CHALMERS

Investigation of Interconnect Algorithms in a Radio Base Station Environment

Master of Science Thesis

GABRIEL RIZOPULOS
NIKLAS STRÖM

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2012

Investigation of Interconnect Algorithms in a Radio Base Station Environment

GABRIEL RIZOPULOS
NIKLAS STRÖM

Examiner: ELAD M. SCHILLER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden. June 2012

Abstract

We investigate the problem of allocating communication paths in the interconnect network within radio base stations (RBS). With the enlargement of RBSs to accommodate the growing capacity requirements there is a need to automate the allocation. The problem is modelled as the Unsplittable Flow Problem and several algorithms described in the literature in the area are evaluated against the specific circumstances. An evaluation framework is designed and a few algorithms including shortest path and branch-and-cut based are implemented. Based on the frame work measurements, one can give some recommendations on when the different algorithms may be appropriate and the strengths and weaknesses for each. Interestingly, our evaluation results do not show improvements in solution quality for the well-known branch-and-cut algorithms. We note that the framework can be easily extended to evaluate many new algorithms and compare the existing algorithms on real and syntactic data.

Acknowledgements

We would like to thank the people at Ericsson who gave us the opportunity to do this thesis. Especially we would like to thank our supervisor Peter Karlsson whose advice has greatly influenced this thesis. Without him this would not have been possible. We would also like to thank Peter Eriksson, Staffan Ehnebom and Åke Gustavsson for their valuable feedback, support and discussions as well as everyone else at Ericsson who has helped us during our research.

Contents

ABBREVIATIONS

COIN-OR The Computational Infrastructure for Operations Research

DIP Decomposition in Integer Programming

EDP Edge Disjoint Paths

HetNet Heterogenous Network

ILP Integer Linear Programming/Integer Linear Program

IP Integer Programming

ISP Internet Service Provider

LP Linear Programming/Linear Program

LTE Long Term Evolution

MCFP Multi-Commodity Flow Problem

MCPF Minimum Capacity Path First

MDP Maximum Disjoint Paths

MSPF Minimum Shortest Path First

OD Origin-Destination

ODIMCFP Origin-Destination Integer Multi-Commodity Flow Problem

RBS - Radio Base Station

RE Radio Equipment

REC Radio Equipment Controller

RMP Restricted Master Problem

SEC Subtour Elimination Constraint

SSP Simple Shortest Path

TDMA Time Division Multiple Access

TSP Traveling Salesperson Problem

UFP Unsplittable Flow Problem

DEFINITIONS

Arc A link between two nodes in a Graph

Capacity Number of timeslots available on an arc in the network.

Channel - A path through an RBS interconnect network with a certain reserved capacity.

Column generation A solution technique for solving LPs by generating variables in the formulation that is potentially nonzero.

Competitive ratio A bound on the difference between an online algorithm and some optimal offline algortihm.

Cuts See Cutting planes.

Cutting planes Inequalities added to a LP relaxation in order to obtain a result closer to the solution of the ILP.

Degree The degree of a node in a network is the number of arcs from/to that node. To distinguish the arcs from a node from the arcs to a node, sometimes outdegree and indegree is used. In our case, in- and outdegree is the same and we refer to it simply as degree.

Dual For every LP, called the primal problem, the duality theory of linear programming says there is a dual problem, an alternative formulation but with the same optimal value. For a maximization problem the dual is a minimization problem and vice versa. If one knows a finite optimal solution to the primal problem, the complementary slackness theorem allows one to find the optimal solution to the dual problem by solving a system of equations. For each constraint in the primal problem there is a variable in the dual problem. Hence one can talk of the dual variable associated with a constraint in the primal formulation.

Greedy algorithm A greedy algorithm makes choices based on some local optimum, hoping that doing whats best right now also leads to the optimal solution in the long run. A key characteristic of a greedy algorithm is that it does not change its decisions once they have been made.

Integer Linear Programming Similar to LP but with added constraints that the variables are integer.

Linear Programming Linear programming is a way of modelling a problem as an optimization of a linear function subject to a set of linear constraints.

LP relaxation A simplified version of an ILP where the integer constraints have been removed, resulting in a LP.

Maximum degree The maximum degree of all the nodes in the network.

NP-Complete An NP-Complete problem is a decision problem where it is believed to be impossible to find a polynomial time algorithm capable of solving it optimally.

NP-Hard An NP-Hard problem is any problem that is at least as hard as an NP-Complete problem, meaning that if you can efficiently solve the NP-Hard problem then you can also efficiently solve NP-Complete problems.

Offline algorithm An offline algorithm receives the complete set of request at once and can take all requests into account when making its decisions.

Online algorithm An online algorithm receives requests one at a time and must process the requests in that order, only taking into account the current request and the decisions in the past.

Path A path is a series of arcs, without circles, that connects one node to another.

Request A request for a path with a certain capacity between an origin and a destination node.

Requested capacity The capacity needed on each arc in a path in order to service a request.

Site Geographic place where an RBS in located.

