrangian relaxation. A critique that can be done to the work of Chung et al. (1997) is that the goal of optimizing the Steiner tree with delay cost is not what is

required in most applications. For example, a solution can be optimal for this goal, however some path from s to a destination d can still have delay greater than a constant Δ. This happens because the global optimum does not implies that each source-destination path is restricted to the maximum delay.

2.3.3 The On-line Version of Multicast Routing

The multicast routing problem can be generalized in the following way. Suppose that a multicast group can be increased or reduced by means of on-line requests posted by nodes of the network. This is a harder problem, since optimal solutions, when considering just a fixed group, can quickly become inaccurate, and even very far from the optimum, after a number of additions and removals.

Researchers in the area of multicasting routing have devised some ways to deal with the problem of reconfiguring a multicast tree when inclusions and departures of members of a group occur (Aguilar et al., 1986; Waxman, 1988). A common approach consists of modifying the simple existing algorithms in order to avoid the re-computation of the entire tree for each change. However, as noted in Pasquale et al. (1998), a problem with such methods is that the global optimality of the resulting trees is lost at each change, and a very bad solution can emerge after many such local modifications.

One of the difficulties of the source tree based techniques in this respect is that, for each change in the multicast group, a new tree must be computed to restore service at the required level. The algorithms necessary to create this tree are, however, expensive, and this makes the technique not suitable for dynamic groups. Kheong et al. (2001) proposed an algorithm to speed up the creation of multicast routing trees in the case of dynamic changes. The idea is to maintain caches of pre-computed multicast trees from previous groups.

The cache can be used to quickly find new paths, connecting some of the members of the group. An algorithm for retrieving data from the path cache was proposed, which finds similarities between the previous and the current multicast groups. Then the algorithm constructs a connecting path using parts of the paths store in the cache.

The difficulty of adapting source based techniques to the dynamic case has motivated the appearance of specialized algorithms for the on-line version of the problem. For example, Waxman (1988) defines two types of on-line multicast heuristics. The first type allows a rearrangement of the routing tree after some number of changes, while the second type does not allow such reconfigurations. The theoretical model for this problem is given by the so-called *on-line Steiner problem*. In this version of the problem, one needs to construct a solution to a Steiner problem subject to the addition and deletion of nodes (Imase and Waxman, 1991; Westbrook and Yan, 1993; Sriram et al., 1999). This is clearly a $\mathcal{NP}$-hard problem, since it is a generalization of the Steiner problem.

Waxman (1988) studied how a routing tree must be changed when new nodes are added or removed. To better describe this situation, he proposed a random graph model, where the probability of existing an edge between two nodes depends on the Euclidean distance between then. This probability decreases exponentially with the increase of distance between nodes. The random inclusion of links can be used to represent the random addition of new users to a multicast group. Waxman also described a greedy heuristic to approximately solve instances generated according to this model.

Hong et al. (1998) proposed a dynamic algorithm which is capable of handling additions and removals of elements to an existing multicast group. The algorithm is again based on the Steiner tree problem, with added delay

constraints. However, to decrease the computational complexity of the problem, the authors employed the Lagrangian relaxation technique. According to their results, the algorithm finds solutions very close to the optimum when the network is sparse.

Feng and Yum (1999) devised a heuristic algorithm with the main goal of allowing easy insertion of new nodes in a multicast group. The algorithm is similar to Prim's algorithm for spanning trees in which it, at each step, takes a non-connected destination with minimum cost and tries to add this destination to the current solution. The algorithm also uses a priority queue Q where the already connected elements are stored. The key in this priority queue is the total delay between the elements and the source node. The algorithm uses a parameter k to determine how to compute the path from a destination to the current tree. Given a value of k, the algorithm computes k minimum delay paths, from the current destination d to each of the smallest k elements in the priority queue. Then, the best of the paths is chosen to be part of the routing tree. An interesting feature of the resulting algorithm is that, changing the value of the parameter k will change the amount of effort needed to connect destination. Clearly, when increasing the value of k, better results will be obtained. This algorithm facilitates the inclusion of new elements, because the same procedure can be used to grow the existing tree, in order to accommodate a new node.

Sriram et al. (1999) proposed new algorithms for the on-line, delay constrained minimum cost multicast routing that try to maintain a fixed quality of service by specifying minimum delays. The algorithm is able to adapt the routing tree to changes in membership due to inclusions and exclusions of users. One of the problems they try to solve is how to determine the moment in which the tree must be recomputed, and for how long should the algorithm

just do modifications to the original tree. To answer this question, the authors introduced the concept of *quality factor*, which measures the usefulness of part of the routing tree to the rest of the users. When the quality factor of part of a tree decreases to a specific threshold, that part of the tree must be modified. The authors discuss a technique to rearrange the tree such that the minimum delays continue to be respected.

The first algorithm proposed by Sriram et al. (1999) starts by creating a set of delay constrained minimum cost paths. For each destination, a path is created with bandwidth greater than the required bandwidth B, and with delay less than the maximum delay Δ. The next phase uses the resulting paths to create a complete routing tree. The algorithm adds sequentially the edges in each path, and at each step removes the loops created by the addition of the path. Loops are removed in a way such that the delay constraints are not violated.

The second algorithm proposed in Sriram et al. (1999) is a distributed protocol, where initially each destination receives a message in order to add new paths to the source tree. The nodes are kept in a priority list, ordered by increasing delay requirements. According to the order in the list, the destinations receive messages, which ask them to compute parameters over the available paths, and then construct the new paths that will form the final routing tree.

A technique that has been employed by some researchers consists of using information available from unicast protocols to simplify the creation of multicast routes. For example, Baoxian et al. (2000) proposed a heuristic for routing with delay constraints which is based on the information given by

OSPF. Reusing this information, the resulting algorithm can run with improved performance, in this case with complexity $\mathcal{O}(|D||V|)$, where D is the set of destinations.

The resulting algorithm has two steps. In the first step, it checks, for each destination d_i, if there is some path from the source s to destination d_i satisfying the delay. In the second step, the algorithm uses another heuristic to construct a unicast path from s to d_i. This heuristic basically construct a path using information about predecessor nodes from the unicast protocol as well as the delay information.

2.3.4 Distributed Algorithms

The multicast routing problem is in fact a distributed problem, since each node involved has some available processing power. Thus, it is natural to look for distributed algorithms which can use this computational power in order to reduce their time complexity. A number of papers have focused on distributed strategies for delay constrained minimum spanning tree (Jia, 1998; Chen et al., 1993).

A good example is the algorithm presented in Chen et al. (1993). The authors propose a heuristic that is similar to the general technique used in the KMB heuristic for Steiner tree, and the algorithm in Kompella et al. (1993b), for example. However, the main difference is that a distributed algorithm is used to compute the minimum spanning tree, which must be computed twice during the execution of the heuristic. The method used to find the MST is based on the distributed algorithm proposed by Gallager et al. (1983).

Kompella et al. (1993a) proposed some distributed algorithms targeting applications of audio and video delivering over a network, where the restriction on maximum delay plays an important role. The authors try to improve over

previous algorithms by using a distributed method. The main objective of the distributed procedure is to reduce the overall computational complexity. It must be noted, however, that, using decentralized algorithms, some of the global information about the network becomes harder to find (for example, global connectivity). Thus, a simplified version of the algorithm must be proposed which does not use global information. Nonetheless, according to the authors, the resulting algorithms stay within 15%-30% of the optimal solution, for most test instances.

The first algorithm in Kompella et al. (1993a) is just a version of Bellman-Ford algorithm (Bellman, 1957) which finds a minimum delay tree from the source to each destination. During the construction of each path, the algorithm verifies the cost of the available edges, and choose the one with lowest cost which satisfies the delay constraints. The algorithm has the objective of achieving feasibility, and therefore the results are not necessarily locally optimal. This can be achieved, however, using another optimization phase, such as a local search algorithm.

In the second algorithm, the strategy employed is similar to the Prim's algorithm for minimum spanning tree construction. Its consists of growing a routing tree, starting from the source node, until all destinations are reached. The resulting algorithm is specialized according to different techniques for selecting the next edge to be added to the tree. In the first edge selection strategy proposed, edges with smallest cost are selected, such that the delay restrictions are satisfied. The second edge selection rule tries to balance the cost of the edge with the delay imposed by its use. This is done by a "bias" factor, which gives higher priority to edges with smaller delay, among edges

with the same cost. The factor used for edge (i, j) is

$$b(i,j) = \frac{w(i,j)}{\Delta - (D(s,i) + d(i,j))},$$

where $D(s, i)$ is the minimum total delay between the source s and node i, Δ is the maximum allowed delay, and, as usual, $w(i, j)$ and $d(i, j)$ are the cost and delay between nodes i and j.

The authors also discuss the problem of termination, which is an important question for distributed algorithms. In this case, the problem exists because some configurations can report an infeasible problem, while feasibility can be restored by making some changes to the current solution.

Shaikh and Shin (1997) presented a distributed algorithm where the focus is to reduce the complexity of distributed versions of heuristics for the the delay constrained Steiner problem. In their paper, the authors try to adapt the model of Prim's and Dijkstra's algorithms to the harder task of creating a multicast routing tree. In this way, they aim to reduce the complexity associated with the heuristics for Steiner tree, while producing good solutions for the problem. The methods employed by Dijkstra's shortest path and Prim's minimum spanning tree algorithm are interesting because they require only local information about the network, and therefore they are known to perform well in distributed environments. The operation of these algorithms consists of adding at each step a new edge to the existing tree, until some termination condition is satisfied.

In the algorithm proposed by Shaikh and Shin, the main addition done to the structure of Dijkstra's algorithm is a method for distinguishing between destinations and non-destination nodes. This is done by the use of an indicator function I_D which returns 1 if and only if the argument is not a destination node. The general strategy is presented in Algorithm 3. Note

that in this algorithm the accumulated cost of a path is set to zero every time a destination node is reached. This guides the algorithm to find paths that pass through destination nodes with higher probability. This strategy is called *destination-driven multicast*. The resulting algorithm is simple to implement and, according to the authors, perform well in practice.

```
input: G(V, E), s
for v ∈ V do d[v] ← ∞
d[s] ← 0
S ← 0
Q ← V /* Q is a queue */
while Q ≠ ∅ do
    v ← get_min(Q)
    S ← S ∪ {v}
    for u ∈ N(v) do
        if u ∉ S and d[u] > d[v]I_D[v] + w(u, v) then
            d[u] ← d[v]I_D[v] + w(u, v)
        end
    end
end
```

Algorithm 3: Modification of Dijkstra's algorithm for multicast routing, proposed by Shaikh and Shin (1997).

Mokbel et al. (1999) discuss a distributed heuristic algorithm for delay-constrained multicasting routing, which is divided in a number of phases. The initial phase of the algorithm consists of discovering information about nodes in the network, particularly about delays incurred by packages. In this phase, a packet is sent from a source to all other nodes in the neighborhood. The packet is duplicated at each node, using the flooding technique. At each node visited, information about the total delay and cost experienced by the packet is collected, added to the packet, and retransmitted. Each destination will receive packets with information about the path traversed during the delay time previously defined. After receiving the packets, each destination can

select the resulting path with lowest cost to be the chosen path. As the last step of the initial phase, all destinations send this information to the source node.

In the second phase, the source node will receive the selected paths for each destination and construct a routing tree based on this information. This is a centralized phase, where existing heuristics can be applied to the construction of the tree. To improve the performance of the algorithm, and to avoid an overload of packets in the network during the flooding phase, each node is required to maintain at most K packets at any time. Using this parameter, the time complexity of the whole algorithm is $\mathcal{O}(K^2|V|^2)$.

Sparse Groups

An important case of multicast routing occurs when the number of sources and destinations is small compared to the whole network. This is the typical case for big instances, where just a few nodes will participate in a group, at each moment. For this case, Sriram et al. (1998) proposed a distributed algorithm which tries to explore the sparsity of the problem. The algorithm initially uses information available through a unicast routing protocol to find pre-computed paths in the current network. However, problems can appear when these paths induce loops in the corresponding graph. In the algorithm, such intersections are treated and removed dynamically. The algorithm starts by creating a list of destinations, ordered according to the their delay constraints. Nodes with more strict delay constraints have the opportunity of searching first for paths. Each destination d_i will independently try to find a path from d_i to the source s. If during this process a node v previously added to the routing tree is found, then the process stops and a new phase starts. In this new phase, paths are generated from v to the destination i. The destination i chooses one

of the paths according to a selection function SF (similar to the function used by Kompella et al. (1993a)), which is defined for each path P and given by

$$SF(P) = \frac{C(P)}{\Delta - D_{T_{i-1}}(s, v) - D(P)},$$

where $C(P)$ and $D(P)$ are the cost and delay of path P, Δ is the maximum delay in this group, and $D_{T_{i-1}}(s, v)$ is the current delay between the source s and node v.

A problem that exists when the multicast group is allowed to have dynamic membership, is that a considerable amount of time is spent in the process of connection configuration. Jia (1998) proposes a distributed algorithm which addresses this question. A new distributed algorithm is employed, which integrates the routing calculation with the connection configuration phase. Using this strategy, the number of messages necessary to set up the whole multicast group is decreased.

2.3.5 Integer Programming Formulation

Integer programming has been very useful in solving combinatorial optimization problems, via the use of relaxation and implicit enumeration methods. An example of this approach to multicast routing is given by Noronha and Tobagi (1994), who studied the routing problem using an integer programming formulation. They discuss a general version of the problem in which there are costs and delays for each link, and a set $\{1, .., T\}$ of multicast groups, where each group i has its own source s_i, a set of n_i destinations $d_{i1} \ldots, d_{in_i}$, a maximum delay Δ_i, and a bandwidth request r_i. There is also a matrix $B^i \in R^{n \times n_i}$, for each group $i \in \{1, \ldots, T\}$, of source-destination requirements. The value of B^i_{jk} is 1 if $j = s$, -1 if $j = d_{jk}$, and 0 otherwise. The node-edge incidence matrix is represented by $A \in Z^{n \times m}$.

The network considered has n nodes and m edges. The vectors $W \in R^m$, $D \in R^m$ and $C \in R^m$ give respectively the costs, delays and capacities for each link in the network. The variables in the formulation are $X^1, \ldots, X^T$ (where each X^i is a matrix $m \times n_i$), $Y^1, \ldots, Y^T$ (where each Y^i is a vector m elements), and $M \in R^T$. The variable $X^i_{jk} = 1$ if and only if link j is used by group i to reach destination d_{ik}. Similarly, variable $Y^i_j = 1$ if and only if link j is used by multicast group i. Also, variable M_i represents the delay incurred by multicast group i in the current solution.

In the following formulation, the objectives of minimizing total cost and maximum delay are considered. However, the constant values β_c and β_d represent the relative weight given to the minimization of the cost and to the minimization of the delays, respectively. Using the variables shown, the integer programming formulation is given by:

$$\min \quad \sum_{i=1}^{T} r_i \beta_c C Y^i + \beta_d M_i \tag{2.1}$$

subject to

$$AX^i = B^i \qquad \text{for } i = 1, \ldots, T \tag{2.2}$$

$$X^i_{jk} \leq Y^i_j \leq 1 \qquad \text{for } i = 1, \ldots, T,\ j = 1, \ldots, n,\ k = 1, \ldots, n_i \tag{2.3}$$

$$M_i \geq \sum_{j=1}^{k} D_j X^i_{jk} \qquad \text{for } i = 1, \ldots, T,\ k = 1, \ldots, n_i \tag{2.4}$$

$$M_i \leq L_i \qquad \text{for } i = 1, \ldots, T \tag{2.5}$$

$$\sum_{i=1}^{T} r_i Y^i \leq C \tag{2.6}$$

$$X^i_{jk}, Y^i_j \in \{0, 1\}, \quad \text{for } 1 \leq i \leq T,\ 1 \leq j \leq K,\ 1 \leq k \leq n_i. \tag{2.7}$$

The constraints in the above integer program have the following meaning. Constraint (2.2) is the flow conservation constraint for each of the multicast groups. Constraint (2.3) determine that an edge must be selected when it is

used by any multicast tree. Constraints (2.4) and (2.5) determine the value of the delay, since it must be greater than the sum of all delays in the current multicast group and less than the maximum acceptable delay L_i. Finally, constraint (2.6) says that each edge i can carry a value which is at most the capacity C_i. This is a very general formulation, and clearly cannot be solved exactly in polynomial time because of the integrality constraints (2.7).

This formulation is used in Noronha and Tobagi (1994) to derive an exact algorithm for the general problem. Initially, the decomposition technique was used to decompose the constraint matrix in smaller parts, where each part could be solved more easily. This can done using standard mathematical programming techniques, as shown e.g., in Bazaraa et al. (1990). Then, a branch-and-bound algorithm is proposed to the resulting problem. In this branch-and-bound, the lower bounding procedure uses the decomposition found initially to improve the efficiency of the lower bound computation.

2.3.6 Minimizing Bandwidth Utilization

A problem that usually happens when constructing multicast trees is the tradeoff between bandwidth used and total cost of the tree. Traditional algorithms for tree minimization try to reduce the total cost of the tree. However, this in general does not guarantee minimum bandwidth utilization. On the other hand, there are algorithms for minimization of the bandwidth that do not maintain the minimum cost. For example, a greedy algorithm, as described in Fujinoki and Christensen (1999), works by connecting destinations sequentially to the source. Each destination is linked to the nearest node already connected to the source. In this way, bandwidth is saved by reusing existing paths.

Fujinoki and Christensen (1999) proposed a new algorithm for maintaining dynamic multicast trees which try to solve the tradeoff problem discussed above. The algorithm, called "shortest best path tree" (SBPT), uses shortest paths to connect sources to destinations. The authors represent distance between nodes as the minimum number of edges in the path between them.

The first phase in the algorithm consists of computing the shortest path from s to all destinations d_i. In the second phase, the algorithm performs a sequence of steps for each destination $d_i \in D$. Initially, it computes the shortest paths from d_i to all other nodes in G. Then, the algorithm takes the node u which has minimum distance from d_i and at the same time occurs in one of the shortest paths from s to d_i. By doing this choice, the method tries to favor the nodes already in the routing tree giving the smallest possible increase in the total cost.

2.3.7 The Degree-constrained Steiner Problem

If the number of links from any node in the network is required to be a fixed value, then we have the degree-constrained version of the multicast routing problem. For some applications of multicasting is difficult to make a large number of copies of the same data. This is particularly true for high speed switches, where the speed requirements may prohibit in practice an unbounded number of copies of the received information. For example, in ATM networks, the number of out connections can have a fixed limit (Zhong et al., 1993). Thus, it is interesting to consider Steiner tree problems where the degree of each node is constrained.

Bauer (1996) proposed algorithms for this version of the problem, and tried to construct degree-constrained multicast trees as a solution. Bauer and Varma (1995) reviewed the traditional heuristics for Steiner tree, and new

heuristics were given, which consider the restriction in the number of adjacent nodes. They show that the heuristics for degree-constrained Steiner tree give solutions very close to the optimum for sample instances of the general Steiner problem. They also show experimentally that, despite the restriction on the node degrees, almost all instances have feasible solutions which have been found by the heuristics.

2.3.8 Other Restrictions: Non Symmetric Links and Degree Variation

An interesting feature of real networks, which is not mentioned in most of the research papers, is that links are, in general, non symmetric. The capacity in one direction can be different from the capacity in the other direction, for example, due to congestion problems in some links. Ramanathan (1996) considered this kind of restriction. In his work, the minimum cost routing tree is modeled as a minimum Steiner tree with constraints, where the network has non symmetric links. The author proposes an approximation algorithm, with fixed worst case guarantee. The resulting algorithm has also the nice characteristic of being parameterizable, and therefore it allows the trading of execution time for accuracy.

Another restriction, which is normally disregarded, was considered in the approach taken by Rouskas and Baldine (1996), who proposed the minimization of the so called *delay variation*. The delay variation is defined as the difference between the minimum and maximum delay defined by a specific routing tree. In some applications it is interesting that this variation stay within a specific range. For example, it can be desirable that all nodes receive the same information at about the same time.

Table 2–1: Comparison among algorithms for the problem of multicast routing with delay constraints. * k is the number of destinations. ** This algorithm is partially distributed.

Algorithm	Guarantee	Complexity	Types of instances
KMB (Kou et al., 1981)	2	$\mathcal{O}(kn^2)$ *	general
Takahashi and Matsuyama (1980)	2	$\mathcal{O}(kn^2)$ *	general
Kompella et al. (1993b)	—	$\mathcal{O}(n^3\Delta)$	general
Sriram et al. (1998)	—	N/A **	sparse, static groups
Feng and Yum (1999)	—	$\mathcal{O}(n^2)$	general
Kumar et al. (1999)	—	$\mathcal{O}(n^3)$	center based
Jiang (1992)	—	$\mathcal{O}(n^3)$	capacity constrained, videoconferencing
Chung et al. (1997)	—	$\mathcal{O}(n^3)$	sparse instances
Zhu et al. (1995)	—	$\mathcal{O}(kn^3 \log n)$	sparse instances

2.3.9 Comparison of Algorithms

A Comparison of Non-distributed Approaches

Table 2–1 gives a summary of features of the algorithms for the Steiner tree problem with delay constraints discussed in this section. Most of them have similar computational complexity of the order of $\mathcal{O}(n^3)$, where n is the number of nodes in the network. The best result is obtained by Feng and Yum (1999), which combine the work of finding good solutions in terms of cost and delay. The heuristic by Chung et al. (1997) is reported to run faster then other heuristics, however it must be noted that it is optimized for sparse instances.

Regarding approximation, only the first two algorithms in the table have known approximation guarantee (constant and equal to 2). However, it is not difficult to achieve similar performance guarantees on heuristics with the same complexity as KMB, for example. This can be performed by running the heuristic with known performance guarantee, followed by the normal heuristic, and then reporting the best solution.

Table 2–2: Comparison among algorithms for the problem of multicast routing with delay constraints. * k is the number of destinations, T^{SP} is the time to find a shortest path in the graph. ** In this case amortized time is the important issue, but was not analyzed in the original paper.

Algorithm	Complexity	Types of instances
Kompella et al. (1993a)	$\mathcal{O}(n^3)$	general on-line instances
Baoxian et al. (2000)	$\mathcal{O}(mn)$	based on unicast information
Sriram et al. (1999)	$\mathcal{O}(n^3)$	instances with QoS
Hong et al. (1998)	$\mathcal{O}(mk(k+T^{SP})$ *	dynamic, delay sensitive
Kheong et al. (2001)	N/A **	general instances, cache must be maintained

A Comparison of On-Line Approaches

Table 2–2 presents a comparison of algorithms proposed for the on-line version of the Steiner tree problem with delay constraints, as discussed in Section 2.3.3. These algorithms in general do not provide a guarantee of approximation, due to the dynamic nature of the problem.

From the algorithms shown in Table 2–2, the one with lowest complexity is given by Baoxian et al. (2000). However, this complexity is kept low due to the dependence of the algorithm on information given by other protocols operating unicast routing. Hong et al. (1998), however, consider the construction of a complete solution, and have the on-line issues as an additional feature. Kheong et al. (2001) also consider a technique where information is reused, but in this case from previous iterations of the algorithm. It is difficult to evaluate the complexity of the whole algorithm, since it depends on the amortized complexity on a large number of iterations. This kind of analysis is not carried out in the paper.

A Comparison of Distributed Approaches

Distributed approaches for the Steiner tree problem with delay constraints are more difficult to evaluate in the sense that other features become important. For example, in distributed algorithms the message complexity, i.e., the number of message exchanges, is an important indicator of performance. These factors are sometimes not derived explicitly in some of the papers.

For the algorithm proposed by Chen et al. (1993), the message complexity is shown to be $\mathcal{O}(m + n(n + \log n))$, and the time complexity is $\mathcal{O}(n^2)$. Also, the worst-case ratio of the solution obtained to the cost of any given minimum cost Steiner tree T is $2(1 - 1/l)$, where l is the number of leaves in T.

On the other hand, Shaikh and Shin (1997) do not give much information about the complexity of their distributed algorithm. It can be noted however that the complexity is similar to that of distributed algorithms for the computation of a minimum spanning tree. Finally, Mokbel et al. (1999) derive only the total time complexity of their algorithm, which is $\mathcal{O}(K^2n^2)$, where K is a constant value introduced to decrease the number of message exchanges required.

2.4 Other Problems in Multicast Routing

In this section we present some other problems occurring in multicast routing and which have interesting characteristics, in terms of combinatorial optimization. The first of these problems is the *multicast packing problem*, where the objective is to optimize the design of the entire network in order to provide capacity for a specific number of multicast groups. Then, we discuss the *point-to-point connection problem*, which is a generalization of the Steiner tree problem.

2.4.1 The Multicast Packing Problem

A more general view of the multicast routing problem can be found if we consider the required constraints when more than one multicast group exists. In this case, there is a number of applications that try to use the network for the purpose of establishing connection and sending information, organized in different groups. Thus, the network capacity must be shared accordingly with the requirements of each group. These capacity constraints are modeled in what is called the *multicast packing problem* in networks. This problem has attracted some attention in the past few years (Wang et al., 2002; Priwan et al., 1995; Chen et al., 1998).

The congestion λ_e on edge e is given by the sum of all load imposed by the groups using e. The maximum congestion λ is then defined as the maximum of all congestion λ_e, over edges $e \in E$. If we assume that there are K multicast groups, and each group k generates an amount t^k of traffic, an integer programming formulation for the multicast packing problem is given by

$$\min \lambda \tag{2.8}$$

subject to

$$\sum_{i=1}^{K} t^k x_e^k \leq \lambda \qquad \text{for all } e \in E \tag{2.9}$$

$$x_e^k \in \{0,1\}^{|E|} \qquad \text{fox } i = 1, \ldots, K, \tag{2.10}$$

where variable x_e^k is equal to one if and only if the edge e is being used by multicast group k.

A number of approaches have been proposed for solving this problem. For example, Wang et al. (2002) discuss how to set up multiple groups using routing trees, and formalized this as a packing problem. Two heuristics were

then proposed. The first one is based on known heuristics for constructing Steiner trees. The second is based on the cut-set problem. The constraints considered for the Steiner tree problem are, first, the minimum cost under bounded tree depth; and second, the cost minimization under bounded degree for intermediate nodes.

Priwan et al. (1995) and Chen et al. (1998) proposed formulations for the multicast packing problem using integer programming. The last authors considered two ways of modeling the routing of multicast information. In the first method, the information is sent according to a minimum cost tree among nodes in the group, and therefore give rise to the Steiner tree problem. In the second, more interesting version, the information is sent through a ring which visits all elements in the group, and therefore this results in a problem similar to the traveling salesman problem. Using these formulations they describe heuristics that can be applied to get approximate solutions. Comparisons were done between the two proposed formulations with respect to the quality of the solutions found to the multicast packing problem.

In the integer formulation of the multicast packing problem, there is a variable x_e for each edge $e \in E$, which is equal to one if and only if this edge is selected. Each edge has also an associated cost w_e. Then, the integer formulation for the tree version of the problem is given by

$$\min \sum_{e \in E} w_e x_e \tag{2.11}$$

subject to

$$\sum_{e \in \delta(S)} x_e \geq 1 \quad \text{for all } S \subset V \text{ such that } m_1 \in S \text{ and } M \not\subset S \tag{2.12}$$

$$x \in \{0,1\}^{|E|}, \tag{2.13}$$

where M is the set of nodes participating in a multicast group and $\delta(S)$ represents the edges leaving the set $S \subset V$. The integer program for the ring-based version is given by

$$\min \sum_{e \in E} w_e x_e \tag{2.14}$$

subject to

$$\sum_{e \in \delta(v)} x_e = 2 \quad \text{for all } v \in M \tag{2.15}$$

$$\sum_{e \in \delta(v)} x_e \leq 2 \quad \text{for all } v \in V \setminus M \tag{2.16}$$

$$\sum_{e \in \delta(S)} x_e \geq 2 \quad \text{for all } S \subset V \text{ s.t. } u \in S \text{ and } M \not\subset S \tag{2.17}$$

$$x \in \{0,1\}^{|E|}. \tag{2.18}$$

Here, u is any element of M. The integer solution of this problem defines a ring passing through all nodes participating in group M. In Chen et al. (1998) these two problems are solved using *branch-and-cut* techniques, after the identification of some valid inequalities.

2.4.2 The Multicast Network Dimensioning Problem

Another interesting problem occurs when we consider the design of a new network, intended to support a specific multicast demand. This is called the *multicast network dimensioning problem*, and it has been treated in some recent papers (Prytz, 2002; Forsgren and Prytz, 2002; Prytz and Forsgren, 2002).

According to Forsgren and Prytz (2002), the problem consists of determining the topology (which edges will be selected) and the corresponding capacity of the edges, such that a multicast service can be deployed in the resulting network. Much of the work for this problem has used mathematical programming techniques to define and give exact and approximate solutions to the problem.

The technique used in Forsgren and Prytz (2002) has been the Lagrangian relaxation applied to an integer programming model. We assume that there are T multicast groups. The model uses variables $x_e^k \in \{0,1\}$, for $k \in \{1,\ldots,T\}$, $e \in E$, which represent if edge e is used by group k. There are also variables $z_e^l \in \{0,1\}$, for $l \in \{1,\ldots,L\}$, $e \in E$, where L is the highest possible capacity level, which determine if the capacity level of edge e is equal to l. Now, let d_k, for $k \in \{1,\ldots,T\}$, be the bandwidth demanded by group k; c_e^l, for $l \in \{1,\ldots,L\}$, and $e \in E$, be the capacity available for edge e at the level l; and w_e^l, for $l \in \{1,\ldots,L\}$, and $e \in E$, be the cost of using edge e at the capacity level l. Also, $b \in Z^n$ is the demand vector, and $A \in R^{n\times m}$ is the node-edge incidence matrix. We can now state the multicast network dimensioning problem using the following integer program

$$\min \quad \sum_{e\in E}\sum_{l=1}^{L} w_e^l z_e^l \tag{2.19}$$

$$\text{subject to} \quad \sum_{k=1}^{T} d^k x_e^k \le \sum_{l=1}^{L} c_e^l z_e^l \qquad \text{for all } e \in E \tag{2.20}$$

$$\sum_{l\in L} z_e^l \le 1 \qquad \text{for all } e \in E \tag{2.21}$$

$$Ax = b \tag{2.22}$$

$$x, z \in \mathbb{Z}. \tag{2.23}$$

In this integer program, constraint (2.20) ensures that the bandwidth used on each edges is at most the available capacity. Constraint (2.21) selects just one capacity level for each edge. Finally, constraint (2.22) enforces the flow conservation in the resulting solution.

The problem proposed above has been solved using a *branch-and-cut* algorithm, employing some basic types of cuts. The authors also use Lagrangian relaxation to reduce the size of the linear program that need to be solved. Some primal heuristics have been designed to exploit certain similarities with

the Steiner tree problem. These primal heuristics were used to improve the upper bounds found during the branching phase. The resulting algorithm has been able to solve instances with more than 100 nodes.

2.4.3 The Point-to-Point Connection Problem

An interesting generalization of the Steiner problem is known as the *point-to-point connection* problem (PPCP). In the PPCP, we are given two disjoint sets S and D of sources and, respectively, destinations. We require that $|S| = |D|$. However, that is not an important restriction, since for every network we can extend the set of sources using dummy nodes, if needed. As usual, there is a cost function $w : E \to N$. The objective is to find a minimum cost forest $F \subseteq E$, where each destination is connected to at least one source, and similarly each source is connected to at least one destination.

This problem was first proposed by Li et al. (1992), who proved that all four versions of the PPCP (directed, undirected, with fixed or non-fixed destinations) are $\mathcal{NP}$-hard, when p is given as input. Natu (1995) proposed a dynamic programming algorithm for $p = 2$ with time complexity $\mathcal{O}(mn + n^2 \log n)$. Goemans and Williamson (1995) presented an approximation algorithm for a class of forest constrained problems, including the PPCP and the Steiner problem, that runs in $\mathcal{O}(n^2 \log n)$ and gives its results within a factor of $2 - 1/p$ of the optimal solution.

The PPCP is useful to model situations where there are multiple sources. Some metaheuristic algorithms have been applied to the problem by Correa et al. (2003), and Gomes et al. (1998). The main idea of these methods is to design simple heuristics and combine them in a framework called *asynchronous teams* (Talukdar and de Souza, 1990), where each heuristic is considered an autonomous agent, capable of improving the existing solutions.

Some of the heuristics proposed in Correa et al. (2003) explore basic features of optimal solutions. For example, one of the heuristics uses the *triangle inequality* property: given three nodes a, b, and c in one of the paths in the solution, the cost of paths between a and b and between b and c must be at most the minimum path between b and c. Given a solution, we can check for each three nodes a, b, and c in a path, if this condition is satisfied. If it is not, then we can always improve the solution by making the correct substitution, using the minimum path.

2.5 Concluding Remarks

In this paper we have shown a number of applications and problems associated with multicast routing. We have also shown that most of them are related to other important problems in the area of combinatorial optimization. The topics addressed show that this is an evolving area, still in its development stages. Moreover, most of the interesting problems can be addressed with techniques developed by the combinatorial optimization and operations research communities.

We believe that in the next years an increased number of applications and models will continue to evolve from this field and make it an important source of problems and results.

CHAPTER 3
STREAMING CACHE PLACEMENT PROBLEMS

We study a problem in the area of multicast networks, called the streaming cache placement problem (SCPP). In the SCPP one wants to determine the minimum number of multicast routers needed to deliver content to a specified number of destinations, subject to predetermined link capacities. We initially discuss the different versions of the SCPP found in multicast networks applications. Then, a transformation from the SATISFIABILITY problem is used in order to prove $\mathcal{NP}$-hardness to all of these versions of the SCPP. Complexity results are derived for the cases of directed and undirected graphs, as well as with different assumptions about the type of flow in the network.

3.1 Introduction

Multicast protocols are used to send information from one or more sources to a large number of destinations using a single *send operation*. Networks supporting multicast protocols have become increasingly important for many organizations due to the large number of applications of multicasting, which include data distribution, video-conferencing (Eriksson, 1994), groupware (Chockler et al., 1996), and automatic software updates (Han and Shahmehri, 2000). Due to the lack of multicast support in existing networks, there is an arising need for updating unicast oriented networks. Thus, there is a clear economical impact in providing support for new multicast enabled applications.

In this chapter we study a problem motivated by the economical planning of multicast network implementations. The streaming cache placement problem (SCPP) has the objective of minimizing costs associated with the

implementation of multicast routers. This problem has only recently received attention (Mao et al., 2003; Oliveira et al., 2003a) and presents many interesting questions still unanswered from the algorithmic and complexity theoretic point of view.

3.1.1 Multicast Networks

In multicast networks, nodes interested in a particular piece of data are called a *multicast group*. The main objective of such groups is to send data to destinations in the most efficient way, avoiding duplication of transmissions, and therefore saving bandwidth. With this aim, special purpose multicast protocols have being devised in the literature. Examples are the PIM (Deering et al., 1996) and core-based (Ballardie et al., 1993) distribution protocols. The basic operation in these routing protocols is to send data for a subset of nodes, duplicating the information only when necessary.

Network nodes that understand a multicast protocol are called *cache nodes*, because they can send multiple copies of the received data. Other nodes simply act as intermediates in the multicast transmission. The main problem to be solved is deciding the route to be used by packages in such a network. One of the simplest strategies for generating multicast routes is to maintain a *routing tree*, linking all sources and destinations. A similar strategy, which may reduce the number of needed cache nodes, consists in determining a *feasible flow* from sources to destinations such that all destinations can be satisfied.

A main economical problem, however, is that not all nodes understand these multicast routing protocol. Moreover, upgrading all existing nodes can be expensive (or even impossible, when the whole network is not owned by the same company, as happens in the Internet).

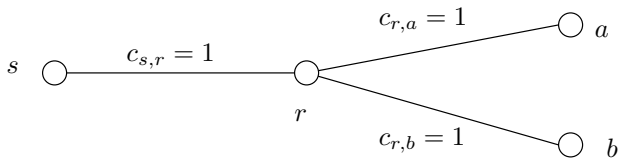

Figure 3–1: Simple example for the cache placement problem.

Suppose an extreme situation, where no nodes have multicast capabilities. In this case, the only possible solution consists in sending a separate copy of the required data to each destination in the group. However, in this case instances can become quickly infeasible, as shown in Figure 3–1, Here, all edges have capacity equal to one, and nodes a and b are destinations. In this example, a feasible solution is found when r becomes a cache node. Thus, it is interesting to determine the minimum number of cache nodes required to handle a specified amount of multicast traffic, subject to link capacity constraints. This is called the *streaming cache placement problem* (SCPP).

Formal Description of the SCPP. Suppose that a graph $G = (V, E)$ is given, with a capacity function $c : E \to Z_+$, a distinguished source node $s \in V$ and a set of destination nodes $D \subset V$. It is required that data be sent from node s to each destination. Thus, we must determine a set R of cache nodes, used to retransmit data when necessary, and the amount of information carried by each edge, which is represented by variables $w_e \in \mathbb{R}_+$, such that $w_e \leq c_e$, for $e \in E$. The objective of the SCPP is to find a set R of minimum size corresponding to a flow $\{w_e | e \in E\}$, such that for each node $v \in D \cup R$ there is a unit flow from some node in $\{s\} \cup R$ to v, and the capacity constraints $w_e \leq c_e$, for $e \in E$, are satisfied.

A characterization of the set of cache nodes can be given in terms of the surplus of data at each node $v \in V$. Suppose that the number of data units sent by node v, also called *surplus*, is given by variable $b_v \in \mathbb{Z}$. Note that

the node s must send at least one unit of information, so b_s must be greater than zero. Each destination is required to receive a unit of data, so it has a negative surplus (requirement) of -1. Now, suppose that, due to capacity constraints, we need to establish v as a cache node. Then, the surplus at this node cannot be negative, since it is also sending forward the received data. If v is also a destination, than the minimum surplus is zero (in this case it is receiving one unit and sending one unit); otherwise, $b_v \geq 1$. Thus, the set of cache nodes $R \subseteq V \setminus \{s\}$ is the one such that $b_v \geq 0$ and $v \in D$ or $b_v > 0$ and $v \in V \setminus D \cup \{s\}$, for all $v \in R$.

3.1.2 Related Work

Problems in multicast routing have been investigated by a large number of researchers in the last decade. The most studied problems relate to the design of routing tables with optimal cost. In these problems, given a set of sources and a set of destinations, the objective is to send data from sources to destinations with minimum cost.

In the case in which there are no additional constraints, this reduces to the Steiner tree problem on graphs (Du et al., 2001). In other words, it is required to find a tree linking all destinations to the source, with minimum cost. Using this technique, the source and destinations are the *required nodes*, the remaining ones being the Steiner nodes. Many heuristic algorithms have been proposed for this kind of problem (Chow, 1991; Feng and Yum, 1999; Kompella et al., 1993b,a; Kumar et al., 1999; Salama et al., 1997b; Sriram et al., 1998).

The problems above, however, consider that all nodes support a multicast protocol. This is not a realistic assumption on existing networks, since most routers do not support multicasting by default. Thus, some sort of upgrade

must be applied, in terms of software or even hardware, in order to deploy multicast applications. Despite this important application, only recently researchers have started to look at this kind of problem.

In Mao et al. (2003) the Streaming Cache Placement problem is defined, in the context of Virtual Private Networks (VPNs). In this paper, the SCPP was proven to be $\mathcal{NP}$-hard, using a reduction from the Exact Cover by 3-Sets problem, and a heuristic was proposed to solve some sample instances. However, the paper does not give details about possible versions of the problem, and proceeds directly to deriving local search heuristics.

Another related problem is the Cache Placement Problem (Li et al., 1999). Here, the objective is to place replicas of some static document on different points of a network, in order to increase accessibility and also decrease the average access time by any client. The important difference between this problem and the SCPP is that Li et al. (1999) does not consider multicast transmissions. Also, there are no restriction on the capacity of links, and data is considered to be placed at the locations before the real operation of the network.

3.2 Versions of Streaming Cache Placement Problems

In this section, we discuss two versions of the SCPP. In the *tree streaming cache placement problem* (TSCPP), the objective is to find a routing tree which minimizes the number of cache nodes needed to send data from a source to a set of destinations. We also discuss a modification of this problem where we try to find any feasible flow from source to destinations, minimizing the number of cache nodes. The problem is called the *flow streaming cache placement problem* (FSCPP).

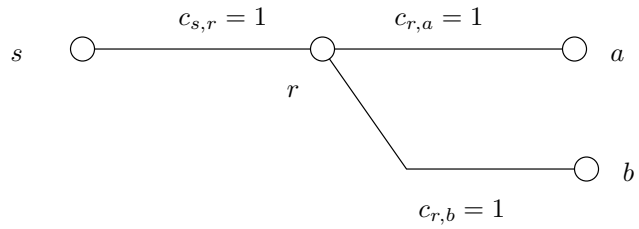

Figure 3–2: Simple example for the Tree Cache Placement Problem.

3.2.1 The Tree Cache Placement Problem

Consider a weighted, capacitated network $G(V, E)$ with a source node s and a routing tree T rooted on s and spanning all nodes in V. Let D be a subset of the nodes in V that have a demand for a data stream to be sent from node s. The stream follows the path defined by T from s to the demand nodes and takes B units of bandwidth on every edge that it traverses. For each demand node, a separate copy of the stream is sent. Edge capacities cannot be violated. Note that, depending on the network structure, an instance of this problem can easily become infeasible.

To handle this, we allow stream splitters, or caches, to be located at specific nodes in the network. A single copy of the stream is sent from s to a *cache node* r and from there multiple copies are sent down the tree. The optimization problem consists in finding a routing tree and to locate a minimum number of cache nodes.

Figure 3–2 shows an small example for this problem. In this example, if nodes a and b each require a stream (with $B = 1$) from s, and node r is not a cache node, then we send two units from s to r and one unit from r to a and from r to b. We get an infeasibility on edge (s, r), since two units flow on it, and it has capacity $c_{s,r} = 1 < 2$. However, if node r becomes a cache node, we can send one unit from s to r and then one unit from r to a and one unit from r to b. The resulting flow is now feasible.

To simplify the formulation of the problem, we can consider, without loss of generality, that the bandwidth used by each message is equal to one.

The tree cache placement problem (TSCPP) is defined as follows. Given a graph $G(V, E)$ with capacities c_{uv} on the edges, a source node $s \in V$ and a subset $D \subset V$ representing the destination nodes, we want to find a spanning tree T (which determines the paths followed by a data stream from s to $v \in D$) such that the subset $R \subseteq V \setminus \{s\}$, which represents the cache nodes, has minimum size. For each node $v \in D \cup R$, there must be a data stream from some node $w \in R \cup \{s\}$ to v, such that the sum of all streams in each edge $(i, j) \in T$ does not exceed the edge capacity c_{ij}.

To state the problem more formally, consider an integer programming formulation for the TSCPP. Define the variables

$$y_e = \begin{cases} 1 & \text{if edge } e \text{ is in the spanning tree } T \\ 0 & \text{otherwise,} \end{cases}$$

$$x_i = \begin{cases} 1 & \text{if node } i \neq s \text{ is a cache node} \\ 0 & \text{otherwise} \end{cases}$$

$$b_i \in \{-1, \ldots, |V| - 1\} \qquad \text{the flow surplus for node } i \in V$$

$$w_e \in \{0, \ldots, |V|\} \qquad \text{the amount of flow in edge } e \in E.$$

Given the node-arc incidence matrix A, the problem can be stated as

$$\min \sum_{i=1}^{|V|} x_i \tag{3.1}$$

subject to

$$Aw = b \tag{3.2}$$

$$\sum_{i \in V} b_i = 0 \tag{3.3}$$

$$b_s \geq 1 \qquad \text{for source } s \tag{3.4}$$

$$x_i - 1 \leq b_i \leq x_i(|V|-1) - 1 \qquad \text{for } i \in D \tag{3.5}$$

$$x_i \leq b_i \leq x_i(|V|-1) \qquad \text{for } i \in V - (D \cup \{s\}) \tag{3.6}$$

$$\sum_{e \in E} y_e = |V| - 1 \tag{3.7}$$

$$\sum_{e \in G(H)} y_e \leq |H| - 1 \qquad \text{for all } H \subset V \tag{3.8}$$

$$0 \leq w_e \leq c_e y_e \qquad \text{for } e \in E \tag{3.9}$$

$$x \in \{0,1\}^{|V|}, \quad y \in \{0,1\}^{|E|} \tag{3.10}$$

$$b \in Z, \quad w \in Z^+, \tag{3.11}$$

where $G(H)$ is the subgraph induced by the nodes in H. Constraint (3.2) imposes flow conservation. Constraints (3.3) through (3.6) require that there must be a number of data streams equal to the number of nodes in $R \cup D$. Constraints (3.7) and (3.8) are the spanning tree constraints. Finally, constraint (3.9) determine the bounds for flow variables, implying that the flow specified by w can be carried only on edges in the spanning tree.

3.2.2 The Flow Cache Placement Problem

An interesting extension of the TSCPP arises if we relax the constraints in the previous integer programming formulation that require the solution to be a tree of the graph G. Then we have the more general case of a flow sent from the source node s to the set of destination nodes D. To see why this extension is interesting, consider the example graph, shown in Figure 3–3. In this example all edges have costs equal to one. If we find a solution to the TSCPP on this graph, then a stream can be sent through only one of the two edges (s, a) or (s, b). Suppose that we use edges (s, a) and (a, c). This implies that c must be a cache node, in order to satisfy demand nodes d_1 and

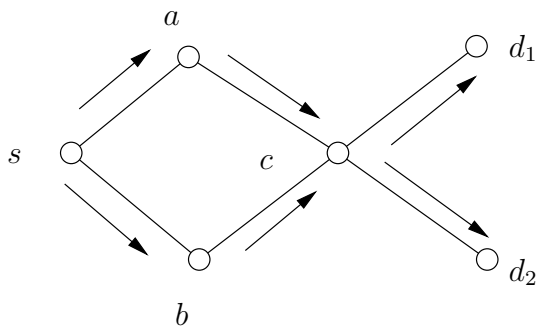

Figure 3–3: Simple example for the Flow Cache Placement Problem.

d_2. However, in practice the number of caches in this optimal solution for the TSCPP can be further reduced.

Routing protocols, like OSPF, achieve load balancing by sending data through parallel links. In the case of Figure 3–3, the protocol could just send another stream of data over edges (s, b) and (b, c). If this happens, we do not need a cache node, and the solution will have fewer caches.

We define the Flow Cache Placement Problem (FSCPP) to be the problem of finding a feasible flow from source s to the set of destinations D, such that the number of required caches $R \subseteq V \setminus \{s\}$ is minimized. The integer linear programming model for this problem is similar to (3.1)-(3.11), without the integer variable y and relaxing constraints (3.7)-(3.8).

3.3 Complexity of the Cache Placement Problems

We prove that both versions of the SCPP discussed above are $\mathcal{NP}$-hard, using a transformation from Satisfiability. This transformation allows us to give a proof of non-approximability by showing that it is a gap-preserving transformation.

3.3.1 Complexity of the TSCPP

In this section we prove that the TSCPP is $\mathcal{NP}$-hard, by using a reduction from Satisfiability (SAT) (Garey and Johnson, 1979).

SAT: Given a set of clauses $C_1, \ldots, C_m$, where each clause is the disjunction of $|C_i|$ literals (each literal is a variable $x_j \in \{x_1, \ldots, x_n\}$ or its negation $\overline{x}_j$), is there a truth assignment for variables $x_1, \ldots, x_n$ such that all clauses are satisfied?

Definition 1 *The TSCPP-D problem is the following. Given an instance of the TSCPP and an integer k, is there a solution such that the number of cache nodes needed is at most k?*

Theorem 2 *The TSCPP-D problem is $\mathcal{NP}$-complete.*

Proof: This problem is clearly in $\mathcal{NP}$, since for each instance I it is enough to give the spanning tree and the nodes in R to determine, in polynomial time, if this is a "yes" instance.