1. Introduction

Mobile data communication had a tremendous growth during the last years and it is expected to further increase at exponential rate in the coming decade [14]. The increased use of smartphones, tablets and ultra-light laptops have been the driving factors so far. A forecast by Ericsson, a world-leading provider of telecommunications equipment, is that there will be 50 billion connected devices in the world by year 2020 [15]. Hand-held devices and laptops are only a part of this number, other examples of devices that will be connected are televisions, vehicles and utility meters for all kinds of applications. Machine-to-machine communication is also believed to grow as more devices and services collaborate to make life more comfortable.

To meet the increasing demand it is necessary to increase the capacity of the mobile communication networks. More specifically it is necessary to increase the capacity on the outer-most links of the network. This link has a Radio Base Station (RBS) in one end, connected to the network backbone, and a user device such as a cell phone in the other end. The RBS communicates with the user devices by radio signals. Increasing the capacity is done by using advanced technologies, such as the Advanced Long Term Evolution (LTE-Advanced) [29], by extending the infrastructure with more radio base stations and by upgrading existing RBSs. The capacity can be increased by introducing more advanced modulation techniques and by using a larger frequency bandwidth for the carrier. However, advanced modulation techniques can only increase the data rate during good radio conditions, usually not obtainable anywhere but close to the antenna. It is also difficult to increase the capacity by using a larger frequency bandwidth since building a radio with large bandwidth is hard. Operators may also be limited by that they do not have the bandwidth as a continuous band which means that the carriers can not be contiguous. A large capacity increase thus requires installation of more radios. [24]

The greatest need for more bandwidth is in so called hot-spots, places of limited area where there are many users at the same time, such as train stations, airports, malls and large offices. A good approach for increasing the capacity is thus to install micro- or pico cells in hot-spots. These cells are basically low power radios with a much smaller range than those radios that are used for the macro cell which provides the basic coverage over a large area. The micro and pico cells can either be RBSs by themselves or be connected to an existing RBS, becoming a part of it. Having a single radio in an RBS may not be very cost efficient for an operator. Therefore, a good solution is to have a single RBS

comprising of a few radios with large range covering different sectors of the macro cell and a number of micro and pico cells within the macro cell providing extra coverage and capacity in areas that requires it. This solution with multiple access networks is called a heterogeneous network or a HetNet [16].

An example of an RBS with many low power radios is in large towers with several hundred storeys. Typically there are many radios throughout the tower providing a high capacity on all floors while the equipment for controlling all the radios is placed in the basement. This also reflects the conceptual model of an RBS. In one end are the antennas and the equipment for generating radio signals. These are grouped into a Radio Equipment (RE) unit and each RBS may have many REs. The other end consists of one or several Radio Equipment Controllers (REC). A REC, as the name suggests, controls a number of REs. It manages user calls, handles signal processing and pass data onto the mobile network backbone. A REC has a limit for how many REs it can manage, so for a large RBS with many high capacity REs several RECs will be needed. The RECs and REs are connected with a network consisting of cables and switches, the interconnect network. Each RE is managed by a few specific RECs, typically one or two, and it is necessary to have a path with guaranteed bandwidth between each pair of RE and its corresponding REC. Therefore it is necessary to set up virtual circuits over the interconnect network. Each link on the network has a limited bandwidth capacity and each virtual circuit will have a certain requested bandwidth. For a sufficiently large and complex interconnect network, deciding how to allocate the requested paths, which will also be called requests, is a non trivial problem.

1.1 Interconnect model

We describe the key terms and concepts that are needed for describing the studied problem. This section consider a model of the interconnect network and the policies that govern its allocation are described. The interested reader can find more information about mobile communication technology in [32], as well as in [34] which also provides further details about increasing the capacity in the networks through a technology called LTE-Advanced. For theoretical and practical insights into commonly used network models, such as those used in this work, we refer the readers to [3].

In order to describe the problem further it is necessary to understand the characteristics of the interconnect network. Figure 1.1 shows an abstraction of an RBS with an interconnect network connecting RECs and REs. The RECs are usually located in the same place while the REs may be distributed over a large area. The network infrastructure consists of copper cables, optical fibers and in large networks also switches. Each link in the network uses a Time Division Multiple Access (TDMA) protocol and has a certain capacity, or bandwidth, that is measured in the number of time-slots on the link. Thus, all time-slots over all links carry the same amount of data but a high bandwidth link has more time-slots than a link with less bandwidth. Point-to-point channels are set up over these links to connect specific pairs of RECs and REs and each channel requires a certain amount of bandwidth on each link along its path. For each channel from a

REC to an RE there is also a matching channel in the opposite direction, which may require a different capacity and can be allocated on a different path. Both link capacities and requested capacity of a channel are variable but it may be assumed that the highest requested capacity is lower than or equal to the smallest arc capacity. This is sometimes referred to as the no-bottleneck constraint [12]. The channels are semi-static, meaning that once set up they will not be changed or removed as long as no failure occurs or unless maintenance or reconfiguration is necessary, for example due to installation of new hardware. User telephone calls are routed through these channels and a take down of a channel can thus result in many disconnected calls. The channels may be set up both one at a time, such as after maintenance or after installation of a new RE, or many at once as could be the case when a new RBS is taken into operation or after a major capacity upgrade. [24]

In addition there exist the possibility that one may want to shift the available resources between different areas as part of normal operations. One reason for doing that is that the demand in a particular area can vary significantly between different points in time and this can then be used in order to reduce the number of hardware units necessary. One example of this can be that one has more resources in an office complex during office hours and then shift the resources to a residence area in the afternoon. This can lead to savings in terms of space, costs and energy consumption.[24] In all these cases there may already be existing channels that preferably should not be affected by the new channels since it is not desired to break any existing calls.