We reduce SAT to TSCPP-D. Given an instance I of SAT, composed of m clauses $C_1, \ldots, C_m$ and n variables $x_1, \ldots, x_n$, we build a graph $G(V, E)$, with $c_e = 1$ for all $e \in E$, and choose $k = n$. The set V is defined as

$$V = \{s\} \cup \{x_1, \ldots, x_n\} \cup \{\overline{x}_1, \ldots, \overline{x}_n\} \cup \{T'_1, \ldots, T'_n\}$$

$$\cup \{T''_1, \ldots, T''_n\} \cup \{T'''_1, \ldots, T'''_n\} \cup \{C_1, \ldots, C_m\},$$

and the set E is defined as

$$E = \bigcup_{i=1}^{n} \{(s, x_i), (s, \overline{x}_i)\} \cup \bigcup_{i=1}^{n} \{(x_i, T'_i), (\overline{x}_i, T'_i)\} \cup \bigcup_{i=1}^{n} \{(x_i, T''_i)\}$$

$$\cup \bigcup_{i=1}^{n} \{(\overline{x}_i, T'''_i)\} \cup \bigcup_{i=1}^{m} \left\{ \bigcup_{x_j \in C_i} (x_j, C_i) \bigcup_{\overline{x}_j \in C_i} (\overline{x}_j, C_i) \right\}. \tag{3.12}$$

Figure 1 shows the construction of G for a small SAT instance. Define $D = \{C_1, \ldots, C_m\} \cup \{T'_1, \ldots, T'_n\} \cup \{T''_1, \ldots, T''_n\} \cup \{T'''_1, \ldots, T'''_n\}$. Clearly, destination nodes T'_i, T''_i and T'''_i are there just to saturate the arcs leaving s and force one of $x_i, \overline{x}_i$ to be chosen as a cache node. Also, each node C_i forces

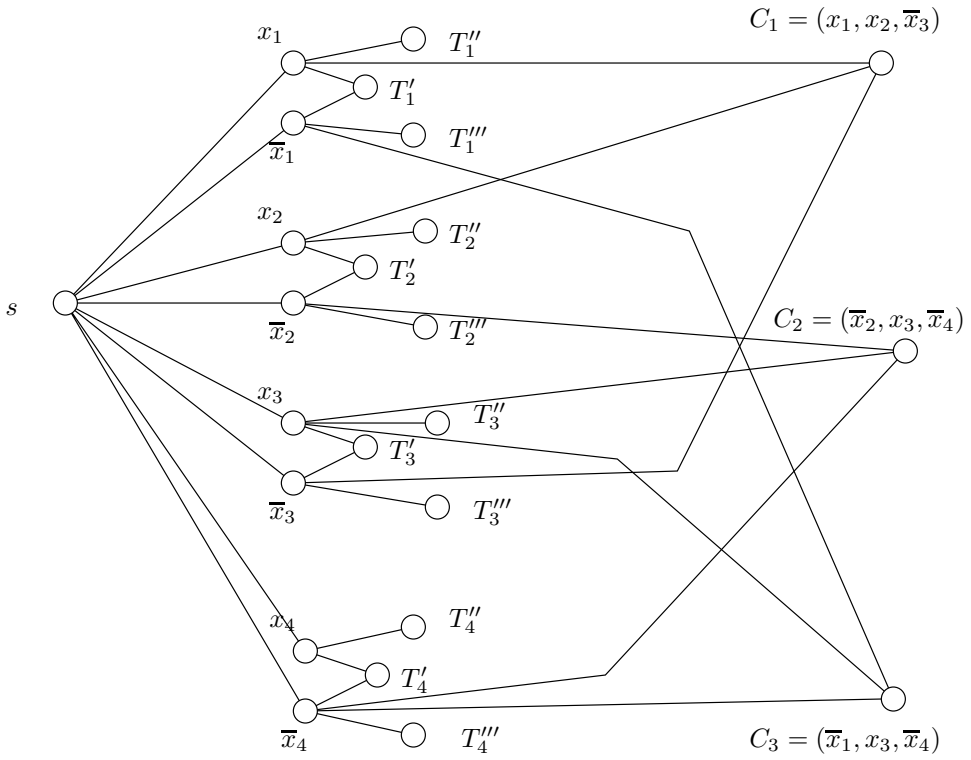

Figure 3–4: Small graph G created in the reduction given by Theorem 2. In this example, the SAT formula is $(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee x_3 \vee \overline{x}_4)$.

the existence of at least one cache among the nodes corresponding to literals appearing in clause C_i.

Suppose that the solution of the resulting TSCPP-D problem is true. Then, we assign variable x_i to true if node x_i is in R, otherwise we set x_i to false. This assignment is well-defined, since exactly one of the nodes $x_i, \overline{x}_i$ must be selected. Clearly, this truth assignment satisfies all clauses C_i, because the demand of each node C_i is satisfied by at least one node corresponding to literals appearing in clause C_i.

Conversely, if there is a truth assignment Γ which makes the SAT formula satisfiable, we can use it to define the nodes which will be caches, and, by construction of G, all demands will be satisfied. Finally, the resulting construction is polynomial in size, thus SAT reduces in polynomial time to TSCPP-D. $\square$

```
Input: a tree T
Output: a set R of cache nodes
forall v ∈ V do
    if v ∈ D then demand(v) ← 1
    else demand(v) ← 0
end
call findR(s)
return R

procedure findR(v)
begin
    forall w such that (v, w) ∈ T do findR(w)
1   if v = s then return R
    else p ← parent(v)
    if c_{p,v} < demand(v) then
        R ← R ∪ {v}
        demand(p) ← demand(p) + 1
    end
    else demand(p) ← demand(p) + demand(v)
end
```

Algorithm 4: Find the optimal R for a fixed tree.

As a simple consequence of this theorem, we have the following corollary.

Corollary 3 *The TSCPP is $\mathcal{NP}$-hard.*

It is interesting to observe that the problem remains $\mathcal{NP}$-hard even for unitary-capacity networks, since the proof remains the same for edges with unitary capacity.

Some simple examples serve to illustrate the problem. For instance, if G is the complete graph K^n, then the optimal solution is simply a star graph with s at the center, and $R = \emptyset$. On the other hand, if the graph is a tree with n nodes, then the number of cache nodes is implied by the edges of the tree, thus the optimum is completely determined.

Algorithm 4 determines an optimal set R from a given tree T. The algorithm works recursively. Initially it finds the demand for all leaves of T. Then

it goes up the tree determining if the current nodes must be a cache node. The correctness of this method is proved bellow.

Theorem 4 *Given an instance of the TSCPP which is a tree T, then an optimal solution for T is given by Algorithm 4.*

Proof: The proof is by induction on the height h of a tree analyzed when Algorithm 4 arrives at line (1). If $h = 0$ then the number of cache nodes is clearly equal to zero. Assume that the theorem is true for trees with height $h \geq 1$. If the capacity of the arc (p, v) is greater than the demand at v, then there is no need of a new cache node, and therefore the solution remains optimal. If, on the other hand, (p, v) does not have enough capacity to satisfy all demand at v, then we do not have a choice other than making v a cache node. Combining this with the assumption that the solution for all children of v is optimal, we conclude that the new solution for a tree of height $h + 1$ is also optimal. □

3.3.2 Complexity of the FSCPP

We can use the transformation from SAT to TSCPP to show that FSCPP is also $\mathcal{NP}$-hard. In the case of directed edges, this is simple, since given a graph G provided by the reduction, we can give an orientation of G from source to destinations. This is stated in the next theorem.

Theorem 5 *The FSCPP is $\mathcal{NP}$-hard if the instance graph is directed.*

Proof: The proof is similar to the proof of Theorem 4. We need just to make sure that the polynomial transformation given for the TSCPP-D also works for a decision version of the FSCPP. Given an instance of SAT, let G be the corresponding graph found by the reduction. We orient the edges of G from S to destinations D, i.e., use the implicit orientation given in (3.12). It can be checked that in the resulting instance the number of cache nodes cannot be

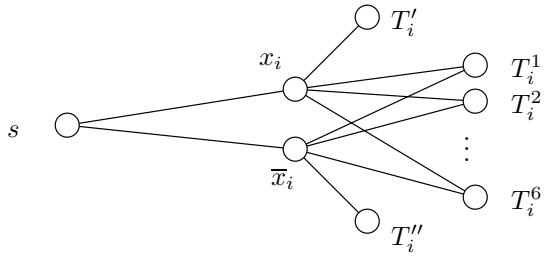

Figure 3–5: Part of the transformation used by the FSCPP.

reduced by sending additional flow in other edges other than the ones which form the tree in the solution of TSCPP. Thus, the resulting R is the same, and FSCPP is $\mathcal{NP}$-hard in this case. □

Next we prove a slightly modified theorem for the undirected version. To do this we need the following variant of SAT:

3SAT(5): Given an instance of SATISFIABILITY with at most three literals per clause and such that each variable appears in at most five clauses, is there a truth assignment that makes all clauses true?

The 3SAT(5) is well known to be $\mathcal{NP}$-complete (Garey and Johnson, 1979).

Theorem 6 *The FSCPP is $\mathcal{NP}$-hard if the instance graph is undirected.*

Proof: When the instance of FSCPP is undirected, the only thing that can go wrong is that some of the destinations T_i', T_i'', or T_i''' are being satisfied by flow coming from nodes C_j connected to their respective x_i, $\overline{x}_i$ nodes. What we need to do to prevent this is to bound the number of occurrences of each variable and add enough absorbing destinations to the subgraph corresponding to that variable. We do this by reduction from 3SAT(5). The reduction is essentially the same as the reduction from SAT to TSCPP, but now for each variable x_i we have nodes x_i, $\overline{x}_i$, T_i', T_i'', and T_i^k, for $1 \leq k \leq 6$ (see Figure 3–5). Also,

for each variable x_i we have edges (s, x_i), $(s, \overline{x}_i)$, (x_i, T_i'), $(\overline{x}_i, T_i'')$, (x_i, T_i^k), $(\overline{x}_i, T_i^k)$, for $1 \leq k \leq 6$.

We claim that in this case for each pair of nodes x_i, $\overline{x}_i$, one of them must be a cache node (which says that the corresponding variable in 3SAT(5) is true or false). This is true because from the eight destinations not corresponding to clauses (T_i', T_i'', and T_i^k, $1 \leq k \leq 6$) attached to x_i, $\overline{x}_i$, two can be directly satisfied from s without caches. However, the remaining six cannot be satisfied from nodes C_j linked to the current variable nodes, because there are at most five such nodes. Thus, we must have one cache node at x_i or $\overline{x}_i$, for each variable x_i. It is clear that these are the only cache nodes needed to make all destinations satisfied. This gives us the correct truth assignment for the original 3SAT(5) instance. Conversely, any non-satisfiable formula will transform to a FSCPP instance which needs more than n cache nodes to satisfy all destinations. Thus, the decision version of FSCPP is $\mathcal{NP}$-complete, and this implies the theorem. □

Note that there is a case of TSCPP-D that is solvable in polynomial time, and this happens when $k = 0$, i.e. determining if any cache node is needed. The solution is given by the following algorithm. Run the maximum flow algorithm from node s to all nodes in D. This can be accomplished, for example, by creating a dummy destination node d and linking all nodes $v \in D$ to d by arcs with capacity equal to 1. If the maximum flow from s reaches each node in D, then the answer is true, since no cache node is needed to satisfy the destinations. Otherwise, the answer must be false because then at least one cache node is needed to satisfy all nodes in D.

3.4 Concluding Remarks

In this chapter we presented and analyzed two combinatorial optimization problems, the tree cache placement problem (TSCPP) and its flow-based, generalized version, the flow cache placement problem (FSCPP). We prove that both problems, on directed and undirected graphs, are $\mathcal{NP}$-hard. For this purpose, we use a transformation from the SATISFIABILITY problem.

Many questions remain open for these problems. For example, it would be interesting to find algorithms with a better approximation guarantee, or improved non-approximability results. Some of these issues will be considered in the next chapters.

CHAPTER 4
COMPLEXITY OF APPROXIMATION FOR STREAMING CACHE PLACEMENT PROBLEMS

As shown in the previous chapter, the SCPP in its two forms is $\mathcal{NP}$-hard. We improve the hardness results for the SCPP, by showing that it is very difficult to give approximate solutions for such problems. General non-approximability is proved using the reduction from SATISFIABILITY given in the previous chapter. Then, we improve the approximation results for the FSCPP using a reduction from SET COVER. In particular, given k destinations, we show that the FSCPP cannot have a $\mathcal{O}(\log\log k - \delta)$-approximation algorithm, for a very small δ, unless $\mathcal{NP}$can be solved in sub-exponential time.

4.1 Introduction

We continue in this chapter the study of the streaming cache placement problem (SCPP). In the SCPP one wants to determine the minimum number of multicast routers needed to deliver content to a specified number of destinations, subject to predetermined link capacities. The SCPP is known to be $\mathcal{NP}$-hard, as shown Chapter 3. We give approximation results for the SCPP in its different versions, using properties of the SATISFIABILITY problem. We use the transformation described in the previous chapter to achieve this non-approximability result. We show that there is a fixed $\epsilon > 1$ such that no SCPP problem can be approximated in polynomial time with guarantee better than ϵ. This is equivalent to say that the SCPP is in the MAX SNP-hard class (Papadimitriou and Yannakakis, 1991).

We are also able to improve the approximation results for the FSCPP, using a reduction from Set Cover. In this case, we are interested in general flows and directed arcs. In particular, given k destinations, we show that the FSCPP cannot have a $\mathcal{O}(\log\log k - \delta)$-approximation algorithm, for a very small δ, unless $\mathcal{NP}$can be solved in sub-exponential time.

This chapter is organized as follows. In Section 4.2 we discuss the non-approximation result for FSCPP based on the Satisfiability problem. Then, in Section 4.3, we discuss the improved result for the FSCPP based on Set Cover. Section 4.4 gives some concluding remarks.

4.2 Non-approximability

The transformation used in Theorem 2 provides a method for proving a non-approximability result for the TSCPP and FSCPP. We employ standard techniques, based on the gap-preserving transformations. To do this we use an optimization version of 3Sat(5).

Max-3Sat(5): Given an instance of 3Sat(5), find the maximum number of clauses that can be satisfied by any truth assignment.

Definition 2 *For any ϵ, $0 < \epsilon < 1$, an approximation algorithm with guarantee ϵ (or equivalently, an ϵ-approximation algorithm) for a maximization problem Π is an algorithm A such that, for any instance $I \in \Pi$, the resulting cost $A(I)$ of A applied to instance I satisfies $\epsilon \cdot OPT(I) \leq A(I)$, where we denote by $OPT(I)$ the cost of the optimum solution. For minimization problems, $A(I)$ must satisfy $A(I) \leq \epsilon \cdot OPT(I)$, for any fixed $\epsilon > 1$.*

The following theorem from Arora and Lund (1996) is very useful to prove hardness of approximation results.

Theorem 7 (Arora and Lund (1996)) *There is a polynomial time reduction from SAT to* Max-3Sat(5) *which transforms formula ϕ into a formula ϕ' such that, for some fixed ϵ (ϵ is in fact determined in the proof of the theorem),*

- *if ϕ is satisfiable, then $OPT(\phi') = m$, and*
- *if ϕ is not satisfiable, then $OPT(\phi') < (1-\epsilon)m$,*

where m is the number of clauses in ϕ'.

In the following theorem we use this fact to show a non-approximability result for TSCPP.

Theorem 8 *The transformation used in the proof of Theorem 2 is a gap-preserving transformation from* Max-3SAT(5) *to TSCPP. In other words, given an instance ϕ of* Max-3SAT(5) *with m clauses and n variables, we can find an instance I of TSCPP such that*

- *If $OPT(\phi) = m$ then $OPT(I) = n$; and*
- *If $OPT(\phi) \leq (1-\epsilon)m$ then $OPT(I) \geq (1+\epsilon_1)n$.*

where ϵ is given in Theorem 7 and $\epsilon_1 = \epsilon/15$.

Proof: Suppose that ϕ is an instance of Max-3SAT(5). Then, we can use the transformation given in the proof of Theorem 2 to construct a corresponding instance I of TSCPP. If ϕ has a solution with $OPT(\phi) = m$, where m is the number of clauses, then by Theorem 2, we can find a solution for I such that $OPT(I) = n$.

Now, if $OPT(\phi) \leq (1-\epsilon)m$, then there are at least ϵm clauses unsatisfied. In the corresponding instance I we have at least n cache nodes due to the constraints from nodes T_i', T_i'' and T_i''', $1 \leq i \leq n$. These cache nodes satisfy at most $(1-\epsilon)m$ destinations corresponding to clauses. Let U be the set of unsatisfied destinations. The nodes in U can be satisfied by setting one extra cache (in a total of two, for nodes x_j and $\overline{x}_j$) for at least one variable x_j appearing in the clause corresponding to c_i, for all $c_i \in U$.

Thus, the number of extra cache nodes needed to satisfy U is at least $|U|/5$, since a variable can appear in at most 5 clauses. We have

$$OPT(I) \geq n + |U|/5 \geq n + \epsilon m/5 \geq (1 + \epsilon/15)n.$$

The last inequality follows from the trivial bound $m \geq n/3$. The theorem follows by setting $\epsilon_1 = \epsilon/15$. □

Definition 3 *A PTAS (Polynomial Time Approximation Scheme) for a minimization problem* Π *is an algorithm that, for each* $\epsilon > 0$ *and instance* $I \in \Pi$*, returns a solution* $A(I)$*, such that* $A(I) \leq (1+\epsilon)OPT(I)$*, and* A *has running time polynomial in the size of* I*, depending on* ϵ *(see, e.g. Papadimitriou and Steiglitz (1982), page 425).*

Corollary 9 *Unless* $\mathcal{P} = \mathcal{NP}$*, the TSCPP cannot be approximated by* $(1+\epsilon_2)$ *for any* $\epsilon_2 \leq \epsilon_1$*, where* ϵ_1 *is given in Theorem 8, and therefore there is no polynomial time approximation scheme (PTAS) for the TSCPP.*

Proof: Given an instance ϕ of SAT, we can use the transformation given in Theorem 7, coupled with the transformation given in the proof of Theorem 2, to give a new polynomial transformation τ from SAT to TSCPP. Now, let I be the instance created by τ on input ϕ. Suppose there is an ϵ_2 approximation algorithm A for TSCPP, with $0 \leq \epsilon_2 \leq \epsilon_1$. Then, when A runs on an instance I constructed by τ from a satisfiable formula ϕ, the result must have cost $A(I) \leq (1+\epsilon_2)n < (1+\epsilon_1)n$. Otherwise, if ϕ is not satisfiable, then the result given by this algorithm must be greater than $(1+\epsilon_1)n$, because of the gap introduced by τ. Thus, if there is an ϵ_2-approximation algorithm, then we can decide in polynomial time if a formula ϕ is satisfiable or not. Assuming $\mathcal{P} \neq \mathcal{NP}$, there is no such algorithm.

The fact that there is no PTAS for the TSCPP is a consequence of this result and the definition of PTAS. □

The above theorem and corollary can be easily extended to the FSCPP. The fact that the same transformation can be used for both problems can be used to demonstrate the non-approximability result to the FSCPP as well. We state this as a corollary.

Corollary 10 *Unless $\mathcal{P} = \mathcal{NP}$, the FSCPP has no PTAS.*

Proof: The transformation from SAT to FSCPP is identical, so Theorem 8 is also valid for the FSCPP. This implies that the FSCPP has no PTAS, unless $\mathcal{P} = \mathcal{NP}$. □

4.3 Improved Hardness Result for FSCPP

In this section, we are interested in the case of general flows and directed arcs. This version of the problem is called the flow streaming cache placement problem (FSCPP). In particular, given k destinations, we show that the FSCPP cannot have a $\mathcal{O}(\log\log k - \delta)$-approximation algorithm, for a very small δ, unless $\mathcal{NP}$can be solved in sub-exponential time.

We have shown above that, given a instance of the FSCPP, there is an $\epsilon > 0$ such that the FSCPP cannot be approximated by $1 + \epsilon$, thus demonstrating that FSCPP is MAX SNP-hard (Papadimitriou and Yannakakis, 1991) and discarding the possibility of a PTAS. We show a stronger result: there is no approximation algorithm that can give a performance guarantee better than $\log\log k$, where k is the number of destinations. The proof is based on a reduction from the Set Cover problem.

Set Cover: Given a ground set $T = t_1, \ldots, t_n$, with subsets $S_1, \ldots, S_m \subset T$, find the minimum cardinality set $C \subseteq \{1, \ldots, m\}$ such that $\bigcup_{i \in C} S_i = T$.

It is known (Feige, 1998) that Set Cover does not have approximation algorithms for any guarantee better than $\mathcal{O}(\log n)$. Thus, if we find a transformation from Set Cover to FSCPP that preserves approximation, we can

prove a similar result for FSCPP. We show how this transformation, which will be represented by $\phi : \text{SC} \rightarrow \text{FSCPP}$, can be done.

For each instance I_{SC} of set cover, we must find a corresponding instance I_{FSCPP} of the FSCPP. The instance I_{SC} is composed of sets T and $S_i, \ldots, S_m$ as shown above. The transformation consists of defining a capacitated graph G with a source and a set D of destinations. Let G be the graph composed of the following nodes:

$$V = \{s\} \cup \{w_1, \ldots, w_m\} \cup \{v_1, \ldots, v_n\} \cup \{s_1, \ldots, s_m\}.$$

Also, let the edges E of the graph G be

$$E = \{(w_j, v_i) \mid t_i \in S_j\} \cup \bigcup_{i=1}^{m} \{(s, w_i)\} \cup \bigcup_{i=1}^{m} \{(w_i, s_i)\}.$$

In the instance of FSCPP, the set of destination nodes D is given by

$$D = \{v_1, \ldots, v_n\} \cup \{s_1, \ldots, s_m\},$$

and s is the source node. Thus, there is an one to one correspondence between nodes w_i and sets S_i, for $1 \leq i \leq m$. There is also an one to one correspondence between nodes v_i and ground elements $t_i \in T$, for $1 \leq i \leq n$. There is a directed edge between the source and each node w_i, and between nodes w_i and nodes representing elements appearing in the set S_i. Nodes w_i are also linked to each s_i. Finally, each edge e has capacity $c_e = 1$. See an example of such reduction in Figure 4–1. The ground set in this example is $T = \{t_1, \ldots, t_6\}$, and the subsets are $S_1 = \{t_1, t_2, t_4, t_5\}$, $S_2 = \{t_1, t_2, t_4, t_6\}$, and $S_3 = \{t_2, t_4, t_6\}$.

Theorem 11 *The transformation described above is a polynomial time reduction from* Set Cover *to FSCPP.*

Proof: Let I_{SC} be the instance of Set Cover and I_{FSCPP} the corresponding instance of the FSCPP. It is clear that the transformation is polynomial, since

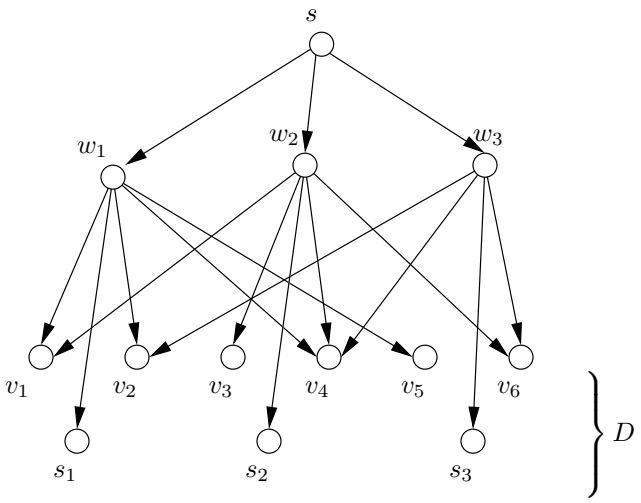

Figure 4–1: Example for transformation of Theorem 11.

the number of edges and nodes is given by a constant multiple of the number of elements and sets in the instance of Set Cover. We must prove that I_{IS} and I_{SCCP} have equivalent optimal solutions. Let S' be an optimal solution for I_{SC}. First we note that the destination nodes s_i, $1 \leq i \leq m$, can be reached only from nodes w_i, and therefore each s_i must be satisfied with flow coming from w_i. Thus, each node s_i saturates the corresponding w_i, which means that to satisfy any other node from w_i we must make it a cache node. Then, we can clearly make $R = \{w_i \mid i \in S'\}$, and serve all remaining destinations in $v_1, \ldots, v_n$, by definition of S'. Each node in R will be a cache node, and therefore R is a solution for I_{FSCPP}. This solution must be optimal, because otherwise we could use a smaller solution R' to construct a corresponding set $S'' \subset \{1, \ldots, m\}$ with $|S''| < |S'|$, covering all elements of T, and therefore contradicting the fact that S' is an optimum solution for the SC instance. Thus, the two instances I_{SC} and I_{FSCPP} have equivalent optimal solutions. □

Corollary 12 *Given an instance I of SC, and the transformation ϕ described above, then we have $OPT(I) = OPT(\phi(I))$.*

The following theorem, proved by Feige (Feige, 1998), will be useful for our main result.

Theorem 13 (Feige (Feige, 1998)) *If there is some $\epsilon > 0$ such that a polynomial time algorithm can approximate set cover within $(1-\epsilon)\log n$, then $\mathcal{NP} \subset TIME(n^{\mathcal{O}(\log\log n)})$.*

This theorem implies that finding approximate solutions with guarantee better than $(1-\epsilon)\log n$ for SET COVER is equivalent to solve any problem in $\mathcal{NP}$in sub-exponential time. It is strongly believed that this is not the case. We use this theorem and the reduction above to give a related bound for the approximation of FSCPP. To do this, we need a gap *preserving transformation* from SC to FSCPP, as stated in the following lemma.

Lemma 14 *If I is an instance* SET COVER*, then the transformation ϕ from SC to FSCPP described above is* gap preserving*, that is, it has the following property:*

(a) *If $OPT(I) = k$ then $OPT(\phi(I)) = k$; and*

(b) *If $OPT(I) \geq k \cdot \log n$ then $OPT(\phi(I)) \geq k \cdot \log\log|D| - \delta$,*

where k is a fixed value, depending on the instance, and

$$\delta = -k \log\left\{1 - \log(1 + \frac{n}{2^n})/\log|D|\right\} \approx 0$$

for large n.

Proof: Part (a) is a simple consequence of Corollary 12. Now, for part (b), note that the maximum number of sets in an instance of SC with n elements is 2^n. Consequently, in the instance of FSCPP created by transformation ϕ, $|D| = m + n \leq 2^n + n$. Thus, we have

$$\log|D| \leq \log(2^n + n) = n + \delta',$$

where $\delta' = \log(1 + \frac{n}{2^n})$. This implies that,

$$\log n \geq \log(\log |D| - \delta) = \log\log |D| + \delta'',$$

where $\delta'' = \log(1 - \frac{\delta'}{\log |D|})$. Therefore,

$$OPT(\phi(I)) \geq k \log n \geq k \log\log |D| - \delta,$$

where $\delta = -k\delta''$ (note that δ is a positive quantity). Finally, note that the quantity

$$-k \log \left\{ 1 - \log(1 + \frac{n}{2^n}) / \log |D| \right\}$$

goes very fast to zero, in comparison to n, thus the value $\log\log |D|$ is asymptotically optimal. □

The reduction shown in Theorem 11 is gap preserving, since it maintains an approximation gap, introduced by the instances of Set Cover. Note however that the name "gap preserving" is misleading in this case, since the new transformation has a smaller gap than then original.

Finally, we get the following result.

Theorem 15 *If there is some $\epsilon > 0$ such that a polynomial time algorithm A can approximate FSCPP within $(1 - \epsilon) \log\log k$, where $k = |D|$, then $\mathcal{NP} \subset TIME(n^{\mathcal{O}(\log\log n)})$.*

Proof: Suppose that an instance I of the SC is given. The transformation ϕ described above can be used to find an instance $\phi(I)$ of the FSCPP. Then, A can be used to solve the problem for instance $\phi(I)$. According to Lemma 14, transformation ϕ reduces any gap of $\log n$ to $\log\log k$. Thus, with such an algorithm one can differentiate between instances I with a gap of $\log n$. But this is not possible in polynomial time, according to (Feige, 1998, Theorem 10) unless $\mathcal{NP} \subset TIME(n^{\mathcal{O}(\log\log n)})$. □

4.4 Concluding Remarks

The SCPP is a difficult combinatorial optimization problem occurring in multicast networks. We have shown that the SCPP in general cannot have approximation algorithms with guarantee better than ϵ, for some $\epsilon > 1$. Thus, different from other optimization problems (such as the connected dominating set in Chapter 7), the SCPP cannot have a polynomial time approximation scheme (PTAS).

We have also proved that the FSCPP cannot be approximated by less then $\log \log k$, where k is the number of destinations, unless $\mathcal{NP}$can be solved in sub-exponential time. This shows that it is very difficult to find near optimal results for general instances of the FSCPP.

CHAPTER 5
ALGORITHMS FOR STREAMING CACHE PLACEMENT PROBLEMS

The results of the preceding chapter show that the SCPP is very difficult to solve, even if only approximate solutions are required. We describe some approximation algorithms that can be used to give solutions to the problem, and decrease the gap between known solutions and non-approximability results. We also consider practical heuristics to find good near-optimal solutions to the problem. We propose two general types of heuristics, based on complementary techniques, which can be used to give good starting solutions for the SCPP.

5.1 Introduction

In this chapter, we propose algorithms for solution of SCPP problems. Initially, we discuss algorithms with performance guarantee, also known as *approximation algorithms*. We give a general algorithm for SCPP problems, and also a better algorithm based on *flow techniques*. Approximation algorithms are very interesting as a way of understanding the complexity of the problem, but specially on this case, due to the negative results shown in Chapter 4, they are not very practical.

Thus, considering the complexity issues, we propose polynomial time construction algorithms for the SCPP, based on two general techniques: adding destinations to a partial solution, and reducing the number of infeasible nodes in an initial solution. We report the results of computational experiments based on these two algorithms and its variations.

This chapter is organized as follows. In Section 5.2 we present algorithms with performance guarantee for the SCPP. In Section 5.3, we turn to algorithms without performance guarantee, and discuss a number of possible construction strategies. Then, in Section 5.4 we proceed to an empirical evaluation of the solutions returned by the proposed construction heuristics. Final remarks and future research directions are discussed in Section 5.5.

5.2 Approximation Algorithms for SCPP

In this section, we present algorithms for the TSCPP and FSCPP and analyze their approximation guarantee. To simplify our results, we use the notation $A(I) = |R \cup \{s\}|$, where R is the set of cache nodes found by algorithm A applied to instance I. Also, $OPT(I) = |R^* \cup \{s\}|$, where R^* is an optimal set of cache nodes for instance I. Note that $A(I) \geq 1$ and $OPT(I) \geq 1$, which makes Definition 2 valid for our problems.

5.2.1 A Simple Algorithm for TSCPP

It is easy to construct a simple approximation algorithm for any instance of the TSCPP. We denote by $\delta_G(v)$ the degree of node v in the graph G.

Input: Graph G, destinations D, source s
Output: a set R of cache nodes
STEP*1:* Construct a spanning tree T of G
STEP*2:* Remove recursively all leaves of T which are not in $D \cup \{s\}$
STEP*3:* Let S_1 be the set of internal nodes v with $\delta_T(v) > 2$
STEP*4:* Let S_2 be the set of internal nodes v with $\delta_T(v) = 2$ and $v \in D$
STEP*5:* Return $R = S_1 \cup S_2$.

Algorithm 5: Spanning Tree Algorithm

Note that steps 3 and 4 of Algorithm 5 represent a worst case for Algorithm 4. The correctness of the algorithm is shown in the next lemma.

Lemma 16 *Algorithm 5 returns a feasible solution to the TSCPP.*

Proof: The operation in step 2 maintains feasibility, since leaves cannot be used to reach destinations. The result R includes all internal nodes v with $\delta_T(v) > 2$, and all internal nodes v with $\delta_T(v) = 2$ and $v \in D$. It suffices to prove that if $\delta_T(v) = 2$ and $v \notin D$ then v is not needed in R.

Suppose that v is an internal node with $\delta_T(v) = 2$ and $v \notin D$. If the number of destinations down the tree from v is equal to 1, then v does not need to be a cache. Now, assume that the number of cache nodes down the tree from v is two or more. Then, there are two cases. In the first case, there is a node w, between v and the destinations, with $\delta_T(w) > 2$. In this case, w is in R, and we need just to send one unit of flow from v to w, thus v does not need to be in R. In the second case, there must be some destination w with $\delta_T(w) = 2$ between v and the other destinations. Again, in this case w will be included in R from S_2. Thus v does not need to be in R. This shows that R is a feasible solution to the TSCPP. □

Lemma 17 *Algorithm 5 gives an approximation guarantee of* $|D|$.

Proof: Let us partition the set of destinations among D_1 and D_2, where $D_1 = D \setminus S_2$ and $D_2 = S_2$. Denote by D' the set of destinations which are leaves in T. Initially, note that for any tree the number of nodes v with degree $\delta(v) > 2$ is at most $|L| - 2$, where L is the set of nodes with $\delta(v) = 1$ (the leaves). But L in this case is $D' \cup \{s\} \subseteq D_1 \cup \{s\}$. This implies that $|S_1| \leq |D_1 \cup \{s\}| - 2$. Thus,

$$|R| = |S_1 \cup S_2| \leq |D_1 \cup \{s\}| - 2 + |D_2| = |D| - 1,$$

and

$$A(I) = |R| + 1 \leq |D| \leq |D| OPT(I),$$

since $OPT(I) \geq 1$. □

Let $\Delta = \Delta(G)$ be the maximum degree of G. In the case in which all capacities c_e, for $e \in E(G)$, are equal to one, we can give a better analysis of the previous algorithm with an improved performance.

Theorem 18 *When $c_e = 1$ for all $e \in E$, then Algorithm 1 is a k-approximation algorithm, where $k = \min\{\Delta(G), |D|\}$.*

Proof: The key idea to note is that if $c_e = 1$ for all $e \in E$, then $\lceil |D|/\Delta \rceil \leq OPT(I)$, for any instance I of the TSCPP. This happens because each cache node (as well as the source) can serve at most Δ destinations. Let $A(I)$ be the value returned by Algorithm 2 on instance I. We know from the previous analysis of Lemma 17 that $A(I) \leq |D|$. Thus $A(I) \leq \Delta OPT(I)$. The theorem follows, since we know that this is also an $|D|$-approximation algorithm. □

5.2.2 A Flow-based Algorithm for FSCPP

In this section, we present an approximation algorithm for the FSCPP. The algorithm is based on the idea of sending flow from the source to destination nodes. We show that this algorithm performs at least as good as the previous algorithm for the TSCPP. In addition, we show that for a special class of graphs this algorithm gives essentially the optimum solution. Therefore, for this class of graphs the FSCPP is solvable in polynomial time.

We give now standard definitions about network flows. For more details about the subject, see Ahuja et al. (1993). Let $f(x, y) \in R^+$ be the amount of flow sent on edge (x, y), for $(x, y) \in E$. A flow is called a *feasible flow* if it satisfies flow conservation constraints (3.2).

Let $F(f, s, t) = \sum_{v \in V} f(s, v)$ be the total flow sent from node s. We assume that s can send at most $\sum_{(s,v)\in E} c_{sv}$ units of flow, and t can receive at most $\sum_{(u,t)\in E} c_{ut}$ units of flow. A feasible flow f is the *maximum flow* from s to t if there is no feasible flow f' such that $F(f', s, t) > F(f, s, t)$. A node v is a

reached node from s by flow f if $\sum_{w \in V} f(w, v) > 0$. It is well known that when f is the maximum flow, then $F(f, s, t) = C(s, t)$, where $C(s, t)$ represents the minimum capacity of any set of edges separating s from t in G (the *minimum cut*). We also use the notation $C(U, \overline{U})$ to denote the total capacity of edges linking nodes in U to nodes in $\overline{U}$, where $U \subset V$ and $\overline{U} = V \setminus U$.

Denote any feasible flow starting from node v by f_v. The algorithm works by finding the maximum flow from s to all nodes in D. If the total cost of this maximum flow is $F(f_s, s, D) \geq |D|$, then the problem is solved, since all destinations can be reached without cache nodes. Otherwise, we put all nodes reachable from s in a set Q. Then we repeat the following steps until $D \setminus Q$ is empty: for all nodes $v \in Q$, compute the maximum flow f_v from v to D. Find the node v^* such that f_{v^*} is maximum. Then add v^* to R and add to Q all nodes reachable from v^*. Also, reduce the capacity of the edges in E by the amount of flow used by f_{v^*}. These steps are described more formally in Algorithm 6.

$Q \leftarrow \{s\}$
while $D \setminus Q \neq \emptyset$ **do**
  forall $v \in Q$ **do**
    find the maximum flow f_v from v to $D \setminus Q$
  end
  Let v^* be the node such that $F(f_{v^*}, s, D)$ is maximum
  $R \leftarrow R \cup \{v^*\}$
  Add to Q the nodes reached by f_{v^*}
  for *each edge* $(u, v) \in E$ **do**
    Reduce capacity $c_{u,v}$ by $f_{v^*}(u, v)$
  end
end

Algorithm 6: Flow algorithm

In the following theorem, we show that the running time of this algorithm is polynomial and depends on the time needed to find the maximum flow.

Denote by $C^M(G)$ the maximum value of the minimum cut between any pair of nodes $v, w \in V(G)$, i.e.

$$C^M(G) = \max_{v,w \in V} C(v, w).$$

Similarly, we define

$$C^m(G) = \min_{v,w \in V} C(v, w).$$

Theorem 19 *Algorithm 6 has running time equal to* $\mathcal{O}(n \cdot |D| \cdot T^{mf}/C^m(G))$, *where* T^{mf} *is the time needed to run the maximum flow algorithm.*

Proof: The most costly operations in this algorithm are calls to the maximum flow algorithm. Therefore we count the number of calls. Note that, at each of the N iterations of the while loop, a new element is added to the set of cache nodes. Thus, $A(I)$ is equal to the number of such iterations.

Let v_i be the node added to the set of cache nodes at iteration i, and Q^i be the content of set Q at iteration i of Algorithm 6. At each step, the number of elements of D found by the algorithm is, according to the maximum-flow/minimum-cut theorem, equal to the minimum cut from v_i to the remaining nodes in $D \setminus Q^i$ (recall that all demands are unitary). Then,

$$|D| = \sum_{i=1}^{N} C(v_i, D \setminus Q^i) \geq \sum_{i=1}^{N} \min_{w \in D} C(v_i, w) \geq A(I) \min_{v,w \in V} C(v, w).$$

Thus we have

$$N = A(I) \leq \frac{|D|}{C^m(G).} \tag{5.1}$$

At each iteration of the while loop, the number of calls to the maximum flow algorithm is at most n. The total number n_c of such calls is given by $n_c \leq n|D|/C^m(G)$. Thus the running time of Algorithm 6 is $\mathcal{O}(n|D|T^{mf}/C^m(G))$. □

Based on the performance analysis just shown, the following theorem gives an approximation guarantee for Algorithm 6.

Theorem 20 *Algorithm 6 is a k-approximation algorithm, where* $k = C^M(G)/C^m(G)$.

Proof: If we denote by R the set of cache nodes in the optimal solution, we have

$$|D| \leq \sum_{v_i \in R \cup \{s\}} C(v_i, D) \leq \sum_{i \in R \cup \{s\}} \max_{v,w \in V} C(v, w) = OPT(I) C^M(G) \qquad (5.2)$$

Combining inequalities (5.1) and (5.2) results in

$$A(I) \leq \frac{C^M(G)}{C^m(G)} OPT(I) \qquad \square$$

Note that the quantity $C^M(G)/C^m(G)$ can become large. However, for some types of graphs, the preceding algorithm gives us a better understanding of the problem. For example, if the graph has maximum degree $\Delta(G)$ and fixed capacity l, then it is easy to see that

$$\frac{C^M(G)}{C^m(G)} \leq \frac{\Delta(G) \cdot l}{l} = \Delta(G).$$

If the edge capacity is not fixed, then $C^M(G)/C^m(G)$ becomes at most $\Delta(G) \cdot c^M/c^m$, where c^M represents the maximum capacity and c^m the smallest capacity of edges in G.

5.3 Construction Algorithms for the SCPP

We this section, we provide construction algorithms for SCPP that give good results for a class of problem instances. The algorithms are based on dual methods for constructing solutions. In the first heuristic, the method used consists of sequentially adding destinations to the current flow, until all destinations are satisfied. The second method uses the idea of turning an

initial infeasible solution into a feasible one, by adding cache nodes to the existing infeasible flow.

The general method used can be summarized by saying that it consists of the selection, at each step, of subsets of the resulting solution, while a complete solution is not found. To select elements of the solution, an ordering function is employed. This is done in a way such that parts of the solution which seem promising in terms of objective function are added first. In the remaining of this section we describe the specific techniques proposed to create solutions for the SCPP.

5.3.1 Connecting Destinations

The first method we propose to construct solutions for the SCPP is based on adding destinations sequentially. The algorithm uses the fact that each feasible solution for the SCPP can be clearly described as the union of paths from the source s to the set D of destinations.

More formally, let $D = \{d_1, \ldots, d_k\}$ be the set of destinations. Then, a *feasible flow* for the SCPP is the union of a set of paths $s = P_1 \cup \ldots \cup P_k$, such that $P_1 = (s, x_1 \ldots, x_{j_1-1}, x_{j_1})$, $P_2 = (s, x_1 \ldots, x_{j_2-1}, x_{j_2})$, $\ldots$ $P_2 = (s, x_1 \ldots, x_{j_k-1}, x_{j_k})$, and where $x_{j_i} = d_i$, for $i \in \{1, \ldots, k\}$.

In the proposed algorithm, we try to construct a solution that is the union of paths. It is assumed initially that no destination has been connected to the source, and A is the set of non connected destinations, i.e., $A = D$. Also, the set of cache nodes R is initially empty. During the algorithm execution, S represents the current subgraph of G connected to the source s. At each step of the algorithm, a path is created linking S to one of the destinations $d \in A$. First, the algorithm tries to find a path $\mathcal{P}$ from d to one of the nodes in $R \cup \{s\}$. If this is not possible (this is represented by saying that $\mathcal{P} = nil$),

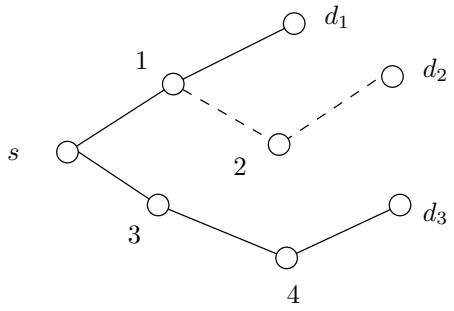

Figure 5–1: Sample execution for Algorithm 7. In this graph, all capacities are equal to 1. Destination d_2 is being added to the partial solution, and node 1 must be added to R.

then the algorithm tries to find a path $\mathcal{P}$ from d to some node in the connected subgraph S. Let w be the connection point between $\mathcal{P}$ and S. Then, add w to the set R of caches, since this is necessary to make to the solution feasible. Finally, the residual flow available in the graph is updated, to account for the capacity used by $\mathcal{P}$. The algorithm finishes when all destinations are included, and R is returned as the solution for the problem.

The formal steps of the proposed procedure are described in Algorithm 7. A number of important decisions, however, are unspecified in the description of the algorithm given so far. For example, there are many possible methods that can be used to select the next destination added to the current partial solution. Also, a path to a destination v can be found using diverse algorithms, which can result in different selections for a required cache node. These possible variations in the algorithms are represented by two functions, `get_path`(v, S), and `get_dest`(A). Thus, changing the definition of these functions we can actually achieve different implementations.

The first feature that can be changed, by defining a function `get_dest` : $2^V \rightarrow V$, is the order in which destinations are added to the final solution. Among the possible variations, we can list the following, which seem more useful:

Input: graph G, set D of destinations
$A \leftarrow D$
$S \leftarrow \emptyset$ /* current flow */
$R \leftarrow \emptyset$ /* set of cache nodes */
while $A \neq \emptyset$ **do**
    $v \leftarrow$ `get_dest`(A) /* choose a destination */
    $A \leftarrow A \setminus \{v\}$
    $\mathcal{P} \leftarrow$ `get_path`(v, R, G)
    if $\mathcal{P} = nil$ **then**
        $\mathcal{P} \leftarrow$ `get_path`(v, S, G)
        Let w be the node connecting $\mathcal{P}$ to S
        $R \leftarrow R \cup \{w\}$
    end
    Remove from G the capacity used in $\mathcal{P}$
    $S \leftarrow S \cup \mathcal{P}$
end
return R

Algorithm 7: First construction algorithm for the SCPP.

- give precedence to destinations closer to the source;
- give precedence to destinations further from the source;
- give precedence to destinations in lexicographic order.

The second basic decision that can be made is, once a destination v is selected, what type of path that will be used to join v to the rest of the graph. This decision is incorporated by the function `get_path` : $V \rightarrow 2^V$. A second parameter involved in the definition of `get_path` is the specific node $w \in S$ which will be connected to v. Note that such a node always exist, since at each step of the construction there is a path from destinations to at least one node already reached by flow. Using a greedy strategy, the best way to link a new destination d is through a path from d to some node $v \in R \cup \{s\}$. However, it may not be possible to find such v, and this requires the addition of a new node to R. In both situations it is not clear what is the optimal node to be linked to the current destination. Thus, another important decision in

Algorithm 7 concerns how to choose, at each step, the node to be linked to the current destination.

Shortest Path Policy. Perhaps, the simplest and most logical solution to the above questions is to link destination nodes using shortest paths. This policy is useful, since it can be applied to answer the two questions raised above: the path is created as a shortest path, and the node $v \in R \cup \{s\}$ selected is the closest from d. Thus, function `get_path`(v, R, G) in Algorithm 7 becomes: select the node $v \in R \cup \{s\}$ such that $dist(v, d)$ (the shortest path distance between v and d) is minimum; find the the shortest path $d \rightsquigarrow v$ from d to v; add $d \rightsquigarrow v$ to the current solution. If there is no path from d to $R \cup \{s\}$, then let v be the node closest to d and reached by flow from s and add v to R.

Other Policies. We have tested other two methods of connecting sources to destinations. In the first method, destinations are connected through a path found using the *depth first search* algorithm. In this implementation, paths are followed until a node already in R, or connected to some node in R, is found. In the last case, the connection node must be added to R. The second method used employs *random paths* starting from the destination nodes. This method is just useful to understand how good are the previous algorithms compared to a random solution.

Once defined the policy to be used in the constructor, it is not difficult to prove the correctness, as well as finding the complexity of the algorithm.

Theorem 21 *Given an instance I of the SCPP, Algorithm 7 returns a correct solution.*

Proof: At each step, a new destination is linked to the source node. Thus, at the end of the algorithm all destinations are connected. The paths determined by such connections are valid, since they use some of the available capacity

(according to the information stored in the residual graph G). Nodes are added to R only when it is required to make the connection possible. Thus, the resulting set R is correct, and corresponds to a valid flow from s to the set D of destinations. □

Theorem 22 *Using the closest node policy for destination selection and the shortest path policy for path creation, the time complexity of the Algorithm 7 is* $\mathcal{O}(|D|n^2)$.

Proof: The external loop is executed $|D|$ times. The steps of highest complexity inside the `while` loop are exactly the ones corresponding to the `get_path` procedure. As we are proposing to use the shortest path algorithm for the implementation of `get_path`, the complexity is $\mathcal{O}(n^2)$ (but can be improved with more clever implementations for the shortest path algorithm). Other operations have smaller complexity, thus the total complexity for Algorithm 7 is $\mathcal{O}(|D|n^2)$. □

Other implementations of Algorithm 7 would result in a very similar analysis of complexity.