Since each REC can manage several REs, the number of REs is usually higher than the number of RECs in an RBS. The total number of units and the complexity of the interconnect network depend on the size of the RBS. A small RBS may consist of a single or a handful of REs and a single REC while the interconnect network is just a simple star, ring or chain connecting all REs to the REC. In such a network there may only be a single path available for each requested channel. An RBS with more RECs and REs may have a more advanced interconnect network where there are multiple paths to choose from for all channels. The increasing capacity requirements in the future may necessitate even larger RBSs. For such interconnect networks it will be necessary to have algorithms that can decide how to allocate the channels.

1.2 Purpose

The purpose of this thesis work is to find and evaluate several algorithms that can be used for automatic allocation of the interconnect resources in a future radio base station. In current radio base stations the interconnect networks are small enough so that there is no need for an algorithm for finding good channel allocations, it can easily be done by hand. Since the number of different variations of topologies is small the configurations can be hard coded. For the larger and more advanced interconnect networks that are expected in the future it will be more difficult to set up the channels in a way that efficiently uses the available capacities. The increasing amount of different topologies will also make it impractical for hard coded configurations. In addition there is the

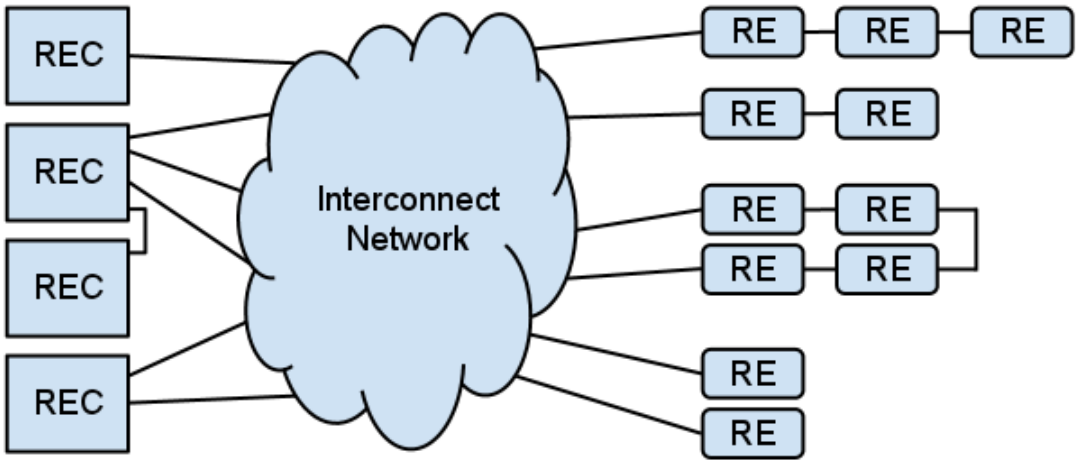


Figure 1.1: An interconnect network. REs are typically cascaded or in rings and RECs may have several paths to the REs to choose from.

possibility of a failure of one of the links connecting the nodes in the network. The traffic over the channels allocated on such a link will be disturbed and the channels must be reallocated to functioning links as fast as possible in order to provide service again. Thus the development of an allocation-algorithm that automatically maps requests to the available links in an efficient way without exceeding any capacity constraints is highly motivated.

1.3 Problem Definition

The main goal of the allocation algorithms is to find paths with sufficient capacities for all required channels to be allocated concurrently. There may also be additional goals such as minimizing latency, congestion, and energy consumption or allocating redundant paths with as few common links as possible that can be switched to in case of link failures. Depending on different policies the importance of the different goals may vary.

The main problem that this thesis investigates is what kind of algorithms that are most suitable for automating allocation in the RBS interconnect network. The suitability is evaluated based on the execution speed of the algorithms, how well they manage to allocate the requests and how hard they are to implement. We also investigate under which assumptions and problem characteristics those algorithms are suitable. We try to find if and how the relative performance of the algorithms depend on the size and connectivity of the topology and on the request set.

To perform the evaluations a framework where the algorithms can be implemented and evaluated is needed. Thus, as part of the contribution of this thesis, an evaluation framework is developed. Within this framework we define a set of measurements that are used to evaluate the implementations of the algorithms.

1.4 Method

The search for suitable algorithms is based on literature studies where a large number of articles about similar allocation problems have been surveyed. During the research suitable algorithms were identified based on how well their constraints and goals matched the RBS allocation problem.