5.3.2 Adding Caches to a Solution

We propose a second general technique for creating feasible solutions for the SCPP. The algorithm consists of adding caches sequentially to an initial infeasible solution, until it becomes feasible. The steps are presented in Algorithm 8. At the beginning of the algorithm, the set of cache nodes R is empty, and a possibly infeasible subgraph, linking the source to all destinations gives the initial flow. Such an initial infeasible solution can be created easily with any spanning tree algorithm. In the description of our procedure, we define the set I of infeasible nodes to be the nodes $v \in V \setminus \{s\}$ such that

$$\sum_{(w,v)\in E(G)} c(w,v) - \sum_{(v,w)\in E(G)} c(v,w) \neq b_v,$$

input: Graph G
$R \leftarrow \emptyset$
$S \leftarrow$ `spanning_tree`(G)
Remove from G the capacity on edges used by S
$I \leftarrow$ infeasible nodes in S
while $I \neq \emptyset$ *(there are infeasible nodes)* **do**
    $v \leftarrow$ `select_unfeasible_node`
    Try to find different paths to satisfy v
    if *a set* $\mathcal{P}$ *of paths is found* **then**
        Remove from G the capacity used by $\mathcal{P}$
        $S \leftarrow S \cup \mathcal{P}$
        $I \leftarrow I \setminus \{v\}$
    else
        $R \leftarrow R \cup \{v\}$
    end
end
return R

Algorithm 8: Second construction algorithm for the SCPP.

where b_v is the demand of v, which can be 0 or 1.

In the `while` loop of Algorithm 8, the current solution is initially checked for feasibility. This verification determine if there is any node $v \in V$ such that the amount of flow leaving the node is greater then the arriving flow, or in other words, $I \neq \emptyset$. The formal description of this verification procedure is given in Algorithm 9. If the solution is found to be infeasible, then it is necessary to improve its feasibility by increasing the number of properly balanced nodes.

The correction of infeasible nodes $v \in I$ is done in the body of the `while` loop in Algorithm 8. The procedure consists of selecting a node v from the set of infeasible nodes I in the flow graph, and trying to make if feasible by sending more data from one of the nodes $w \in R \cup \{s\}$. If this can done in such a way that v becomes feasible again, then the algorithm just needs to update the current subgraph S and the set of infeasible nodes I.

Input: current solution S, destinations set D
for $v \in D$ **do** $b_v \leftarrow 1$
$I \leftarrow \emptyset$
for $v \in V$ **do**
  $\delta \leftarrow \sum_{(w,v)\in E(S)} c(w,v) - \sum_{(v,w)\in E(S)} c(v,w)$
  if $\delta \neq b_v$ **then**
    $I \leftarrow I \cup \{v\}$
  end
end
return I /* returns the infeasible nodes */

Algorithm 9: Feasibility test for candidate solution.

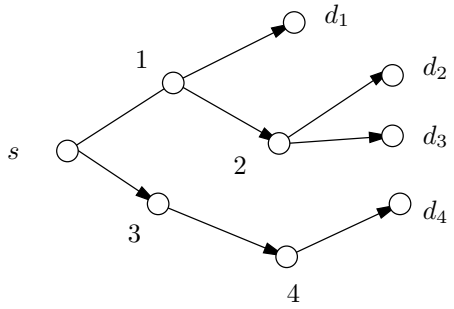

Figure 5–2: Sample execution for Algorithm 8, on a graph with unitary capacities. Nodes 1 and 2 are infeasible, and therefore are candidates to be included in R.

However, if v cannot receive enough additional data, then it must be added to the list of cache nodes. This clearly will make the node feasible again, since there will be no restrictions on the amount of flow departing from v. After adding v to R, the graph is modified as necessary. Example of possible modifications are changing the flow required by v to one unit and deleting additional paths leading to v, since only one is necessary to satisfy the flow requirements. We assume that these changes are done randomly, if necessary.

This construction technique can be seen as a dual of the algorithm presented in the previous subsection. In Algorithm 7 the assumption is that a

partial solution can be incomplete, but always feasible with relation to the flow sent from s to the reached destinations. On the other hand, in Algorithm 8 a solution is always complete, in the sense that all destinations are reached. However, it does not represent a feasible flow, until the end of the algorithm.

Procedure `select_unfeasible_node` has the objective of finding the most suitable node to be processed in the current iteration. This is the main decision in the implementation of Algorithm 8, and can be done using a greedy function, which will find the best candidate according to some criterion. We propose some possible candidate functions, and determine empirically (in the next section) how these functions perform in practice.

- *Function* `largest_infeasibility`*:* select the node that has greatest infeasibility, i.e., the difference between entering and leaving flow minus demand is maximum (breaking ties arbitrarily). This strategy tries to add to the set R a node which can benefit most from being a cache.
- *Function* `closest_from_source`*:* select the infeasible node which is closer to the destination. The advantage in this case is that a node v selected by this rule can help to reduce the infeasibility of other nodes down the in the path from s to the destinations.
- *Function* `uniform_random`*:* select uniformly a node $v \in I$ to be added to R. This rule is useful for breaking the biases existing in the previous methods. It has also the advantage of being very simple to compute, and therefore very fast.

Theorem 23 *Given an instance of the SCPP, Algorithm 8 returns a correct solution.*

Proof: In the algorithm, the set of infeasible nodes I will decrease monotonically. This happens because at each step one infeasible node is selected

and turned into a feasible node. Also, feasible nodes cannot become infeasible, since each operation requires that there is enough capacity in the network (this is guaranteed by the used of the residual graph G). Thus, the algorithm terminates.

The data flow from the source to the destinations must be valid at the end, by definition of the set I (which must be empty at the end). Similarly, the set R must be valid, since it is used only to make nodes feasible in the case that no additional paths can be found to satisfy their requirements. Thus, the solution return by Algorithm 8 is correct for the SCPP. □

Theorem 24 *The time complexity of the Algorithm 8 is $\mathcal{O}(nmK)$, where K is the sum of capacities in the SCPP instance.*

Proof: A spanning tree can be found in $\mathcal{O}(m \log n)$. Then, it follows the `while` loop that will perform at most n iterations. The procedure `select_unfeasible_node` can be implemented in $\mathcal{O}(n)$ by the use of some preprocessing, in each of the proposed implementations. Finding paths to infeasible nodes is clearly the most difficult operation in the loop. This can be performed in $\mathcal{O}(m + n)$ for each path, using a procedure such as depth first search. However, it may be necessary to run this step a number of times proportional to the sum of capacities in the graph (K), which results in $\mathcal{O}(mK)$. Other operations have low complexity, thus the maximum complexity for algorithm 8 is $\mathcal{O}(nmK)$. □

5.4 Empirical Evaluation

In this section we present computational experiments carried out with the construction algorithms proposed above.

All algorithms were implemented using the C programming language (the code is available by request). The resulting program was executed in a PC,

Table 5–1: Computational results for different variations of Algorithm 7 and Algorithm 8.

Instance		Constructor 1					Constructor 2		
n	m	DFS	Shortest	Random			LI	CS	UR
50	500	9.9	2.9	18.8	18.3	18.4	3.0	2.8	3.2
50	800	12.9	4.9	29.3	29.6	28.4	5.0	4.8	5.2
60	500	35.1	8.8	56.8	56.4	55.5	9.6	9.2	9.9
60	800	44.7	11.7	77.6	76.9	76.8	12.5	11.7	12.8
70	500	78.9	17.0	111.3	111.2	110.4	18.9	18.1	19.6
70	800	102.2	20.3	139.3	140.1	139.7	22.5	21.4	23.3
80	500	147.6	27.8	180.7	182.2	181.3	32.0	31.1	33.4
80	800	186.1	32.4	218.3	218.8	218.9	36.9	36.2	38.9
90	500	239.8	42.1	268.4	267.8	268.5	49.5	49.9	52.0
90	800	285.4	47.5	312.6	313.4	313.3	56.5	57.4	59.8
100	500	344.1	60.3	368.0	369.0	369.1	71.3	74.2	75.8
100	800	398.6	67.7	420.1	421.6	422.3	80.3	84.3	85.4
110	500	466.4	84.2	482.6	484.8	485.8	100.3	106.1	106.2
110	800	525.5	92.5	541.7	545.1	543.7	111.9	119.5	118.4
120	500	599.7	115.1	611.3	614.9	613.3	137.6	146.9	144.9
120	800	668.7	125.5	676.9	680.6	679.2	151.9	164.0	159.4
130	500	748.8	157.4	751.4	755.6	754.9	180.3	194.6	189.0
130	800	824.7	169.5	823.3	827.7	827.8	199.3	215.7	208.5
140	500	910.6	213.2	906.2	909.8	908.9	233.4	252.4	243.9
140	800	995.9	228.2	984.8	990.2	989.3	254.6	276.3	265.5
150	500	1092.0	281.8	1074.4	1080.3	1080.6	292.0	319.6	303.6
150	800	1185.4	299.6	1162.6	1170.0	1169.1	315.8	347.0	353.6

with 312MB of memory and a 800MHz processor. The GNU gcc compiler (under the Linux operating system) was used, with the optimization flag `-O2` enabled.

Table 5–1 presents a summary of the results of experiments with the proposed algorithms. The first two columns give the size of instances used. The remaining columns give results returned by Algorithm 7 and Algorithm 8 under different policies. Each entry reported in this table represents the averaged results over 30 instances of the stated size. Each instance was created using a random generator for connected graphs, first described in (Gomes et al., 1998). This generator creates graphs with random distances and capacities, but guarantees that the resulting instance is connected. Destinations were also defined randomly, with the number of destinations being equal to 40% of the

size of the number of nodes. All instances assume that node 1 is the source node.

In columns 3 to 7 of Table 5–1 we show the comparison of results returned by different variations of Algorithm 7. The second and third columns give the solutions returned by the policies depth first search and shortest path, respectively. For both policies, it was not observed any significative change in behavior by using different orderings of the destinations. On the other hand, in the columns 5 to 7 we report the values returned by the random path policy using different ordering methods (closest, farthest from source, and lexicographic order, respectively). It seams that, given the weakness of the random path policy, the ordering method becomes a significant parameter in the determination of the solution quality.

Columns 8 to 10 of Table 5–1 present results for the execution of Algorithm 8. They correspond, respectively, to the largest infeasibility, closest from source, and uniform random policies. It is clear from the results that Algorithm 8 is very effective, in comparison with of Algorithm 7. Although there is a tendency of getting better results with the 'closest from source' policy, it seams that the output is relatively independent of the order of selection for infeasible nodes. Thus, it is probably better to use a simpler implementation, such as the 'uniform random' policy.

Values for running time of the proposed algorithms are compared in Table 5–2. In this table, all values are given in milliseconds. As expected by the computational complexity results, Algorithm 8 has shown to spend more time than Algorithm 7. We note that, although these values can be improved by careful implementation, both algorithms have demonstrated good performance in practice.

Table 5–2: Comparison of computational time for Algorithm 7 and Algorithm 8. All values are in milliseconds.

Instance		Constructor 1					Constructor 2		
n	m	DFS	Shortest	Random			LI	CS	UR
50	500	10	14	7	7	2	12	17	11
50	800	23	37	20	18	5	20	29	18
60	500	32	59	30	26	12	48	69	42
60	800	47	96	43	41	16	67	98	63
70	500	62	125	56	54	20	106	163	104
70	800	81	164	71	72	24	135	211	139
80	500	97	201	82	86	32	210	335	197
80	800	124	249	100	107	36	267	429	236
90	500	144	293	117	124	42	383	619	354
90	800	175	354	142	148	48	472	769	412
100	500	199	416	163	167	56	622	1025	617
100	800	231	493	190	194	66	761	1259	772
110	500	255	562	214	217	77	985	1627	924
110	800	289	648	246	247	87	1169	1946	938
120	500	318	741	276	274	100	1501	2486	1329
120	800	356	844	312	308	112	1760	2925	1612
130	500	385	940	342	337	128	2136	3555	1984
130	800	424	1061	380	375	144	2491	4153	2093
140	500	458	1154	413	409	163	2981	4980	2905
140	800	508	1300	459	455	183	3431	5729	3321
150	500	555	1417	505	500	204	4105	6817	3729
150	800	611	1591	561	556	226	4675	7758	4015

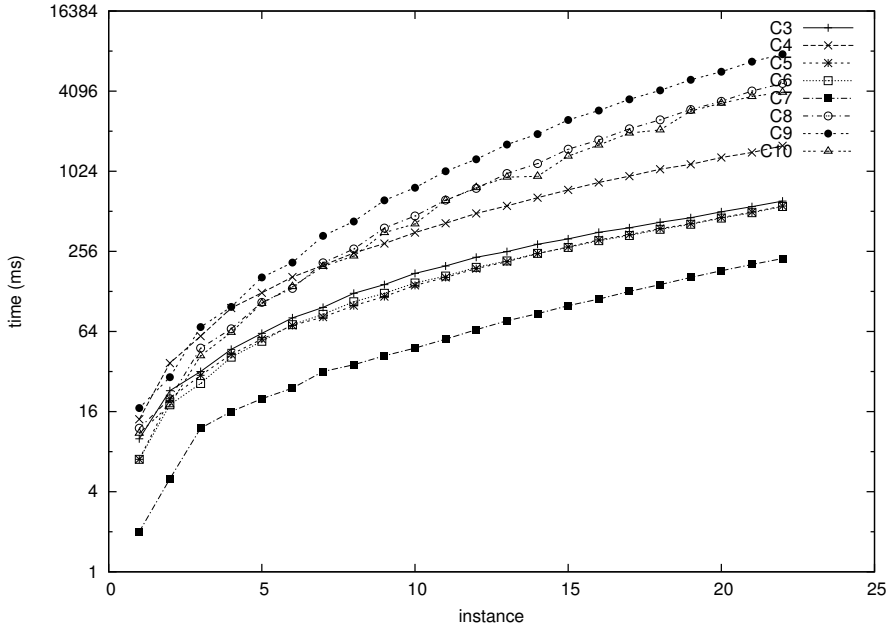

Figure 5–3: Comparison of computational time for different versions of Algorithm 7 and Algorithm 8. Labels 'C3' to 'C10' refer to the columns from 3 to 10 on Table 5–2.

5.5 Concluding Remarks

The SCPP is a difficult combinatorial optimization problem occurring in Multicast Networks. In this chapter, we described algorithms for solving the SCPP and some of its versions. Initially, we presented approximation algorithms for the SCPP. Although theoretically interesting, these algorithms cannot give good approximation guarantee in practice, due to the inherent problem complexity.

We also discussed how good solutions can be created by applying construction techniques. Two general types of techniques have being proposed, using dual points of view of the construction process. We described policies for implementation of these construction algorithms, and how different variants of the algorithms can be derived. Finally, we performed computational experiments to determine the quality of solutions generated by the different techniques described.

We believe that the techniques here exposed can be still further refined and improved. It would also be interesting to see these algorithms integrated with other heuristics. A future step is to use the proposed algorithms in the framework of general purpose metaheuristics, since this is the best way of achieving high quality solutions for most combinatorial problems. Clearly, other interesting open problems concern the proof of similar results for variants of the SCPP.

CHAPTER 6
HEURISTIC ALGORITHMS FOR ROUTING ON MULTICAST NETWORKS

An important problem on multicast networks asks for the determination of an optimum route to be followed by packages in a multicast group. This is known as the multicast routing problem (MRP). A large number of heuristic algorithms have been proposed in the last years to solve the MRP, which is of great interest for network engineers. In this chapter, a heuristic for the multicast routing problem is proposed, which improves over the well known algorithm of Kou et al. (1981). The resulting construction algorithm is used in the implementation of a metaheuristic approach for the MRP. In this approach, a restarting procedure similar to the greedy adaptive search (GRASP) applies the heuristic together with a local search method in order to find near optimal solutions to the MRP. Simulation results show that the proposed technique is superior in terms of solution quality, when compared to traditional heuristics. We also proposed novel improvements to GRASP that contribute to reduce the computational time and improve the overall quality of solutions.

6.1 Introduction

Multicast services are used in modern applications to allow direct communication between a source node and a set of destinations. In recent years, the number of applications of multicast has increased steadily, following the rapid advances of the Internet and intranet networks on the corporate world.

A number of algorithmic issues, however, remain as a major problem for the wide deployment of multicast applications. For example, routing is an

issue that has not been completely solved on multicast systems. While for traditional unicast systems the routing problem can be solved in polynomial time using, e.g., the Dijkstra's algorithm, the multicast routing problem is better modeled by the Steiner tree problem, which is known to be $\mathcal{NP}$-hard (Garey and Johnson, 1979).

Given the complexity of solving exactly the routing problem, a large number of heuristic algorithms have been proposed to find good, non-optimal solutions (Ballardie et al., 1993; Hong et al., 1998; Kompella et al., 1996; Salama et al., 1997b; Sriram et al., 1999; Zhu et al., 1995). Such methods, however, lack any guarantee of local optimality. This turns out to be an important shortcoming, specially for instances with a large number of group members, since the efficient use of resources is critical in this case.

6.1.1 The Multicast Routing Problem

A multicast network has the main objective of allowing communication from a source to a set of destinations with one single send operation. This is made possible by retransmitting data whenever two or more destinations can be reached from one single node. A set of nodes interested in the same piece of data is called a *multicast group*. The main task faced in the operation of a multicast network consists of finding routes for delivering data to all members of a multicast group.

The most common way of linking the source to destinations is through a tree spanning the involved nodes (source and destinations). The objectives and additional constraints of the problem may vary depending on the specific application and the respective performance requirements. For example, in real time video applications the quality of service (QoS) constraints require that the delay of transmission be less than some fixed threshold Δ. Thus, a basic

objective would be to minimize the total cost of the tree, subject to the constraint that all source-destination pairs have delay less than Δ. This problem is known as the *delay constrained multicast routing problem* (DCMRP).

To give a formal description of the DCMRP let $G = (V, E)$ be a graph where V is the set of nodes and E the set of links between nodes. The source node is represented by s and the destinations are $D = \{d_1, \ldots, d_k\}$, such that $D \subset V$. There is a cost function $c : E \to Z_+$ representing the cost of the links and a delay function $\tau : E \to Z_+$, giving the time elapsed when traversing an edge $e \in E$. The problem asks for a set of edges $E' \subseteq E$ such that s is connected to every node $d \in D$ on $G' = (V, E')$, the maximum delay is bounded, i.e.,

$$\max_{d \in D} \sum_{e \in \mathcal{P}_d} \tau(e) \leq \Delta,$$

where $\mathcal{P}_d$ is the path in E' from s to d, and such that the total cost $\sum_{e \in E'} c(e)$ of E' is minimum.

The DCMRP is easily seen as a generalization of the Steiner problem on graphs. In the Steiner problem, one is given a graph $G = (V, E)$ together with a cost function $c : E \to Z_+$, and a set $R \subset V$ of required nodes. The nodes in $V \setminus R$ are called Steiner nodes. The objective is to find a tree T linking the nodes in D, passing through Steiner nodes if necessary, such that the cost $\sum_{e \in T} c(e)$ is minimum. The Steiner problem on graphs is known to be $\mathcal{NP}$-hard (Garey and Johnson, 1979). Thus, we have a similar result for the DCMRP. Our interest is therefore in finding efficient algorithms that return good, near optimal solutions for a large number of practical instances.

6.1.2 Contributions

In this chapter we are interested in solution methods for the delay constrained multicast routing problem. In particular, we propose a new method

for computing routing trees for multicast networks using a variation of the KMB heuristic (Kou et al., 1981). We also propose to use the resulting heuristic as a constructor in a restarting procedure. The resulting metaheuristic can be viewed as a modification of GRASP (*greedy randomized search procedure*). The strategy is employed to avoid suboptimal solutions produced by existing heuristics.

At the same time, we propose innovative techniques applied to the GRASP metaheuristic. We introduce a new method for computing the candidate list, and show that this method is more effective in terms of computational time. The new method, combined with improved search heuristics, can be used to yield fast implementations for the problem considered.

This chapter is organized as follows. In Section 6.2, a heuristic for the MRP is proposed. Then, in Section 6.3 a metaheuristic based on the described algorithm is proposed. Computational results are discussed in Section 6.4. Concluding remarks are given in Section 6.5.

6.2 An Algorithm for the MRP

We start by describing one of the most important algorithms for the Steiner tree problem. This algorithm was proposed by Kou et al. (1981), and is known in the literature as the KMB heuristic. The main advantage of this algorithm is the fact that it is simple to describe and implement, and yet it gives a performance guarantee of two. In fact, empirical studies show that in most cases the results given by the heuristic are better than this theoretical upper bound. The KMB heuristic is presented in Algorithm 2. For convenience, we reproduce it in Algorithm 10.

The main result about Algorithm 10 is summarized bellow.

Input: Graph $G = (V, E)$, source s, and set D of destinations
Construct a complete graph $K(R, E)$ where the set of nodes is R.
Let the distance $d(i, j)$, $i, j \in R$ be the shortest path from i to j in G.
Find a minimum spanning tree T of K.
Replace each edge (i, j) in T by the complete path from i to j in G.
Let the resulting graph be T'
Compute a minimum spanning tree $\hat{T}$ of T'.
repeat
    $r \leftarrow false$
    if *there is a leaf* $w \in \hat{T}$ *which is not in* R **then**
        Remove w from $\hat{T}$
        $r \leftarrow true$
    end
until not r

Algorithm 10: KMB heuristic for the Steiner tree problem.

Theorem 25 *Algorithm 10 is a* $2 - 1/p$*-approximation algorithm, where* p *is number of required nodes.*

Modifying the solution for MRP. The strategy used for solving the MRP consists of modifying the KMB algorithm in order to account for the additional requirements of the MRP. The formal description of the method is given in Algorithm 11.

At the beginning of the algorithm, the initial solution is created by the KMB heuristic. Therefore, the initial result is known to be feasible for the Steiner tree problem, but it can be violating some of the delay or reliability constraints of the problem. The main part of the algorithm try to satisfy these requirements.

The first part of the algorithm checks the solution and determines if it is feasible for the QoS requirements. If there is any path $\mathcal{P}$ in the solution such that the total delay is greater than Δ, additional steps of the algorithm must be used to "fix" the problem. This is done by running the *shortest*

Input: Graph $G = (V, E)$, source s, and set D of destinations
Run the KMB algorithm (Algorithm 10)
Let T be result of the previous step
if *there is* $\mathcal{P} \in T$ *such that delay* $d(\mathcal{P}) > \Delta$ **then**
    /* Perform some modifications to allow for routing constraints */
    Substitute $\mathcal{P}$ by the shortest path between first and last nodes of $\mathcal{P}$
    Remove redundant edges not required in the tree
end
/* Do something similar when capacity used is more than $\beta\%$ */
while *some edge in* T *(on path* $\mathcal{P}_i$*) uses more* $\beta\%$ *of capacity* **do**
    Create a restricted graph G_r, where:
    • $V(G_r) = V(G)$ and
    • every edge $e \in E(G_r)$ has capacity less than $\frac{\beta}{100}c(e) - c_u(e)$, where $c_u(e)$ is the capacity of $e \in E(G)$
    Find a shortest path $\mathcal{P}'$ in G_r
    Substitute $\mathcal{P}_i$ by the shortest path $\mathcal{P}'$
    Remove redundant edges not required in the tree
end

Algorithm 11: Proposed heuristic for the MRP.

path algorithm (using, e.g., the Dijkstra's technique) and substituting it in the solution. Note that the shortest path is computed using delays as costs, in order to minimize the total delay. We assume that, after running this algorithm, the resulting path $\mathcal{P}'$ has delay less than Δ. If this is not the case, then the instance is clearly not feasible, and the algorithm can terminate. The next step is to substitute the infeasible path $\mathcal{P}$ by the shortest path found as described. Redundant edges must now be removed whenever needed. This can be done in time $\mathcal{O}(m)$, where m is the number of edges in the graph.

The second part of the heuristic deals with the case when the reliability constrained is violated. If this happens, a similar technique is used to find a new path avoiding the violated edges. However, one needs to be more careful because of the increased possibility of finding infeasible solutions. We propose the creation of a *reduced graph*, where edges are selected only when they have enough available capacity. This graph is denoted by $G_r = (V, E')$, with E' composed of all $e \in E$ such that the capacity used by e is less than $\beta\%$. The algorithm now tries to find a new shortest path $\mathcal{P}'$ linking the extreme nodes of one of the violated paths $\mathcal{P}_i$ that link s to a destination. Note that the shortest path $\mathcal{P}'$ is found over the reduced graph G_r. Thus, the new path is guaranteed to be feasible for the MRP, due to the way it was constructed. This step is repeated while there is an infeasible edge in the solution. Again, note that we assume that the instance of MRP is feasible, and therefore there is at least one solution satisfying these requirements.