The evaluation framework will evaluate the implementions of the algorithms both with fixed and randomly generated topologies and requests. The evaluation framework has been developed with an iterative pair programming process. A few algorithms with different approaches to solve the problem have then been implemented and evaluated with the help of the evaluation framework. The evaluations are based on results from a set of test scenarios.

1.5 Scope and limitations

The result of the thesis work is a recommendation of algorithms that can be used depending on the size and connectivity of the topology and on different policies. Both greedy and optimal algorithms are evaluated within our framework in order to find their suitability to the problem. We investigate a number of different variations of greedy algorithms where heuristics are used to guide the algorithms. The evaluation is based more on the measured performance rather than the theoretical worst case performance. The optimization goals that are investigated are primarily to maximize the number of serviced requests or the total allocated capacity while minimizing latency.

The exact topologies of the interconnect networks of the future are not yet finalized. Those proposed are still relatively small and simple and does not provide the full difficulties that future topologies will do. Therefore we do not limit ourselves to only topologies that are typical for future RBS systems. Experimental evaluations of the algorithms both include such graphs as well as some more general topologies. The randomly generated topologies of larger sizes are used in the evaluation in order to accentuate the differences in the relative performance of the algorithm implementations.

The algorithms are developed to run on desktop computers and not for actual use in an RBS system. This means that the choice of implementation language, resource limits and other constraints in the real system are not taken into account. While we aim for reasonable efficiency in the implementation of the algorithms, obtaining the best possible performance is not one of the goals of this thesis.

1.6 Structure of the report

The rest of this report is structured as follows. Chapter 2 further defines the problem and describes the previous work in the field. In Chapter 3 the algorithms selected for further analysis are described in detail. Chapter 4 describes the evaluation framework developed as well as the test cases used in the computational testing of the algorithms.

The results of these tests are presented in Chapter 5. We continue to discuss the results and their implication in Chapter 6, and finally present our conclusions in Chapter 7.

2. Theoretical Framework and Previous Work

The problem described in the previous chapter, and variations thereof, has been studied in the literature under several different names. Among them are Unsplittable Flow Problem (UFP) [27], Origin-Destination Integer Multi-Commodity Flow Problem(ODIMCFP) [9], Virtual Circuit Routing and Call Admission Control and Routing (CAC). These problems have a wide range of applications such as railroad scheduling, routing and admission in telecommunication networks, airline scheduling and transportation of commodities.

The multi-commodity flow problem (MCFP) is a network flow problem where there is more than one kind of flow. Each kind of flow is called a commodity and each flow should be routed through a network where the arcs have limited capacities. For each commodity there are source nodes with a certain supply of the commodity and sink nodes with a certain demand. A book frequently referenced to by research papers about the MCFP is [3], written by Ahuja, Magnanti and Orlin.

UFP and ODIMCFP are variants of the basic MCFP. Here we are restricted to only one source node for each commodity, called origin. There are also one or more sink nodes, called destination nodes. In addition there is also the constraint that the flow of a commodity between an origin and a destination node cannot be split over multiple paths.

In the Interconnect allocation setting we refer to the commodities as requests. A request corresponds to a request for a path with a certain capacity between an origin and a destination node.

2.1 Variations of the UFP

Several different variations of UFP optimization problems exist with different constraints and objective functions. These include uniform or variable link and requested capacities. There may also be a single source and multiple destinations or many Origin-Destination pairs (OD-pairs). The goal in the UFP may be to maximize the total flow or the number of satisfied OD-pairs. There may also be an associated benefit with each OD-pair, a cost for using each arc or both of these at the same time. Then, a secondary goal may be to minimize the cost or maximize the benefit. Another goal is to keep congestion as low as possible or, if the requested capacity of all OD-pairs can not be allocated simultaneously,

routing all requests in as few rounds as possible. The later is only possible if the requests are not permanent.

Algorithms can be used either *online* or *offline*. Offline algorithms have information about all requests in advance and try to consider all requests at once to find an optimal, or near optimal solution. Online algorithms consider each request individually at the time it arrives, without information about future requests. They must decide whether to allocate or reject a request depending on what is likely to be best in the end. Obviously an online algorithm for UFP cannot be optimal in the general case, since the result of an online algorithm depends on the order the requests arrive in.

2.2 Hardness

The Maximum Disjoint Paths problem (MDP) is a special case of the UFP where both arc capacities and the requested capacities equal one. Then, each arc can only be part of the path of a single request. The goal of the MDP is to find the maximum number of requests that can be allocated simultaneously. The decision version of MDP, the Edge Disjoint Paths problem (EDP), is proved by Karp [25] to be *NP-complete*. Thus the UFP is also NP-complete and there is no hope for finding an efficient, that is polynomial time, algorithm that solves it to optimality unless $P = NP$. Therefore the main research in the area has been focused in two different directions.

The first has been aimed at developing efficient approximation algorithms with good approximation bounds and *competitive ratios*. Often these algorithms are supposed to find a subset of requests that gives the highest total profit, such as in call admission where as many calls as possible should be routed simultaneously. This means that a call may be denied because it is expected that if it is set up it may block many other future calls. Some of these algorithms can be suitable in applications that work online, where there is no knowledge of future requests. The downside is of course that requests may be discarded even though they would be allocated in an optimal solution. In [26] Kleinberg shows that an $O(\sqrt{m})$-approximation for the EDP problem exists and in [11] Baveja and Srinivasan shows that this bound also holds for the UFP if the no-bottleneck constraint holds, that is that the highest requested capacity is at most as high as the smallest capacity of an arc.

The second direction of research has been to use branch and bound algorithms to search for an optimal solution. Since the problem is NP-complete the search space is far too large to be searched exhaustively for anything but very small instances. The researchers have therefore focused on using methods such as *column generation*, *cutting planes*, *LP relaxation* and heuristics as well as other techniques and combinations of these to reduce the search space. This has resulted in a large increase of the size of the problem instances that can be efficiently solved to optimality.

To see the combinatorial explosion that makes an exhaustive search infeasible we can first consider the problem of selecting which requests that should be allocated. If we have r requests, each request can either be in the solution or not, thus we have 2^r possible subsets of requests that should be considered. Then we also have the problem

of finding which paths that should be used. A simple upper bound for all possible paths of a single request starting at a specific node would be to consider the maximum degree δ of an undirected graph and the length of the longest path that can be found. The maximum degree is the highest number of arcs connected to a node in the graph. At each node there will be a branching step for the next hop that have up to $\delta - 1$ paths to choose from. The longest path can at most visit all nodes once. Thus for a graph with n nodes and a maximum degree of δ an upper bound is $O((\delta - 1)^n)$. Next, these paths should be combined for all the requests. With the simplification that all active requests can be allocated independently the combination of all the paths of r_a requests is $O(((\delta - 1)^n)^{r_a})$=$O((\delta - 1)^{nr_a})$. Then we should also consider that there are 2^r different subsets of requests. Finally we can mention that δ is higher or equal to three for all graphs that are more complicated than a line or ring. Thus for a problem instance with only a couple of tens of nodes and requests it is easy to see that it would be infeasible to use a naive searching algorithm.

2.3 Previous work

In this section we give a sample of the previous work in the area. It covers both the directions of research that was previously mentioned and shows some of the results that have been achieved. Since the amount of research in this area is vast this survey is in no way extensive. The mentioned sources investigate problems that are the same or very similar to our interconnect allocation problem. However, for many of them there are small differences that make the algorithms they use unsuitable for our problem. Much of the previous work is based on linear programming and other optimization techniques. Describing these is out of the scope of this thesis, instead we refer to [13] for a good introduction to the area.

2.3.1 Maximum allocated capacity

In [27] Kolman and Scheideler investigates online and offline algorithms for the UFP with the objective to maximize the requested capacity of the allocated requests. They introduce a new graph parameter called the *flow number* F that is a fundamental part of their analysis. A graph with better communication properties has a smaller flow number, for example a line has $F = \theta(n)$ and a mesh has $F = \theta(\sqrt{n})$. A simple greedy algorithm that has an approximation ratio of $O(F)$ in offline settings with directed graphs is given and they show that this is essentially the best possible without any further constraints of the problem. For graphs where all arcs have a capacity of at least $logF$ they give another greedy algorithm with approximation ratio $O(logF)$. A third greedy algorithm gives a $\sqrt{m}$-approximation for graphs without the no-bottleneck constraint. In the online setting two greedy algorithms, one where allocated paths can be canceled and one where they can not, are given with competitive ratios of $O(F)$.

2.3.2 Weighted UFP

Guruswami et al. investigates Edge-Disjoint Paths and several related problems including UFP with profits [23]. They prove that the EDP on directed graphs is NP-hard to approximate within a factor of $m^{1/2-\epsilon}$ for any $\epsilon > 0$ where m is the number of arcs in the graph. In UFP with profits, each request is assigned a profit and the objective is to maximize the profit of the allocated request set. Since the UFP is a generalization of EDP approximation algorithms for UFP can only be at most as good as for EDP. They also give a simple greedy offline algorithm that iteratively finds the shortest paths for the requests and allocates them one by one by taking the request that use the least total capacity first. In section 3.3 this algorithm will be further explained and evaluated.

Baveja and Srinivasan [11] studies the UFP where all requests have a weight, similarly to the profit in [7]. The objective is to maximize the total weight of all accepted requests. They formulate the problem as an integer programming (IP) multicommodity flow problem and relax the integer constraints so that it is possible to allocate only a fraction of the requested capacity of a request. This gives an LP-relaxation of the problem that can be solved in polynomial time. By using randomization the fractional solution of the LP-relaxation is rounded to an integer solution. A number of different versions of the problem is studied and several bounds are given for different graph types. For the UFP on general graphs an $O(\sqrt{m})$ approximation is given as well as an $O(d)$ approximation where d is the length of the longest flow path between two nodes.