With this construction algorithm in hands, the next logical step is to use it in a more general framework of a metaheuristic. In the next section, a metaheuristic algorithm is proposed, based on the restarting procedure known as GRASP.

```
Read instance
Initialize data structures
while termination criterion not satisfied do
    s ←create solution
    Improve solution s
    if s is the best solution so far then
        Save s
    end
end
Return best solution
```

Algorithm 12: GRASP

6.3 Metaheuristic Description

Restarting search heuristics have been very successful in a number of problems. One of the most well studied restarting algorithms is the greedy randomized adaptive search procedure (GRASP). GRASP is a metaheuristic proposed by Feo and Resende (1995) aimed at finding near optimal solutions for combinatorial optimization problems. It is composed of a number of iterations, where a new solution is picked from the feasible set, using a construction algorithm, and subsequently improved, using some local search method. GRASP has been very successful in a number of applications such as QAP (Oliveira et al., 2003b), frequency assignment (Gomes et al., 2001), and satisfiability. The steps of GRASP are summarized in Algorithm 12.

The GRASP algorithm is known to be a multi-start method, where at each iteration a new solution is constructed, and subsequently improved. In our implementation, the construction algorithm proposed in the previous section is used for the construction phase. A local search algorithm is employed in the improvement phase.

The construction phase of GRASP is presented in more detail on Algorithm 13. It traditionally consists of creating the solution step-by-step. A

while *solution s is not complete* **do**
    Order the k existing candidate elements
    Select a random α, such that $0 < \alpha \leq k$
    Let RCL be the set of α best candidates
    Randomly select one of the elements in RCL
end

Algorithm 13: GRASP construction phase

each step, a set of candidate elements is selected, and called the restricted candidate list (RCL). An element of the RCL is selected, and added to the solution. Notice that in this case, the elements are the possible paths in a solution. Although this is the most common method of implementation in GRASP, we see in the next section that this scheme can be improved by the careful use of randomization techniques.

6.3.1 Improving the Construction Phase

In the traditional implementation of the GRASP construction algorithm, each iteration must construct a list of candidates (RCL), and select one of its elements randomly. The size of the RCL is given by a parameter α, which frequently is randomly determined.

We propose a new method for the GRASP construction phase that is on average equivalent to the existing method, but which is much more efficient in practice. The method uses the following observation

Observation 26 *Let $x_1, \ldots, x_n$ be an unordered sequence, and $y_1, \ldots, y_n$ the corresponding ordered sequence. Then, to find a random element amongst $y_1, \ldots, y_\alpha$, for $0 < \alpha \leq n$, is on average equivalent to select the best of α random elements of $x_1, \ldots, x_n$.*

Proof: Given the sequence $x_1, \ldots, x_n$, and selecting random elements of the sequence, the probability of selecting one the the elements in the RCL is

```
while solution s is not complete do
    Select a random α, such that 1/n ≤ α ≤ k
    k ← 1/α
    c ← ∞ /* if this is a minimization problem */
    for j = 1, ..., k − 1 do
        Let C_j be the list of candidates at this iteration
        Randomly select one element x in C_j
        c_j ← f(x) if c_j < c then
            c ← c_j
            y ← x
        end
        Insert element y in the solution s
    end
end
```

Algorithm 14: Improved construction for GRASP

.

$n\alpha/n = \alpha$. Consequently, if W is an indicator random variable defined as 1 whenever the element picked is in the RCL, then $E(W) = \alpha$. Therefore, after doing k independent trials, the average number of elements in the RCL is $k\alpha$, by the additivity of expectation. If we have $k = 1/\alpha$, then on average there is just one element of the RCL within the picked elements. Now, to know what of these elements is the one in the RCL, we just need to take the smallest one. □

The observation above gives a very efficient way of implementing the RCL test, which gives, on average, the same results. Start with the full set C of candidate elements. Then, at each step generate a value of α, and pick at random $k = 1/\alpha$ elements of C. From the picked elements, store only the one which is the best fit for the greedy function. This method is depicted in Algorithm 14.

A clear advantage in terms of computational complexity is achieved by the proposed construction method for GRASP. The best advantage is that,

while in the original technique the candidate elements must be sorted, this is not necessary in the proposed algorithm. Moreover, the complexity of traditional construction is dependent on the number of candidate elements. In our method, the complexity is constant for a fixed value of α. For example, if alpha is $n/2$, then we need just two iterations to find an element in the RCL, with high probability.

Theorem 27 *The complexity of selecting elements from the RCL in the modified construction algorithm is* $n \log n$.

Proof: A probabilistic analysis of the complexity of the algorithm will be used. The important step that must be accounted for is the selection of a random element from C. Initially, let us assume that at each iteration of the for loop, the size of the list is n. Knowing that at each step the value of α is chosen from a uniform distribution, we find that the average number N of elements selected in the for loop is

$$\begin{aligned} E(N) &= E(\sum_{j=1}^{n} k) = \sum_{j=1}^{n} E(k|\alpha) \\ &= \sum_{j=1}^{n} \int_{1/n}^{1} E(k|\alpha = t)\, dt \\ &= \sum_{j=1}^{n} \int_{1/n}^{1} 1/t\, dt \\ &= \sum_{j=1}^{n} (\log 1 - \log 1/n) \\ &= \sum_{j=1}^{n} \log n = n \log n. \end{aligned}$$

Now, the decrease in the size of the list will not change the complexity of the result, since

$$\sum_{j=1}^{n} \log(n-j) = n \log n + \log \pi + \log \frac{\operatorname{csc} n\pi}{(n-1)!}.$$

The second term is a constant, and the third goes to zero very fast, as n tends to infinity, because of the value $(n-1)!$ in the denominator. □

6.3.2 Improvement Phase

GRASP has the advantage of being easy to develop, since it is composed of relatively independent procedures (the constructor and local search phases). It is well suited for applications with existing heuristic algorithms, that can be combined with GRASP to find a better solution.

However, one of the weaknesses of GRASP is its incapacity of integrate good solutions found previously into the current search iteration. Since each iteration will create a completely different solution, there is no information added to the system when a good solution is found.

A method that has been used lately to overcome this problem is called *path relinking* (PR) (Aiex et al., 2003; Resende and Ribeiro, 2003). In PR, a subset of the best solutions found is kept in a separate memory, called the *elite set*. At each iteration, one of the solutions s will be selected, and a process of comparing the current solution with s will start. Each component of the solution will be changed to the corresponding value on s, and after this a local search will be initiated to check for local optimality.

The main idea of PR is that, using an element s of the elite set and a different starting solution, we can find alternative paths in the solution space leading to improvements in the objective function. At the end of the process, the current solution will have objective function at most equal to the objective function of s. Due to randomization, however, there is a probability that the algorithm can find a better solution.

We modify the general structure of the GRASP metaheuristic to accommodate Path Relinking. The resulting algorithm is presented in Algorithm 15. The main modifications made to GRASP are:

- We have to create and maintain a set of elite solutions. This is shown in lines 8 and 14 of Algorithm 15.
- When the elite set is complete, each GRASP iteration should run the Path Relinking routine. This is show in line 6 of Algorithm 15.

Algorithm 15: GRASP with Path Relinking for minimization

```
c* ← ∞
while stopping criterion not satisfied do
    p ← GreedyRandomized()
    p ← LocalSearch(p)
    if |E| = ELITE_SIZE then
        PathRelinking
    else
        Insert current solution into E
    end
    if c(p) < c* then
        p* ← p
        c* ← c(p)
    end
    UpdateEliteSet
end
return p*
```

The elite set $\mathcal{E}$ is maintained in memory as a vector of solutions. We describe now how elements are inserted and removed from $\mathcal{E}$. Define the difference $\texttt{diff}(s_1, s_2)$ between solutions s_1 and s_2 to be the number of different edges in these two solutions. In the initial phase of GRASP with Path Relinking, the elite set is empty. While we have less than `ELITE_SIZE` elements, the current solution is inserted into $\mathcal{E}$ if it has a difference of at least 3 to all other elements in $\mathcal{E}$. The second phase starts when $\mathcal{E}$ has `ELITE_SIZE` elements.

In the second phase, new solutions generated by GRASP are inserted into $\mathcal{E}$ according to the update criterion, which is presented in Algorithm 16. If the current solution s is better than all solutions in $\mathcal{E}$, then s is directly inserted. Otherwise, we require that s has a difference of at least 3 to all elements in $\mathcal{E}$. If this is true, and s is at least better than the worst solution in $\mathcal{E}$, then s is inserted. To maintain the size of the elite set constant, we remove the solution x which has smallest difference `diff`(x, s), for all $x \in \mathcal{E}$.

Algorithm 16: `UpdateEliteSet` procedure

Input: current solution s ;
/* Set w to the worst solution */ ;
$w \leftarrow x \in \mathcal{E}$ such that $f(x) \geq y$, $\forall y \in \mathcal{E}$;
/* Set b to the best solution */ ;
$b \leftarrow x \in \mathcal{E}$ such that $f(x) \leq y$, $\forall y \in \mathcal{E}$;
if $f(s) < f(b)$ *or* $(f(s) \geq f(w)$ *and* $\min_{x \in \mathcal{E}}$ `diff`$(s, x) >= 3)$ **then**
    find s_p: $s_p = \arg\min_{x \in \mathcal{E}}$ `diff`(s, x) ;
    $\mathcal{E} \leftarrow \mathcal{E} \setminus \{s_p\}$;
    $\mathcal{E} \leftarrow \mathcal{E} \cup s$;
end

Algorithm 17: `PathRelinking` procedure

Input: current solution s ;
$s' \leftarrow$ random$(\mathcal{E})$;
for $i \leftarrow 1$ *to* $m = |E|$ **do**
    if $e_i \in s'$ *and* $e_i \notin s$ **then**
        $\delta \rightarrow$ `evalij`(s, i, j) ;
        if $\delta > 0$ **then**
            `LocalSearch`(s) ;
            `SaveSolution`(s) ;
        end
    end
end

The Path Relinking procedure, shown in Algorithm 17, starts with the selection of a random solution $s' \in \mathcal{E}$. This solution is called the *guiding*

solution, because it is used throughout the procedure to choose the next change to be performed. In each iteration, if an edge $e \in E(G)$ is in s' but not in s, the algorithm will include e in s and check if this brings an improvement. The change in objective function caused by this substitution is found using the function `evalij`. If there is an improvement, then we apply local search in the resulting solution in order to maintain local optimality. The algorithm does not run local search for non-improving changes, in order to reduce the total computational effort. If the solution found at the current step is better than the best known solution, then it is saved.

The final result of Path Relinking depends on the objective function value of the solutions found during the algorithm. Let s^i, for $0 \leq i \leq n$, be the solution found at the i-th iteration of the Path Relinking procedure (we note that $s^0 = s$ and $s^n = s'$). If, given the initial solution s and the guiding solution s', we can find a solution s^* such that $f(s*) < \min\{f(s), f(s')\}$, and $f(s^*) \leq f(s^i)$, for $1 \leq i < n$, where $f(s)$ is the objective value of s, then s^* is clearly the best solution and is returned by the algorithm. However, if we cannot find an improvement, then we prefer to return some solution which is different from s and s', but still good in some sense.

We define a local optimum relative to Path Relinking to be a solution s^i, such that $s^i < s^{i-1}$ and $s^i < s^{i+1}$, for $1 < i < n$. If there is no such solution then we define s^0 and s^n as the only local optima. We define then the resulting solution as $s^* = \min\{s^i : s^i \text{ is a local optimum relative to Path Relinking}\}$. In this definition, if there is an improvement, then s^* is the best improvement found. Otherwise, if there is some local optimum, we choose the best local optimum. Note that we return the minimum of s, s' only if no other local optimum was found during the execution of Path Relinking. This is done in

order to improve diversity of GRASP, avoiding the unnecessary repetition of solutions already in the elite set.

The complexity of the Path Relinking procedure is determined by the operations performed to find the best improvement. The `evalij` procedure (line 7 of Algorithm 17) has complexity $\mathcal{O}(n)$, because it simply verifies the impact of the change in the solution. Thus, the step with biggest complexity is the local search procedure (line 13). A faster local search implementation was employed, with $\mathcal{O}(n^2)$ complexity. This implies a complexity of $\mathcal{O}(n^2)$ inside the `for` loop (lines 4 to 17). Therefore, we have the following result.

Theorem 28 *The worst case performance of Path Relinking as described in Algorithm 17 is* $\mathcal{O}(n^3)$.

6.3.3 Reverse Path Relinking and Post-processing

The Path Relinking procedure described above can be further generalized, by considering that it can be run not only from the current solution s to a solution s' in the elite set, but also in the reverse direction. The results obtained with this change are not necessarily equal to the previous results. We call this modification of the path relinking procedure *reverse path relinking*. Therefore, in our implementation, for each GRASP iteration the reverse path relinking is also applied, to improve the results of the intensification phase.

As a last step of GRASP, we use Path Relinking as a post-optimization procedure at the end of the program. We call this *final path relinking*. The post-optimization step ensures that the solution returned by GRASP is optimal with respect to the path relinking operation. A description for this step is given in Algorithm 18. At each iteration of the `repeat` loop in line 1, we run Path Relinking for each pair of solutions $s_i, s_j \in \mathcal{E}$. The algorithm stops

when, after doing this for all possible pairs in $\mathcal{E}$, the objective function of the best solution cannot be improved.

Algorithm 18: Final Path Relinking

```
repeat
    for i = 1 to n do
        for j ← 1 to n do
            if i ≠ j then
                e1 ← element i of E ;
                e2 ← element j of E ;
                Path-Relinking(e1, e2) ;
            end
        end
    end
until while best solution can be improved;
```

6.3.4 Efficient implementation of Path Relinking

One of the computational burdens associated with the Path Relinking method is the requirement of performing some kind of local search for new solutions found during its execution. This is done in order to maintain local optimality and further explore the new path in the search space. However the use of local search can make each iteration of GRASP slower and have a negative effect in the overall computational time.

To avoid this situation we improved the original local search used in GRASP by using a non-exhaustive improvement phase. In our implementation, only one of the edges not included in s (selected randomly) is verified and included. This reduces the complexity of local search by a factor of n, leading to a $\mathcal{O}(n^2)$ implementation. This scheme is used in each iteration inside Path Relinking.

To enhance the quality of local search outside Path Relinking we use the following additional method. The modified local search is executed as discussed above. Then, we change randomly two pair of elements in the solution

and return to local search as before. The result of this operation allows the algorithm to explore a different, but closely related neighborhood, which can bring further improvements to the current solution. We continue with the local method, until a terminating criterion is satisfied (we used number of iterations without improvement as the terminating condition).

These changes represented a good improvement over the original local search, both in terms of time as well as quality of solutions found, as discussed in the next section.

6.4 Computational Experiments

The heuristics proposed in this chapter have been implemented using the ANSI-C language. The implementation was devised to be highly portable across platforms and easy to reuse. The compiler employed was the GNU GCC optimized compiler. The level of optimization chosen was `-O2`. The processor used was a Pentium4 with 2.8GHz, and 512MB of available memory.

Table 6–1 presents a summary of the results found by the algorithm proposed in this chapter. The first three columns describe the instances used for testing. We employed random instances of the MRP ranging from 100 to 200 nodes and 500 to 2000 edges. For each size of instance, we made 50 runs with different instances of the same size. The objective of this methodology is to find results that represent the average behavior of instances.

The results presented in Table 6–1 show that the algorithm implemented as proposed above gives a large improvement over the simple KMB heuristic. We see that for most problems, the solution returned by our heuristic is about 30% better than the KMB results. This gives a clear indication that our results are much closer to the global optimum. In column 9, we decided to report just the time spent in the construction phase, since this is the only comparable part

Table 6–1: Summary of results for the proposed metaheuristic for the MRP. Column 9 (*) reports only the time spent in the construction phase.

Instance			**KMB**			**Metaheuristic**			
n	m	$\|D\|$	best	average	time	best	average	time*	improv.
100	500	10	77	93.2	1.5	59	63.3	1.8	32%
100	600	10	56	69.4	1.5	42	43.6	1.9	37%
100	700	10	66	74.2	1.2	43	46.8	1.3	37%
120	800	12	80	87.4	2.4	50	54.1	2.9	38%
120	900	12	73	91.3	2.1	60	64.3	2.5	30%
120	1000	12	56	63.6	2.2	41	43.1	3.5	32%
140	1000	14	82	98.2	3.1	61	65.4	3.6	33%
140	1100	14	65	81.1	3.8	56	57.5	3.7	29%
140	1200	14	65	74.9	3.3	47	50.6	3.9	32%
160	1300	16	108	121.6	4.8	78	80.7	4.9	34%
160	1400	16	97	112.3	4.8	70	77.7	5.6	31%
160	1500	16	89	97.1	4.9	55	64.2	5.8	34%
180	1600	18	101	115.3	6.4	73	80.3	6.9	30%
180	1700	18	113	122	6.5	75	79.4	7.2	35%
180	1800	18	99	110	6.5	69	73.4	8.4	33%
200	1800	20	119	133.6	8.7	87	91.4	9.3	32%
200	1900	20	114	126	8.7	79	84.6	9.7	33%
200	2000	20	112	122.5	8.6	77	83.0	10.2	32%

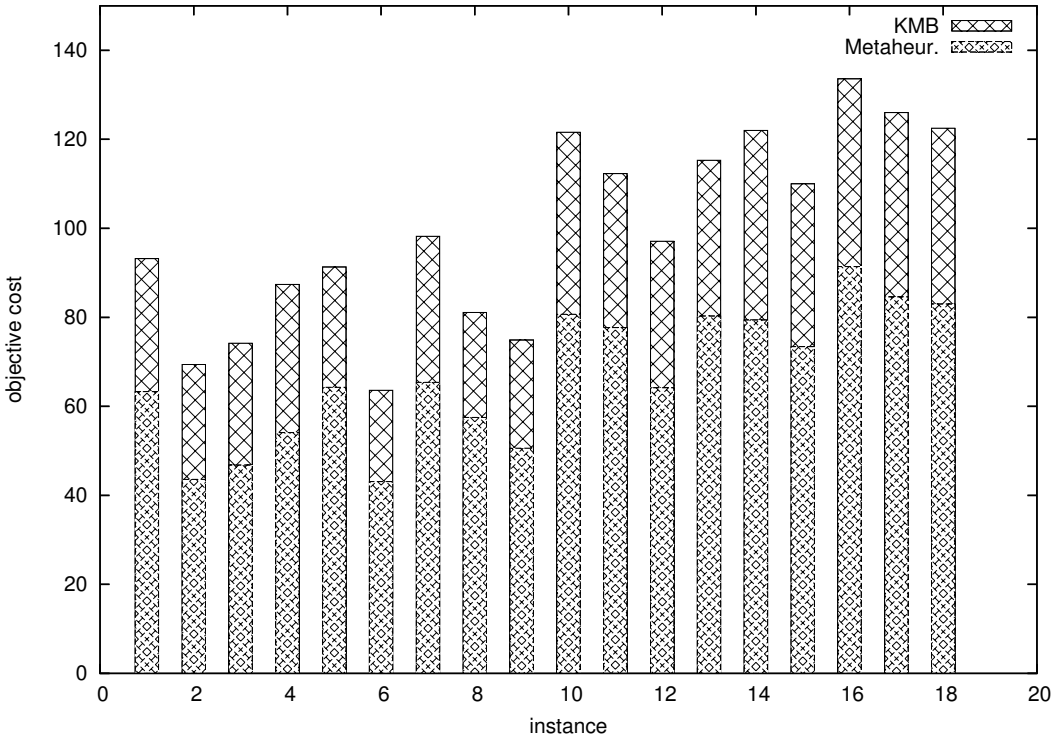

Figure 6–1: Comparison between the average solution costs found by the KMB heuristic and our algorithm.

of the two procedures, in terms of computational time. The metaheuristic was, in fact, run for a maximum of ten minutes, for each iteration. The same results appear also in Figure 6–1.

6.5 Concluding Remarks

In this chapter, we presented a new heuristic algorithm to solve the MRP. Its main technique is to use a modified construction method, based on the KMB heuristic (Kou et al., 1981) to find feasible solutions for the MRP. We combine this construction algorithm with a restarting procedure base on GRASP.

The resulting algorithm was found to yield near optimal solutions, with very good improvements, compared to the original KMB heuristic. A topic for further research would be to compare this solution to other existing methods.

Another interesting topic would be to study distributed versions of this procedure. This would not be very difficult, since it is well known that restarting procedures such as GRASP are simple to parallelize. However, this is important task, since distributed algorithms are essential for the practical implementation of network routing algorithms.

CHAPTER 7
A NEW HEURISTIC FOR THE MINIMUM CONNECTED DOMINATING SET PROBLEM ON AD HOC WIRELESS NETWORKS

Given a graph $G = (V, E)$, a dominating set D is a subset of V such that any vertex not in D is adjacent to at least one vertex in D. Efficient algorithms for computing the minimum connected dominating set (MCDS) are essential for solving many practical problems, such as finding a minimum size backbone in ad hoc networks. Wireless ad hoc networks appear in a wide variety of applications, including mobile commerce, search and discovery, and military battlefield. In this chapter we propose a new efficient heuristic algorithm for the minimum connected dominating set problem. The algorithm starts with a feasible solution containing all vertices of the graph. Then it reduces the size of the CDS by excluding some vertices using a greedy criterion. We also discuss a distributed version of this algorithm. The results of numerical testing show that, despite its simplicity, the proposed algorithm is competitive with other existing approaches.

7.1 Introduction

In many applications of wireless networks, such as mobile commerce, search and rescue, and military battlefield, one deals with communication systems having no fixed infrastructure, referred to as *ad hoc wireless networks*. An essential problem concerning ad hoc wireless networks is to design routing protocols allowing for communication between hosts. The dynamic nature of ad hoc networks makes this problem especially challenging. However, in some cases the problem of computing an acceptable virtual backbone can be reduced

to the well known minimum connected dominating set problem in unit-disk graphs (Butenko et al., 2002).

Given a simple undirected graph $G = (V, E)$ with the set of vertices V and the set of edges E, a *dominating set* (DS) is a set $D \subseteq V$ such that each vertex in $V \setminus D$ is adjacent to at least one vertex in D. If the graph is connected, a *connected dominating set* (CDS) is a DS which is also a connected subgraph of G. We note that computing the minimum CDS (MCDS) is equivalent to finding a spanning tree with the maximum number of leaves in G. In a *unit-disk* graph, two vertices are connected whenever the Euclidean distance between them is at most one unit.

Ad hoc networks can be modeled using unit-disk graphs as follows. The hosts in a wireless network are represented by vertices in the corresponding unit-disk graph, where the unit distance corresponds to the transmission range of a wireless device (see Figure 7–1). It is known that both CDS and MCDS problems are $\mathcal{NP}$-hard (Garey and Johnson, 1979). This remains the case even when they are restricted to planar, unit disk graphs (Baker, 1994).

Following the increased interest in wireless ad hoc networks, many approaches have been proposed for the MCDS problem in the recent years (Alzoubi et al., 2002; Butenko et al., 2002; Das and Bharghavan, 1997; Stojmenovic et al., 2001). Most of the heuristics are based on the idea of creating a dominating set incrementally, using some greedy technique. Some approaches try to construct a MCDS by finding a maximal independent set, which is then expanded to a CDS by adding "connecting" vertices (Butenko et al., 2002; Stojmenovic et al., 2001). An *independent set* (IS) in G is a set $I \subseteq V$ such that for each pair of vertices $u, v \in I$, $(u, v) \notin E$. An independent set I is *maximal IS* if any vertex not in I has a neighbor in I. Obviously, any maximal independent set is also a dominating set.