Azar and Regev gives in [7, 8] a number of strongly polynomial algorithms for the UFP with profits. The algorithms are simple and are based on splitting the requests in two sets, one with large and one with small requested capacities. Then the sets are sorted in a certain order and requests are allocated one by one as long as they fulfill certain constraints. The set that gives the highest profit is chosen as the final solution. The algorithms address the classical UFP (where the no-bottleneck constraint apply, i.e. no requested capacity exceeds any arc capacity), the extended UFP (no limits on requested or arc capacities) and the bounded UFP (where requested capacity is at most 1/K of the minimum arc capacity for some K). For the classical UFP an $O(\sqrt{m})$ approximation is given which is the same as for the previously best known approximation algorithms. For the other two variants similar results that equal or exceed the previously best known approximations are shown.

Another approximation algorithm for the weighted UFP in undirected graphs is presented by Chekuri et al. in [12]. They model the problem as an ILP and the algorithm they propose is based on the solution of an LP relaxation of that ILP. In this solution the flow of each commodity may be split in several paths. For each commodity up to one of these paths is randomly selected for allocation. The next step is to sort the commodities in order of descending requested capacity. Finally, in this order, the commodities are allocated one by one unless the allocation of a commodity would result in a violation of a capacity constraint. In this case the violating commodity is simply discarded. This of course means that commodities that are allocated in an optimal solution may be discarded. However, assuming the no-bottleneck constraint applies, the algorithm still obtains an $O(\delta\alpha^{-1}log^2 n)$ approximation ratio, where δ is the maximum degree (the

maximum number of arcs out of a node), α is the expansion (which is related to the connectivity of the graph) and n is the number of vertices in the graph. The authors also investigate a number of variations of the problem with different constraints such as uniform capacities and bounded capacity requests.

2.3.3 Maximum number of requests

Awerbuch et al. [6] prove several lower bounds on the competitive ratio of the online admission control and routing problem where the goal is to accept as many requests as possible. For a $(n+1) * (n+1)$ mesh with any deterministic online algorithm they show a lower bound on the competitive ratio of $\Omega(\sqrt{n})$. They also show that the lower bound for any greedy online admission control and routing algorithm in the case of a general topology with n nodes is $\Omega(n)$. The authors continue the development of the algorithms first presented in [5] focusing on heuristics to improve the performance in practice, even though it may weaken the theoretical competitive ratio. In addition they present algorithms for the case when the requests are not permanent but have a finite duration.

2.3.4 Minimum Cost Multicommodity Flow

Barnhart et al. investigates the problem of Integer Multicommodity Flow Problems [10]. In their formulation they have a cost for assigning each commodity to an arc. The goal is to find the flow that minimizes the cost. The algorithm they design is a variant of branch-and-price, a common algorithm for solving large Integer Linear Programs (ILP) through a combination of column generation and branching. In [9] Barnhard et al. further develops this algorithm to include cuts. One problem with the earlier branch-and-price algorithm arises from the symmetry inherent in the problem. If one commodity, that in the linear programming (LP) solution is split among several paths, is forced to one path the result can be that another commodity is split instead. This is avoided in the algorithm by adding specialized cutting planes to the formulation and re-optimizing. This forces the LP to choose which of the commodities that should be rerouted and according to the authors this is more efficient than leaving that choice to the branching part of the algorithm. This algorithm is further described in chapter 3.6.

Peinhardt [30] investigates a similar problem for routing in fiber optic networks. He formulates an integer multicommodity flow problem with node capacities where flows can be split in integral units. The objective is to maximize the flow sent while at the same time minimizing a cost. A parameter is used to balance the importance of the two objectives. He gives a few different LP formulations of the problem based on different variables as well as a nonlinear program formulation. A subgradient method, using the non-LP formulation, and a branch-and-cut method, using one of the LP formulations, are investigated and compared. The branch-and-cut method proves to perform better on the set of graphs that are used. Peinhardt also gives a nice survey over a variety of multicommodity flow problems and related work in the area.

2.3.5 Other variations

Suri et.al. [33] propose a new algorithm for routing bandwidth-guaranteed flows dynamically in ISP (Internet Service Provider) networks by using traffic profiles. A traffic profile is in this case the expected amount of requested bandwidth between each pair of source and destination nodes. Based on these requests they first solve a fractional multicommodity flow by using an LP-solver. This pre-processing step produces a reduced graph for each profile including which arcs it can be allocated on and how much capacity it may use. Guided by this, requests can either be rejected or allocated online by a simple shortest path algorithm depending on whether its profile has enough capacity left. Computationally this method is fairly cheap, the expensive part is to solve the LP but still this is much cheaper than many other algorithms using branch-and-bound techniques or other more advanced methods. This profile based method is most suitable when there are many small requests for each source-destination pair and when the traffic patterns are predictable or known in advance.