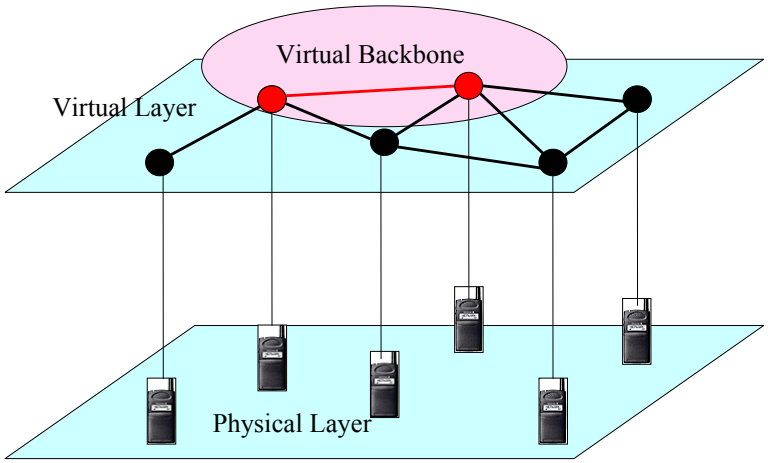

Figure 7–1: Approximating the virtual backbone with a connected dominating set in a unit-disk graph

There are several polynomial-time approximation algorithms for the MCDS problem. For instance, Guha and Khuller (1998) propose an algorithm with approximation factor of $H(\Delta)+2$, where Δ is the maximum degree of the graph and $H(n) = 1+1/2+\cdots+1/n$ is the harmonic function. Other approximation algorithms are given in Butenko et al. (2002); Marathe et al. (1995). A polynomial time approximation scheme (PTAS) for MCDS in unit-disk graphs is also possible, as shown by Hunt III et al. (1998) and more recently by Cheng et al. (2003).

A common feature of the currently available techniques for solving the MCDS problem is that the algorithms create the CDS from scratch, adding at each iteration some vertices according to a greedy criterion. For the general dominating set problem, the only exception known to the authors is briefly explained in Sanchis (2002), where a solution is created by sequentially removing vertices. A shared disadvantage of such algorithms is that they may require additional setup time, which is needed to construct a CDS from scratch. Another

weakness of the existing approaches is that frequently they use complicated strategies in order to achieve a good performance guarantee.

In this chapter, we propose a new heuristic algorithm for computing approximate solutions to the minimum connected dominating set problem. In particular, we discuss in detail the application of this algorithm to the MCDS problem in unit-disk graphs. The algorithm starts with a feasible solution, and recursively removes vertices from this solution, until a minimal CDS is found (here, by a *minimal* CDS we mean a connected dominating set, in which removing any vertex would result in a disconnected induced subgraph). Using this technique, the proposed algorithm maintains a feasible solution at any stage of its execution; therefore, there are no setup time requirements. The approach also has the advantage of being simple to implement, with experimental results comparable to the best existing algorithms.

This chapter uses standard graph-theoretical notations. Given a graph $G = (V, E)$, a subgraph of G induced by the set of vertices S is represented by $G[S]$. The set of adjacent vertices (also called neighbors) of $v \in V$ is denoted by $N(v)$. Also, we use $\delta(v)$ to denote the number of vertices adjacent to v, *i.e.* $\delta(v) = |N(v)|$.

The chapter is organized as follows. In Section 7.2 we present the algorithm and prove some results about its time complexity. In Section 7.3, we discuss a distributed implementation of this algorithm. Results of computational experiments with the proposed approach are presented in Section 7.4. Finally, in Section 7.5 we give some concluding remarks.

7.2 Algorithm for the MCDS Problem

In this section, we describe our algorithm for the minimum connected dominating set problem. As we already mentioned, most existing heuristics for

the MCDS problem work by selecting vertices to be a part of the dominating set and adding them to the final solution. We proceed using the inverse method: the algorithm starts with all vertices in the initial CDS. Then, at each step we select a vertex using a greedy method and either remove it from the current set or include it in the final solution. Algorithm 19 is a formal description of the proposed procedure.

```
/* D is the current CDS; F is the set of fixed vertices */
D ← V
F ← ∅
while D \ F ≠ ∅ do
    u ← argmin{δ(v) | v ∈ D \ F}
    if G[D \ {u}] is not connected then
        F ← F ∪ {u}
    else
        D ← D \ {u}
        forall s ∈ D ∩ N(u) do
            δ(s) ← δ(s) − 1
        end
        if N(u) ∩ F = ∅ then
            w ← argmax{δ(v) | v ∈ N(u)}
            F ← F ∪ w
        end
    end
end
Return D
```

Algorithm 19: Compute a CDS

At the initialization stage, we take the set V of all vertices as the starting CDS (recall that we deal only with connected graphs). In the algorithm, we consider two types of vertices. A *fixed vertex* is a vertex that cannot be removed from the CDS, since its removal would result in an infeasible solution. Fixing a vertex means that this vertex will be a part of the final dominating set constructed by the algorithm. A *non-fixed vertex* can be removed only if their removal does not disconnect the subgraph induced by the current solution. At

each step of the algorithm, at least one vertex is either fixed, or removed from the current feasible solution.

In Algorithm 19, D is the current CDS; F is the set of fixed vertices. In the beginning, $D = V$ and $F = \emptyset$. At each iteration of the **while** loop of Algorithm 19, we select select a non-fixed vertex u, which has the minimum degree in $G[D]$. If removing u makes the graph disconnected, then we clearly need u in the final solution, and thus u must be fixed. Otherwise, we remove u from the current CDS D and select some neighbor $v \in N(u)$ to be fixed, in the case that no neighbor of u has been fixed before. We select a vertex with the highest connectivity to be fixed, since we want to minimize the number of fixed vertices. These steps are repeated while there is a non-fixed vertex in D. In the following theorem we show that the algorithm outputs a CDS correctly.

Theorem 29 *Algorithm 19 returns a connected dominating set, and has the time complexity of* $\mathcal{O}(nm)$.

Proof: We show by induction on the number of iterations that the returned set D is a connected dominating set. This is certainly true at the beginning, since the graph is connected, and therefore $D = V$ is a CDS. At each step we remove the vertex with minimum degree, only if the removal does not disconnect D. The algorithm also makes sure that for each removed vertex u there is a neighbor $v \in N(u)$ which is fixed. Thus, for each vertex not in D, there will be at least one adjacent vertex included in the final set D. This implies that D is a CDS.

To determine the time complexity of Algorithm 19, note that the **while** loop is executed at most $n - 1$ times, since we either remove or fix at least one vertex at each iteration. At each step, the most expensive operation is to determine if removing a vertex disconnects the graph. To do this we need $\mathcal{O}(m + n)$ time, which corresponds to the time needed to run the depth first

search (or breadth first search) algorithm. Thus, the total time complexity of Algorithm 19 is given by $\mathcal{O}(nm)$. □

The proposed algorithm can be considered a convenient alternative to existing methods. Some of the advantages can be seen not only in computational complexity but in other terms as well. First of all, it is the simplicity of the method. Most algorithms for MCDS start by creating a (not necessarily connected) dominating set, and subsequently they must go through an extra step to ensure that the resulting set is connected. In the case of Algorithm 19, no extra step is needed, since connectedness is guaranteed at each iteration. Another favorable consideration is that the algorithm always maintains a feasible solution at any stage of its execution, thus providing a feasible virtual backbone at any time during the computation.

7.3 A Distributed Implementation

In this section, we discuss a distributed heuristic algorithm for the MCDS problem, based on Algorithm 19. For ad hoc wireless network applications, algorithms implemented in a non-centralized, distributed environment have great importance, since this is the way that the algorithm must run in practice. Thus, we propose a distributed algorithm that uses a strategy similar to Algorithm 1. We describe below how the steps of the algorithm are defined in terms of a sequence of messages.

In the description of the distributed algorithm, we say that a link (v, u) is an *active link* for vertex v, if u was not previously removed from the CDS. The messages in the algorithm are sent through active links only, since all other links lead to vertices which cannot be a part of the CDS. We assume, as usual, that there is a starting vertex, found by means of some *leader election algorithm* (Malpani et al., 2000). It is known (Alzoubi et al., 2002) that this

can be done in $\mathcal{O}(n \log n)$ time. We also assume that the leader vertex v_l is a vertex with the smallest number of neighbors. This feature is not difficult to add to the original leader election algorithm, so we will assume that this is the case.

The execution starts from the leader, which runs the `self-removal` procedure. First, we verify if removing this vertex would disconnect the subgraph induced by the resulting set of vertices. If this is the case, we run the `Fix-vertex` procedure, since then the current vertex must be present in the final solution. Otherwise, the `Remove-vertex` procedure is executed.

The `Fix-vertex` procedure will execute the steps required to fix the current vertex in the CDS. Initially it sends the message `NEWDOM` to announce that it is becoming a dominator. Then, the current vertex looks for other vertices to be considered for removal. This is done based on the degree of each neighbor; therefore the neighbor vertex with the smallest degree will be chosen first, and will receive a `TRY-DISCONNECT` message.

The `Remove-vertex` procedure is executed only when it is known that the current vertex v can be removed. The first step is to send the message `DISCONNECTED` to all neighbors, and then select the vertex which will be the dominator for v. If there is some dominator in the neighborhood, it is used. Otherwise, a new dominator is chosen to be the vertex with the highest connectivity in $N(v)$. Finally, the message `SET-DOMINATOR` is sent to the chosen vertex.

Computational Complexity. Note that in the presented algorithm, the step with highest complexity consists in verifying connectedness for the resulting network. To do this, we run a distributed algorithm which verify if the graph is still connected when the current vertex is removed. An example of such algorithm is the distributed breadth first search (BFS), which is known to

run in $\mathcal{O}(D \log^3 n)$, where D is the diameter (length of the maximum shortest path) of the network, and sends at most $\mathcal{O}(m + n \log^3 n)$ messages (Awerbuch and Peleg, 1990). Thus, each step of our distributed algorithm has the same time complexity.

To speed up the process, we can change the requirements of the algorithm by asking the resulting graph to be connected in k steps for some constant k, instead of being completely connected. To do this, the algorithm for connectedness can be modified by sending a message with TTL (time to live) equal to a constant k. This means that after k retransmissions, if the packet does not reach the destination, then it is simply discarded. The added restriction implies that we require connectedness in at most k hops for each vertex. We think that this is not a very restrictive constraint, since it is also desirable that paths between vertices are not very long. With this additional requirement, the diameter of the graph can be thought of as a constant, and therefore the resulting time complexity for each step becomes $\mathcal{O}(\log^3 n)$. The time complexity of the whole algorithm is $\mathcal{O}(n \log^3 n) = \tilde{\mathcal{O}}(n)$. We use the notation $\tilde{\mathcal{O}}(f(n))$ to represent $\mathcal{O}(f(n) \log^k n)$, for some constant k.

The number of messages sent while processing a vertex is also bounded from above by the number of messages used in the BFS algorithm. Thus, after running this in at most n vertices, we have an upper bound of $\mathcal{O}(nm+n^2 \log^3 n)$ (which is $\tilde{\mathcal{O}}(n^2)$ when the graph is sparse) for the total message complexity. These results are summarized in the following theorem.

Theorem 30 *The distributed algorithm with k-connectedness requirement runs in time $\mathcal{O}(n \log^3 n) = \tilde{\mathcal{O}}(n)$ and has message complexity equal to $\mathcal{O}(nm + n^2 \log^3 n)$. For sparse graphs, the message complexity is $\tilde{\mathcal{O}}(n^2)$.*

The details of the resulting algorithm are shown in Figure 7–2. We prove its correctness in the following theorem.

General actions:
  on `TRY-DISCONNECT`, do `Self-removal`
  on `SET-DOMINATOR`, do `Fix-vertex`
  on `NEWDOM`, do
    { dominator← source, `Fix-vertex`}

Self-removal:
  If $\delta(v) = 1$, **then**
    send message `DISCONNECTED` to neighbor
    send message `SET-DOMINATOR` to neighbor
  Else, run distributed BFS algorithm from this
    vertex.
    If some vertex is not reached **then**
      `Fix-vertex`
    Else
      `Remove-vertex`
    End-If
  End-If

Fix-vertex:
  If v is non-fixed, **then**
    set v to fixed
    send message `NEWDOM` to neighbors
    ask the degree of each non-fixed,
      non-removed neighbor
    send message `TRY-DISCONNECT` to
      neighbors, according to increasing degree order
  End-If

Remove-vertex:
  send to active neighbors the message
    `DISCONNECTED`
  If there is no dominator, **then**
    ask the degree of active neighbors
    set u to neighbor with highest degree
  Else
    set u to dominating neighbor
  End-If
  send message `SET-DOMINATOR` to vertex u

Figure 7–2: Actions for a vertex v in the distributed algorithm.

Theorem 31 *The distributed algorithm presented in Figure 7–2 finds a correct CDS.*

Proof: One of the basic differences between the structure of connected dominating sets created by the distributed algorithm and the centralized algorithm is that we now require connectedness in k steps, i.e., the diameter of the subgraph induced by the resulting CDS is at most k. Of course, this implies that the final solution is connected.

To show that the result is a dominating set, we argue similarly to what was proved for Algorithm 19. At each iteration, a vertex will be either removed from the solution, or set to be in the final CDS. In the case that a vertex is removed, it must be dominated by a neighbor, otherwise it will send the message `SET-DOMINATOR` to one of its neighbors. Thus, each vertex not in the solution is directly connected to some other vertex which is in the solution. This shows that the resulting solution is a DS, and, therefore a CDS.

Now, we show that the algorithm terminates. First, every vertex in the network is reached, because the network is supposed to be connected, and messages are sent from the initial vertex to all other neighbors. After the initial decision (to become fixed or to be removed from the CDS), a vertex just propagates messages from other vertices, and does not ask further information. Since the number of vertices is finite, this implies that the flow of messages will finish after a finite number of steps. Thus, the algorithm terminates, and returns a correct connected dominating set. □

7.4 Numerical Experiments

Computational experiments were run to determine the quality of the solutions obtained by the heuristic proposed for the MCDS problem. We implemented both the centralized and distributed versions of the algorithm using

the C programming language. The computer used was a PC with Intel processor and enough memory to avoid disk swap. The C compiler used was the `gcc` from the GNU project, without any optimization. The machine was running the Linux operating system.

In the computational experiments, the testing instances were created randomly. Each graph has 100 or 150 vertices distributed randomly over an area, varying from 100×100 to 180×180 square units. The edges of a unit-disk graph are determined by the size of the radius, whose value ranged from 20 to 60 units. The resulting instances were solved by an implementation of Algorithm 19, as well as by a distributed implementation, described in the previous section. For the distributed algorithm, we used the additional requirement of k-connectedness with $k = 20$. The algorithms used for comparison are the ones proposed in (Alzoubi et al., 2002) and (Butenko et al., 2002). They are referred to in the results (Tables 7–1 and 7–2) as AWF and BCDP, respectively.

The results show that the non-distributed version of Algorithm 19 consistently gives results which are not worse than any of the other algorithms. The distributed version of the algorithm gives comparable results, although not as good as the non-distributed version. This can be explained by the fact that the distributed implementation lacks the benefit of global information, used by Algorithm 19 for, *e.g.*, always finding the vertex with smallest degree. However, despite the restrictions on the distributed algorithm, it performs very well. It must also be noted that the resulting implementation is very simple compared to the other approaches, and therefore can be executed faster.

7.5 Concluding Remarks

In this chapter, we proposed a new approach to the minimum connected dominating set problem. The proposed heuristic algorithm is applied to ad

hoc wireless networks, which are modeled as unit disk graphs. The algorithm is especially valuable in situations where setup time is costly, since it maintains a feasible solution at any time during the computation and thus can be executed without interrupting the network operation. A distributed version of the algorithm is also presented, which tries to adapt the basic algorithmic idea to a distributed setting. The experimental results show that both algorithms are able to find good quality solutions, with values compared to some of the best algorithms. The above mentioned advantages and the simplicity of the proposed algorithm make it an attractive alternative when solving the MCDS problem in dynamic environments.

Table 7–1: Results of computational experiments for instances with 100 vertices, randomly distributed in square planar areas of size 100×100 and 120×120, 140×140, and 160×160. The average solutions are taken over 30 iterations.

Size	Radius	Average degree	AWF	BCDP	Distr.	Non-distr.
100×100	20	10.22	28.21	20.11	20.68	19.18
	25	15.52	20.00	13.80	14.30	12.67
	30	21.21	15.07	10.07	10.17	9.00
	35	27.50	11.67	7.73	8.17	6.30
	40	34.28	9.27	6.47	6.53	4.93
	45	40.70	7.60	5.80	6.13	4.17
	50	47.72	6.53	4.40	4.77	3.70
120×120	20	7.45	38.62	27.71	28.38	27.52
	25	11.21	26.56	18.26	19.00	17.78
	30	15.56	20.67	13.40	14.63	12.40
	35	20.84	15.87	10.03	11.27	9.13
	40	25.09	12.67	8.33	9.00	7.00
	45	30.25	10.47	7.23	7.67	5.77
	50	36.20	8.87	6.00	6.57	4.70
140×140	30	11.64	25.76	18.10	18.90	17.03
	35	15.51	20.00	13.33	14.37	12.73
	40	19.40	16.40	10.87	11.57	9.67
	45	23.48	13.33	9.03	9.40	7.77
	50	28.18	11.33	7.63	8.33	6.23
	55	33.05	9.80	6.57	7.17	5.30
	60	38.01	8.80	5.70	6.20	4.53
160×160	30	9.14	31.88	22.20	23.20	21.88
	35	12.21	24.50	17.07	17.36	16.18
	40	15.63	19.73	13.47	14.07	12.43
	45	19.05	16.33	11.07	11.40	9.93
	50	22.45	13.80	9.47	9.67	7.93
	55	26.72	12.20	7.80	8.33	6.87
	60	30.38	10.33	7.10	7.47	5.80

Table 7–2: Results of computational experiments for instances with 150 vertices, randomly distributed in square planar areas of size 120×120, 140×140, 160×160, and 180×180. The average solutions are taken over 30 iterations.

Size	Radius	Average degree	AWF	BCDP	Distr.	Non-distr.
120×120	50	54.51	9.47	6.30	6.70	4.63
	55	63.02	7.73	5.70	6.40	4.20
	60	71.12	6.67	4.83	5.53	4.00
	65	81.08	6.00	3.73	4.30	3.53
	70	89.22	5.00	3.23	3.67	3.10
	75	98.04	4.79	3.04	2.82	2.54
	80	104.64	4.64	2.82	2.09	2.00
140×140	50	42.91	11.60	7.60	8.33	6.13
	55	49.75	10.07	6.87	7.40	5.20
	60	57.74	8.33	5.87	6.80	4.47
	65	64.70	7.60	5.27	6.20	4.17
	70	72.04	6.93	4.83	5.50	4.00
	75	78.82	5.87	3.87	4.53	3.60
	80	86.55	5.47	3.33	3.93	3.33
160×160	50	33.94	14.00	9.63	10.07	8.43
	55	40.31	12.27	8.47	8.97	6.73
	60	45.89	10.73	7.30	8.13	5.73
	65	53.36	9.60	6.27	6.77	4.83
	70	58.75	8.67	6.10	6.60	4.43
	75	64.39	7.80	5.40	6.13	4.37
	80	72.05	6.60	4.63	5.80	4.00
180×180	50	27.92	17.60	11.57	12.37	10.37
	55	33.05	15.13	10.13	10.43	8.47
	60	38.09	12.40	8.50	9.23	7.27
	65	43.95	11.53	7.70	8.53	6.10
	70	48.75	10.13	7.17	7.43	5.33
	75	55.08	9.33	6.30	6.87	4.53
	80	60.73	8.33	5.70	6.47	4.33

CHAPTER 8
CONCLUSION

This dissertation discussed optimization problems occurring in telecommunication networks, particularly in the areas of multicast and wireless ad hoc network systems. The problems presented here have in common the presence of discrete variables that must be optimized to reach optimum solutions and minimizing the use of network resources. A computational view of these problems was presented, with results about complexity and the design of efficient algorithms.

In Chapter 1, an outline of the problems discussed in the dissertation was presented. Chapter 2 gave a thorough review of the research performed in the area of multicast networks. The chapter presented the motivation and the current basis for the work developed in the area of multicast systems.

The problem of streaming cache placement in multicast networks (SCPP) was introduced in Chapter 3. An introduction to the problem and motivations, together with relevant computational complexity issues was presented. It was shown that the SCPP on its different forms is $\mathcal{NP}$-hard, with novel and insightful transformations from the Satisfiability problem.

The limits of approximation for SCPP problems was discussed in Chapter 4. We proved that it is not possible in general to give good approximate solutions for cache placement problems, due to their inherent complexity. Reductions from difficult problems such as the Set Cover problem were used to prove these results.

Algorithms for solving the SCPP have been presented in Chapter 5. These were the first approximation algorithms proposed in the literature for this problem. We also proposed fast construction algorithms, which can be useful to find good solutions for the problem in practice.

In Chapter 6, a heuristic algorithm was proposed for the multicast routing problem (MRP). We described a construction algorithm that improves over existing techniques. This algorithm has been used in a GRASP restarting procedure to derive practical upper bounds for this problem.

The problem of computing a minimum sized backbone on wireless ad hoc networks was discussed in Chapter 7 of this dissertation. A new algorithm was proposed, which avoids the need of a set up time during the calculation of a network backbone. The results of this algorithm also show that it is very competitive in terms of solution quality and computational time. A distributed version of the algorithm has been proposed, and computational results reported.

Many questions remain as important future research topics, and it was given in each chapter of the dissertation a list of interesting issues for work in the area. In general, for each of the discussed problems it is of high interest to develop upper and lower bounds on approximation. Similarly, the development of new algorithms with better average case or worst case complexity are clearly open questions for future investigation.

REFERENCES

Aguilar, L., Garcia-Luna-Aceves, J., Moran, D., Graighill, E., and Brungardt, R. (1986) "Architecture for a multimedia teleconferencing system," in: "Proceedings of the ACM SIGCOMM," 126–136, Association for Computing Machinery, Baltimore, MD.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993) *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Englewood Cliffs, NJ.

Aiex, R., Binato, S., and Resende, M. (2003) "Parallel GRASP with path-relinking for job shop scheduling," *Parallel Computing*, **29**, 393–430.

A.J. Frank, A. B., L.D. Wittie (1985) "Multicast communication on network computers," *IEEE Software*, **2** (3), 49–61.

Alzoubi, K. M., Wan, P.-J., and Frieder, O. (2002) "Distributed Heuristics for Connected Dominating Set in Wireless Ad Hoc Networks," *IEEE ComSoc/KICS Journal on Communication Networks*, **4** (1), 22–29.

Arora, S. and Lund, C. (1996) "Hardness of Approximations," in: D. Hochbaum (Ed.) "Approximation Algorithms for NP-hard Problems," PWS Publishing, Boston, MA.

Awerbuch, B. and Peleg, D. (1990) "Network synchronization with polylogarithmic overhead," in: "Proc. 31st Symp. Found. Computer Science," 514–522, IEEE Computer Society, New York, NY.

Baker, B. S. (1994) "Approximation algorithms for NP-complete problems on planar graphs," *Journal of the ACM (JACM)*, **41** (1), 153–180.

Baldi, M., Ofek, Y., and Yener, B. (1997) "Adaptive Real-Time Group Multicast," in: "Proceedings of the Conference on Computer Communications (IEEE Infocom)," 683, IEEE, San Francisco, CA.

Ballardie, A., Francis, P., and Crowcroft, J. (1993) "Core-based trees (CBT) – An architecture for scalable inter-domain multicast routing," *Computer Communication Review*, **23** (4), 85–95.

Baoxian, Z., Yue, L., and Changjia, C. (2000) "An Efficient Delay-Constrained Multicast Routing Algorithm," in: "International Conference on Communication Technologies (ICCT 2000)," S07.2, IEEE, San Francisco, CA.

Bauer, F. (1996) *Multicast routing in point-to-point networks under constraints*, Ph.D. thesis, University of California, Santa Cruz.

Bauer, F. and Varma, A. (1995) "Degree-Constrained Multicasting in Point-to-Point Networks," in: "Proceedings IEEE INFOCOM '95, The Conference on Computer Communications," 369–376, IEEE, San Francisco, CA.

Bauer, F. and Varma, A. (1997) "ARIES: A Rearrangeable Inexpensive Edge-Based On-Line Steiner Algorithm," *IEEE Journal of Selected Areas in Communications*, **15** (3), 382–397.

Bazaraa, M., Jarvis, J., and Sherali, H. (1990) *Linear Programming and Network Flows*, John Wiley and Sons, New York, NY, 2nd edition.

Bellman, R. (1957) *Dynamic Programming*, Princeton University Press, Princeton, NJ.

Bellman, R. E. (1958) "On a routing problem," *Quarterly of Applied Mathematics*, **16**, 87–90.

Berry, L. T. M. (1990) "Graph Theoretic Models for Multicast Communications," *Computer Networks and ISDN Systems*, **20** (1), 95–99.

Bharath-Kumar, K. and Jaffe, J. (1983) "Routing to multiple destinations in computer networks," *IEEE Transactions on Communications*, **31** (3), 343–351.

Blokh, D. and Gutin, G. (1996) "An approximate algorithm for combinatorial optimization problems with two parameters," *Australasian J. Combin.*, **14**, 157–164.

Butenko, S., Cheng, X., Du, D.-Z., and Pardalos, P. M. (2002) "On the construction of virtual backbone for ad hoc wireless network," in: S. Butenko, R. Murphey, and P. M. Pardalos (Eds.) "Cooperative Control: Models, Applications and Algorithms," 43–54, Kluwer Academic Publishers, Dordrecht, Neterlands.

Calvert, K. L., Zegura, E. W., and Donahoo, M. J. (1995) "Core Selection Methods for Multicast Routing," in: "IEEE ICCCN '95," 638–642, IEEE, Las Vegas, Nevada.

Chen, G., Houle, M., and Kuo, M. (1993) "The Steiner problem in distributed computing systems," *Information Sciences*, **74** (1), 73–96.

Chen, S., Günlük, O., and Yener, B. (1998) "Optimal packing of group multicastings," in: "Proc. IEEE INFOCOM'98," 980–987, IEEE, San Francisco, CA.

Cheng, C., Riley, R., Kumar, S., and Garcia-Luna-Aceves, J. (1989) "A loop-free extended Bellman-Ford routing protocol without bouncing effect," *ACM Computer Commun. Rev.*, **19** (4), 224–236.

Cheng, X., Huang, X., Li, D., and Du, D.-Z. (2003) "Polynomial-Time Approximation Scheme for Minimum Connected Dominating Set in Ad Hoc Wireless Networks," Submitted to Networks.

Chiang, C., Gerla, M., and Zhang, L. (1998) "Adaptive Shared Tree Multicast in Mobile Wireless Networks," in: "Proceedings of GLOBECOM '98," 1817–1822, IEEE, San Francisco, CA.

Chockler, G. V., Huleihel, N., Keidar, I., and Dolev, D. (1996) "Multimedia Multicast Transport Service for Groupware," in: "TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies," 43–54, IEEE, San Francisco, CA.

Chow, C. (1991) "On multicast path finding algorithms," in: "Proc. IEEE INFOCOMM '91," 1274–1283, IEEE, San Francisco, CA.

Chung, S.-J., Hong, S.-P., and Huh, H.-S. (1997) "A fast multicast routing algorithm for delay-sensitive applications," in: "IEEE GLOBECOM'97," 1898–1902, IEEE, San Francisco, CA.

Correa, R., Gomes, F., Oliveira, C. A., and Pardalos, P. M. (2003) "A parallel Implementation of an Asynchronous Team to the Point-to-point Connection Problem," *Parallel Computing*, **29** (4), 447–466.

Dalal, Y. and Metcalfe, R. (1978) "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM*, **21** (12).

Das, B. and Bharghavan, V. (1997) "Routing in Ad-Hoc Networks Using Minimum Connected Dominating Sets," in: "International Conference on Communications," 376–380, IEEE, San Francisco, CA.

Deering, S. (1988) "Multicast Routing in Internetworks and Extended LANs," in: "ACM SIGCOMM Summer 1988," 55–64, Association for Computing Machinery, Vancouver, BC, Canada.

Deering, S. and Cheriton, D. (1990) "Multicast routing in datagram internetworks and extended LANs," *ACM Transactions on Computer Systems*, 85–111.

Deering, S., Estrin, D., Farinacci, D., Jacobson, V., Liu, C.-G., and Wei, L. (1994) "An Architecture for Wide-Area Multicast Routing," *Computer Communication Review*, **24** (4), 126–135.

Deering, S., Estrin, D. L., Farinacci, D., Jacobson, V., Liu, C.-G., and Wei, L. (1996) "The PIM architecture for wide-area multicast routing," *IEEE/ACM Transactions on Networking*, **4** (2), 153–162.

Dijkstra, E. W. (1959) "A note on two problems in connexion with graphs," *Numer. Math.*, **1**, 269–271.

Diot, C., Dabbous, W., and CrowCroft, J. (1997) "Multipoint Communication: a survey of protocols, functions, and mechanisms," *IEEE Journal on Selected Areas in Communications*, **15** (3).

Doar, M. and Leslie, I. (1993) "How bad is naive multicast routing," in: "Proceedings of the IEEE INFOCOM," 82–89, IEEE, San Francisco, CA.

Du, D. and Pardalos, P. (Eds.) (1993a) *Network Optimization Problems: Algorithms, Complexity and Applications*, World Scientific, Singapore.

Du, D.-Z., Lu, B., Ngo, H., and Pardalos, P. M. (2001) "Steiner tree problems," in: C. Floudas and P. Pardalos (Eds.) "Encyclopedia of Optimization," volume 5, 227–290, Kluwer Academic Publishers, Dordrecht, Neterlands.

Du, D.-Z. and Pardalos, P. M. (1993b) "Subset Interconnection designs: Generalizations of spanning trees and Steiner trees," in: "Network Optimization Problems," 111–124, World Scientific, Singapore.

Ellis, C., Gibbs, S., and Rein, G. (1991) "Groupware: Some issues and experiences," *Commun. ACM*, **34** (1), 39–58.

Eriksson, H. (1994) "MBONE: The multicast backbone," *Communications of ACM*, **37** (8).

Feige, U. (1998) "A Threshold of ln n for Approximating Set Cover," *Journal of the ACM*, **45** (4), 634–652.

Feng, G. and Yum, T. P. (1999) "Efficient multicast routing with delay constraints," *International Journal of Communication Systems*, **12**, 181–195.

Feo, T. and Resende, M. (1995) "Greedy randomized adaptive search procedures," *J. of Global Optimization*, **6**, 109–133.

Ferrari, D. and Verma, D. C. (1990) "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, **8**, 368–379.

Ford, L. (1956) "Network Flow Theory," *Paper p-923*, RAND Corporation, Santa Monica, California.

Forsgren, A. and Prytz, M. (2002) "Dimensioning multicast-enabled communications networks," *Networks*, **39**, 216–231.

Fujinoki, H. and Christensen, K. J. (1999) "The New Shortest Best Path Tree (SBPT) Algorithm for Dynamic Multicast Trees," in: "24th Conference on Local Computer Networks," 204–211, IEEE, San Francisco, CA.

Gallager, R. G., Humblet, P. A., and Spira, P. M. (1983) "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Programming Languages and Systems*, **5** (1), 66–77.

Garey, M. R. and Johnson, D. S. (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco CA.

Goemans, M. X. and Williamson, D. P. (1995) "A General Approximation Technique for Constrained Forest Problems," *SIAM J. Comp.*, **24**.

Gomes, F., Menezes, C., Lima, A., and Oliveira, C. (1998) "Asynchronous Organizations for Solving the Point-to-Point Connection Problem," in: "Proc. of the Intl. Conference on Multiagents Systems (ICMAS)," 144–149, IEEE Computer Society, Paris, France.

Gomes, F. C., Pardalos, P. M., Oliveira, C. A., and Resende, M. G. (2001) "Reactive GRASP with Path Relinking for Channel Assignment in Mobile Phone Networks," in: "Proceedings of the 5th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications," 60–67, ACM Press, Rome, Italy.

Guha, S. and Khuller, S. (1998) "Approximation Algorithms for Connected Dominating Sets," *Algorithmica*, **20** (4), 374–387.

Han, L. and Shahmehri, N. (2000) "Secure Multicast Software Delivery," in: "IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'00)," 207–212, IEEE, San Francisco, CA.

Hong, S., Lee, H., , and Park, B. H. (1998) "An efficient multicast routing algorithm for delay-sensitive applications with dynamic membership," in: "Proceedings of IEEE INFOCOM'98," 1433–1440, IEEE, San Francisco, CA.

Hunt III, H. B., Marathe, M., Radhakrishnan, V., Ravi, S., Rosenkrantz, D., and Stearns, R. (1998) "NC-approximation schemes for NP- and PSPACE-hard problems for geometric graphs," *J. Algorithms*, **26**, 238–274.

Hwang, F. and Richards, D. (1992) "Steiner tree problems," *Networks*, **22**, 55–89.

Hwang, F., Richards, D., and Winter, P. (1992) *The Steiner tree problem*, volume 53 of *Annals of Discrete Mathematics*, North-Holland, Amsterdam, Netherlands.

Im, Y., Lee, Y., and Choi, Y. (1997) "A Delay Constrained Distributed Multicast Routing Algorithm," *Computer Communications*, **20** (1).

Imase, M. and Waxman, B. (1991) "Dynamic Steiner tree problems," *SIAM J. Discrete Math.*, **4**, 369–384.

Jia, X. (1998) "A Distributed Algorithm of Delay-Bounded Multicast Routing for Multimedia Applications in Wide Area Networks," *IEEE/ACM Transactions on Networking*, **6** (6), 828–837.

Jia, X., Pissinou, N., and Makki, K. (1997) "A real-time multicast routing algorithm for multimedia applications," *Computer Commun. J.*, **20** (12), 1098–1106.

Jiang, X. (1992) "Routing broadband multicast streams," *Computer Communications*, **15** (1), 45–51.

Kheong, C., Siew, D., and Feng, G. (2001) "Efficient Setup for Multicast Connections Using Tree-Caching," in: "Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications," 249–258, IEEE, San Francisco, CA.

Kompella, V., Pasquale, J., and Polyzos, G. (1992) "Multicasting for Multimedia Applications," in: "Proceedings of IEEE INFOCOM'92," 2078–2085, IEEE, San Francisco, CA.

Kompella, V., Pasquale, J., and Polyzos, G. (1993a) "Two Distributed Algorithms for the Constrained Steiner Tree Problem," in: "Proceedings of the Second International Conference on Computer Communications and Networking (ICCCN'93)," 343–349, IEEE, San Francisco, CA.

Kompella, V., Pasquale, J., and Polyzos, G. (1996) "Optimal multicast routing with quality of service constraints," *Journal of Network and Systems Management*, **4** (2), 107–131.

Kompella, V. P., Pasquale, J. C., and Polyzos, G. C. (1993b) "Multicast routing for multimedia communication," *IEEE/ACM Trans. Networking*, **1** (3), 286–292.

Kou, L., Markowsky, G., and Berman, L. (1981) "A fast algorithm for Steiner trees," *Acta Informatica*, **15**, 141–145.

Kruskal, J. (1956) "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, **7** (48), 50.

Kumar, G., Narang, N., and Ravikumar, C. (1999) "Efficient Algorithms for Delay Bounded Multicast Tree Generation for Multimedia Applications," in: "6th International Conference on High Performance Computing," 169–173, Springer-Verlag, Berlin.

Levine, B. and Garcia-Luna-Aceves, J. (1998) "A Comparison of Reliable Multicast Protocols," *Multimedia Systems*, **6** (5), 334–348.

Li, B., Golin, M., Italiano, G., Deng, X., and Sohraby, K. (1999) "On the optimal placement of Web proxies in the Internet," in: "Proceedings of IEEE INFOCOM 99," IEEE, San Francisco, CA.

Li, C., McCormick, S., and Simchi-Levi, D. (1992) "The point-to-point delivery and connection problems: complexity and algorithms," *Discrete Applied Math.*, **36**, 267–292.

Malpani, N., Welch, J., and Vaidya, N. (2000) "Leader Election Algorithms for Mobile Ad Hoc Networks," in: "Proc. Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications," 96–103, ACM Press, Boston, MA.

Mao, Z., Johnson, D., Spatscheck, O., van de Merwe, J. E., and Wang, J. (2003) "Efficient and Robust Streaming Provisioning in VPNs," in: "Proceedings of the WWW2003 (May 20-24 2003), Budapest, Hungary," WWW03, Budapest, Hungary.

Marathe, M. V., Breu, H., Hunt III, H. B., Ravi, S. S., and Rosenkrantz, D. J. (1995) "Simple Heuristics for Unit Disk Graphs," *Networks*, **25**, 59–68.

Mokbel, M. F., El-Haweet, W. A., and El-Derini, M. N. (1999) "A Delay Constrained Shortest Path Algorithm for Multicast Routing in Multimedia Applications," in: "Proceedings of IEEE Middle East Workshop on Networking," IEEE, San Francisco, CA.

Moy, J. (1994a) "Multicast Extensions to OSPF, RFC 1584 – IETF Network Working Group," On-line document: http://www.ietf.org/, last accessed on June 1, 2004.

Moy, J. (1994b) "OSPF Version 2, RFC 1583 – IETF Network Working Group," On-line document: http://www.ietf.org/, last accessed on June 1, 2004.

Natu, M. G. (1995) *Network Loading and Connection Problems*, Ph.D. thesis, Operations Research Dept., North Carolina State University.

Noronha, C. and Tobagi, F. (1994) "Optimum Routing of Multicast Streams," in: "IEEE INFOCOM 94," 865–873, IEEE, San Francisco, CA.

Ofek, Y. and Yener, B. (1997) "Reliable Concurrent Multicast from Bursty Sources," *IEEE Journal on Selected Areas in Communications*, **15** (3), 434–444.

Oliveira, C., Pardalos, P., Prokopyev, O., and Resende, M. (2003a) "Streaming Cache Placement Problems: Complexity and Algorithms," *Technical report*, Department of Industrial and Systems Engineering, University of Florida.

Oliveira, C. A., Pardalos, P. M., and Resende, M. (2003b) "GRASP with path-relinking for the QAP," in: "5th Metaheuristics International Conference," 57.1–57.6, MIC03, Kyoto, Japan.

Papadimitriou, C. and Yannakakis, M. (1991) "Optimization, approximation, and complexity classes," *J. Comput. System Sci.*, **43**, 425–440.

Papadimitriou, C. H. and Steiglitz, K. (1982) *Combinatorial Optimization*, Prentice Hall, Englewood Cliffs, NJ.

Pardalos, P., Hsu, F., and Rajasekaran, S. (Eds.) (2000) *Mobile Networks and Computing*, volume 52 of *DIMACS Series*, American Mathematical Society, Providence, RI.

Pardalos, P. and Khoury, B. (1995) "An Exact Branch and Bound Algorithm for the Steiner Problem in Graphs," in: D.-Z. D. . M. Li (Ed.) "Proceedings of COCOON'95 (Xi'an, China, August 24-26, 1995," volume 959 of *Lecture Notes in Computer Science*, 582–590, Springer-Verlag, Berlin.

Pardalos, P. and Khoury, B. (1996) "A Heuristic for the Steiner Problem on Graphs," *Comp. Opt. & Appl.*, **6**, 5–14.

Pardalos, P., Khoury, B., and Du, D.-Z. (1993) "A test problem generator for the steiner problem in graphs," *ACM Transactions on Mathematical Software*, **19** (4), 509–522.

Pardalos, P. M. and Du, D. (Eds.) (1998) *Network Design: Connectivity and Facilities Location*, volume 40 of *DIMACS Series*, American Mathematical Society, Providence, RI.

Park, J. and Park, C. (1997) "Development of a Multi-user & Multimedia Game Engine Based on TCP/IP," in: "Proceedings of IEEE Pacific Rim Conference on Communications Computers and Signal Processing," 101–104, IEEE, Victoria, B.C. Canada.

Pasquale, J., Polyzos, G., and Xylomenos, G. (1998) "The multimedia multicasting problem," *ACM Multimedia Systems Journal*, **6** (1), 43–59.

Paul, P. and Raghavan, S. V. (2002) "Survey of Multicast Routing Algorithms and Protocols," in: "Proceedings of the Fifteenth International Conference on Computer Communication (ICCC 2002)," IEEE, San Francisco, CA.

Prim, R. (1957) "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal*, **36**, 1389–1401.

Priwan, V., Aida, H., and Saito, T. (1995) "The Multicast Tree Based Routing for The Complete Broadcast Multipoint-to-Multipoint Communications," *IEICE Transactions on Communications*, **E78-B** (5).

Prytz, M. (2002) *On optimization in Design of Telecommunications Networks with Multicast and Unicast Traffic*, Ph.D. thesis, Dept. of Mathematics, Royal Institute of Technology, Stockholm, Sweden.

Prytz, M. and Forsgren, A. (2002) "Dimensioning of a multicast network that uses shortest path routing distribution trees," *Paper trita-mat-2002-os1* , Department of Mathematics, Royal Institute of Technology, Stockholm, Sweden.

Ramanathan, S. (1996) "Multicast tree generation in networks with asymmetric links," *IEEE/ACM Trans. Networking*, **4** (4), 558–568.

Resende, M. G. C. and Ribeiro, C. C. (2003) "A GRASP with path-relinking for private virtual circuit routing," *Networks*, **41**, 104–114.

Rouskas, G. N. and Baldine, I. (1996) "Multicast routing with end-to-end delay and delay variation constraints," in: "IEEE INFOCOM'96," 353–360, IEEE, San Francisco, CA.

Sabri, S. and Prasada, B. (1985) "Video conferencing systems," *Proc. of the IEEE*, **73** (4), 671–688.

Salama, H. F., Reeves, D. S., and Viniotis, Y. (1996) "Shared Multicast Trees and the Center Selection Problem: A Survey," *Tr-96/27*, Dept. of Electrical and Computer Engineering, NCSU.

Salama, H. F., Reeves, D. S., and Viniotis, Y. (1997a) "The Delay-Constrained Minimum Spanning Tree Problem," in: "2nd IEEE Symposium on Computers and Communications (ISCC '97)," 699–704, IEEE Computer Society, New York, NY.

Salama, H. F., Reeves, D. S., and Viniotis, Y. (1997b) "A Distributed Algorithm for Delay-Constrained Unicast Routing," in: "Proc. IEEE INFOCOM'97," IEEE, Kobe, Japan.

Salama, H. F., Reeves, D. S., and Viniotis, Y. (1997c) "Evaluation of Multicast Routing Algorithms for Real-Time Communication on High-Speed Networks," *IEEE Journal on Selected Areas In Communications*, **15** (3).

Sanchis, L. A. (2002) "Experimental Analysis of Heuristic Algorithms for the Dominating Set Problem," *Algorithmica*, **33**, 3–18.

Semeria, C. and Maufer, T. (1996) "Introduction to IP Multicast Routing," Internet draft (IETF), available on http://nms.lcs.mit.edu/6.829-f01/draft-ietf-mboned-intro-multicast-03.txt, last accessed on 6/2/2004.

Shaikh, A. and Shin, K. G. (1997) "Destination-Driven Routing for Low-Cost Multicast," *IEEE Journal of Selected Areas in Communications*, **15** (3), 373–381.

Sloman, M. S. and Andriopoulos, X. (1985) "Routing Algorithm for Interconnected Local Area Networks," *Computer Networks and ISDN Systems*, **9** (2), 109–130.

Sriram, R., Manimaran, G., and Siva Ram Murthy, C. (1998) "Algorithms for delay-constrained low-cost multicast tree construction," *Computer Communications*, **21** (18), 1693–1706.

Sriram, R., Manimaran, G., and Siva Ram Murthy, C. (1999) "A rearrangeable algorithm for the construction of delay-constrained dynamic multicast trees," *IEEE/ACM Transactions on Networking*, **7** (4), 514–529.

Stojmenovic, I., Seddigh, M., and Zunic, J. (2001) "Dominating sets and neighbor elimination based broadcasting algorithms in wireless networks," in: "Proc. IEEE Hawaii Int. Conf. on System Sciences," IEEE, San Francisco, CA.

Sun, Q. and Langendoerfer, H. (1995) "Efficient Multicast Routing for Delay-Sensitive Applications," in: "Proceedings of the Second Workshop on Protocols for Multimedia Systems (PROMS'95)," 452–458, PROMS, Salzburg, Austria.

Takahashi, H. and Matsuyama, A. (1980) "An approximate solution for the Steiner problem in graphs," *Mathematica Japonica*, 573–577.

Talukdar, S. N. and de Souza, P. S. (1990) "Asynchronous Teams," *Second SIAM Conf. on Linear Algebra: Signals, Systems, and Control, San Francisco*.

Thomas II, T. (1998) *OSPF network design solutions*, Cisco Systems, Indianapolis, IN.

Waitzman, D., Partridge, C., and Deering, S. (1988) "Distance Vector Multicast Routing Protocol, RFC 1075 – IETF Network Working Group," On-line document: http://www.ietf.org/, last accessed on June 1, 2004.

Wall, D. (1980) *Mechanisms for Broadcast and selective broadcast*, Ph.D. thesis, Stanford University.

Wall, D. (1982) *Mechanisms for broadcast and selective broadcast*, Ph.D. thesis, Computer Science Department, Stanford University.

Wan, P.-J., Du, D.-Z., and Pardalos, P. M. (Eds.) (1998) *Multichannel Optical Networks: Theory and Practice*, volume 46 of *DIMACS Series*, American Mathematical Society, Providence, RI.

Wang, C.-F., Liang, C.-T., and Jan, R.-H. (2002) "Heuristic algorithms for packing of multiple-group multicasting," *Computers & Operations Research*, **29** (7), 905–924.

Waxman, B. (1988) "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, **6** (9), 1617–1622.

Wei, L. and Estrin, D. (1994) "The trade-offs of multicast trees and algorithms," in: "Proceedings of ICCCN'94," IEEE, San Francisco, CA.

Westbrook, J. and Yan, D. (1993) "Greedy Algorithms for the on-line Steiner tree and generalized Steiner problems," in: F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides (Eds.) "Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993," volume 709 of *Lecture Notes in Computer Science*, 621–633, Springer-Verlag, Berlin.

Wi, S. and Choi, Y. (1995) "A Delay-Constrained Distributed Multicast Routing Algorithm," in: "Proceeding of the twelfth International Conference on Computer Communication (ICCC'95)," 883–838, IEEE, San Francisco, CA.

Winter, P. (1987) "Steiner problem in networks: a survey," *Networks*, **17**, 129–167.

Winter, P. and Smith, J. M. (1992) "Path-distance heuristics for the Steiner problem in undirected networks," *Algorithmica*, 309–327.

Yum, T. S., Chen, M. S., and Leung, Y. W. (1995) "Video bandwidth allocation for multimedia teleconferences," *IEEE Trans. on Commun.*, 457–465.

Zhong, W. D., Onozato, Y., and Kaniyil, J. (1993) "A copy network with shared buffers for large-scale multicast ATM switching," *IEEE/ACM Transactions on Networking*, **1** (2), 157–165.

Zhu, Q., Parsa, M., and Garcia-Luna-Aceves, J. J. (1995) "A source-based algorithm for delay-constrained minimum-cost multicasting," in: "Proc. IEEE INFOCOM95," 377–385, IEEE, San Francisco, CA.

BIOGRAPHICAL SKETCH

Carlos Oliveira was born in Fortaleza, Brazil. Starting his studies in a technical school in the area of civil construction, he changed to the exciting area of computer science and decided to make it his career field. Carlos' interests in mathematics led to the choice of operations research as his specific field of study.

Carlos Oliveira holds a bachelor's degree in computer science, from the State University of Ceará, Brazil (1998), where he graduated with high honors. During that years he spent more time studying computer networks than optimization. Even during his period as an undergraduate student, he had assistantships for working with research projects. Among them was the project "minhoco" to build a network prototype for teaching purposes. He also started working with some optimization problems, such as the point-to-point connection problem and the frequency assignment problem.

Carlos Oliveira completed a master's degree in computer science in the Artificial Intelligence Laboratory of the Federal University of Ceará (2000), now in the area of applied optimization. Carlos spent the year of 2000 as a lecturer in the Computer Science Department of the Federal University of Ceará. He decided to pursue a career as a researcher, and applied to the Department of Industrial and Systems Engineering in the University of Florida.

Since the Spring of 2001, Carlos has been a full time student in the Ph.D. program of the ISE department, University of Florida. His advisor is Dr. Panos M. Pardalos. Carlos has been a teaching assistant for courses such as applied probabilities, web-based decision systems, and simulation. He is also a

research assistant in the Center of Applied Optimization. His research interests include discrete optimization, mathematical programming, construction and analysis of algorithms, as well as application areas such as telecommunications, assignment theory, and computational biology.