In [21] Gendron et al. survey a number of models and algorithms for multicommodity capacitated network design problems. In this problem there is not only a cost for routing requests but there is also the possibility to install additional facilities on the arcs, installing new facilities can increase the capacity but comes with a cost as well. Requests have a requested capacity just like in the UFP, however the flows can be split. The objective is to find an optimal routing and installation of facilities that satisfies all requests while minimizing the cost. Thus this is more of a network design problem than a pure allocation problem. They mainly survey simplex-based cutting plane and lagrangian relaxation approaches and conclude that to solve difficult and large-scale instances a combination of these methods as well as heuristics will probably yield the best results. A similar network design problem is investigated by Saniee and Bienstock in [31]. They formulate a mixed integer program that is solved by a combination of pre-processing heuristics, cutting-planes, branching and rounding.

3. Algorithms

In this chapter we present a number of algorithms that have been evaluated in this thesis. The algorithms are described at a high level to give an understanding of how they work without going into implementation details. For each algorithm some theoretical reflections of what can be expected from it is also given. The algorithms described in this chapter are summarized in table 3.1 and computational performances are given in chapter 5. However, we shall first define the problem model and state simplifications that have been made compared to the RBS interconnect allocation problem.

Table 3.1: Summary of the algorithms described in this chapter.

	Online	Greedy	Optimal
SSP	Yes	Yes	No
MSPF	No	Yes	No
MCPF	No	Yes	No
DIP-Cut	No	No	Yes*
DIP-Price	No	No	Yes*
BPC	No	No	Yes*

* Provided an answer is returned before the time limit.

3.1 Model

Let $G = \{V, A\}$ be a graph with $n = |V|$ nodes and $m = |A|$ directed arcs. Each arc a in A represents a contiguous block of time slots on an actual link in the RBS network and has a corresponding capacity $c(a)$. Let R be a set of requests and for each request r let $s(r)$ denote the source node, $d(r)$ denote the destination node and $u(r)$ denote the requested capacity of the request. In a real RBS environment each requests consists of one downlink request from the REC to the RE and one uplink request from the RE to the REC. Any of these two requests can of course not exist without the other since then communication would only be in one direction. However, we have made a simplification and ignored this rule so that all the requests are independent of each other.

3.2 Simple Shortest Path

The Simple Shortest Path (SSP) algorithm is the simplest algorithm imaginable for solving the above described problem. It is a greedy online algorithm, meaning that it considers each request in isolation and once a path has been allocated for a request it will never be changed. The requests are handled in the order they arrive. The algorithm is described in the pseudocode and text below.

Algorithm 3.2.1: SHORTEST_PATH(G, R)

$allocated \leftarrow \emptyset$
for each $r \in R$
  do if $\exists \text{sp}(r, allocated, G)$
  then $allocated \leftarrow allocated + \{r, \text{sp}(r, allocated, G)\}$
return $(allocated)$

Here $\text{sp}(r, allocated, G)$ gives the shortest path for r in G with remaining capacity on all arcs, with regard to the allocations in *allocated*, greater or equal to the requested capacity $u(r)$. For each request the SSP algorithm allocates the shortest path available with sufficient remaining capacity. If no such path is found then the request is discarded. Of course this can give arbitrary bad results in the worst case. Suri et al. describes the following example [33]. Consider a request set of $n+1$ requests, all requests have separate source nodes $S0$ to Sn but a common destination node D. The first request, from $S0$, has a requested capacity of n and all other requests have requested capacities of 1. If this request set and the graph in figure 3.1 is given to SSP the first request would be allocated on $S0 -> C -> D$ which would block all other requests. An optimal solution would allocate the first request on the longer path and then all other requests could be allocated as well[33].

On the other hand SSP is expected to be fast, for each request only a shortest path algorithm is performed and this can be done quickly using well known algorithms. In the case where the length of each arc is one a simple breadth first search is sufficient to find the shortest path. The only complication is the need to consider the capacity constraints of the arcs. This can easily be done by preprocessing the graph to only include arcs with sufficient capacity to carry the request or by modifying the breadth first algorithm to perform the check during the search. In either way, a breadth first search algorithm have a time complexity of $O(n + m)$ since in the worst case it must visit each node and each arc once. The total time complexity of SSP is thus $O(|R|(n + m))$.

An offline variation of SSP could be to sort all the requests by their requested capacity before SSP is run. If requests with small requested capacity comes first more requests can be expected to be allocated while if requests with high requested capacity comes first the small requests can hopefully be allocated on the remaining arc capacities in the end. These offline variations have not been implemented and evaluated in this thesis.

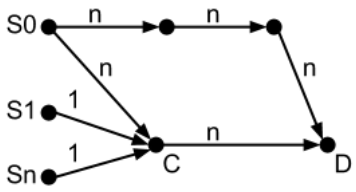


Figure 3.1: A concentrator topology showing the non-optimality of SSP [33].

3.3 Minimum Shortest Path First and Minimum Capacity Path First

A variation of the shortest path algorithm is the Minimum Shortest Path First (MSPF) algorithm. This greedy algorithm requires information about all requests at once and is therefore an offline algorithm.

Algorithm 3.3.1: MINIMUM_SHORTEST_PATH_FIRST(G, R)

$allocated \leftarrow \emptyset$
repeat
  $P \leftarrow \emptyset$
  for each $r \in R$
    do if $\exists \text{sp}(r, allocated, G)$
    then $P \leftarrow P + \{r, \text{sp}(r, allocated, G)\}$
    else $R \leftarrow R - r$
  sort P according to length
  for each $p \in P$
    do if p can be allocated
    then $\begin{cases} allocated \leftarrow allocated + p \\ R \leftarrow R - r \end{cases}$
    else break
until $R = \emptyset$
return $(allocated)$

The basic building block is again the breadth first search but here the shortest path of all requests are computed at the same time. The results are then sorted according to length, shortest lengths first, and the requests are allocated until the first request that cannot be allocated with the preceding requests is encountered. This happens when the remaining capacity of an arc in the previously calculated path is lower than the requested capacity of the request. At that point the algorithm starts over and recomputes the shortest paths of all the remaining requests, taking into account the existing allocations. If no path can be found for a request it is discarded.

MSPF would still not find the optimal solution in the example of figure 3.1 and it

is not guaranteed to be better than SSP in any specific case. However, it might be expected that short paths will have a smaller impact on the overall capacity of the graph and that this would result in that more requests can be allocated. This depend on the requested capacity of the requests. If for example the requests with short paths have high requested capacity while those with longer paths have small requested capacities the impact of allocating the former may be high.

Instead of only sorting according to length, another variation of MSPF is to sort by the total arc capacity that would be used if a path is allocated. We call this algorithm the Minimum Capacity Path First (MCPF) algorithm and it is also described in [23]. The total used capacity of a path equals its requested capacity multiplied with its length. If this measure is used when sorting it can be expected that more requests can be allocated since the capacity of the graph is decreased as slowly as possible. As a result requests with high requested capacity and long paths will be less prioritized.

In the worst case all the shortest paths must be recalculated after each allocated request and in this case the shortest path algorithm will be run $|R|(|R|+1)/2$ times. Thus the time complexity of MSPF and MCPF is $O(|R|^2(n+m))$.

3.4 Naive Branch and Bound

If a computer with unlimited resources was available it would be possible to use a naive branch and bound algorithm that more or less finds and tries all possible combinations of request allocations. A simple bounding procedure is to use the value of the first feasible solution as an upper bound for the rest of the search. If a branch can only lead to a solution with a lower bound higher than the upper bound this branch can be pruned. Without going into further details about the implementation we can mention that we designed and implemented two depth first versions of a naive branch and bound algorithm. One version tries to maximize the number of requests that can be allocated and the other one tries to maximize the requested capacity of the allocated requests. The purposes of these are to see for how large graphs and request sets they could be feasible to use. However, as mentioned in section 2.2 it is easy to see that a problem instance larger than very small should be infeasible.

3.5 The DIP Framework

Decomposition in Integer Programming (DIP) is an open source framework with the intention to decrease the effort needed to implement decomposition based branch and bound algorithms for solving integer programs[17]. The framework takes care of much of the details of the algorithms but still allows the user to customize various parts of the algorithms such as the branching or solving of the subproblems. The framework allows one to easily change between different solving methods such as branch-and-cut and branch-and-price-and-cut. In this thesis these algorithms have been evaluated and will be explained in the following two sections.

Using DIP one can significantly shorten the time to develop a first working prototype of an algorithm while still allowing considerate customizations and optimizations when needed. DIP is built upon and part of the Computational Infrastructure for Operations Research (COIN-OR) project, a project that aims for providing open source libraries and frameworks for developing algorithms in the area of operations research. COIN-OR includes for instance CLP, a linear program solver, CGL a cut generator library as well as common utility functions and data structures. For a deeper understanding of DIP and the theory behind see [19] and [18].

3.5.1 Branch and Cut

Branch and cut is a variation of branch and bound applied to integer programming. There are several variations of branch-and-cut but commonly an LP relaxation is solved at each node in the search tree, giving a lower bound on the value of the optimal solution. In order to obtain a stronger bound the relaxation is strengthened by finding inequalities, called cuts, that should be satisfied by an integer solution but are violated by the optimal solution of the LP relaxation. Generating cuts can be done in several ways which is described in more detail in [28]. One approach uses a problem specific separator routine, when a solution to the relaxation has been found the separator routine is applied in order to identify a violated inequality. This can be used when the number of constraints in a problem formulation is exponential in the problem size. It is then possible to leave out many of the constraints and use the separator routine to dynamically generate only those that are violated. An often given example of this is the Traveling Salesperson Problem (TSP) where the goal is to find the shortest tour that visits all nodes exactly once in a weighted graph. In the integer program formulation of TSP one have the constraint that for each node only two edges are used in a solution. This is not enough so the integer formulation also includes a number of Subtour Elimination Constraints (SEC), these constraints enforce that there must be only one tour in the graph. The problem is that we need one such constraint for each subset of nodes and we therefore have exponentially many constraints. The solution is to leave out the SECs and after solving the LP relaxation identify the offending subtours and add only those SECs that prevents them.

It is also possible to use branch-and-cut without a problem specific separator routine. Using general families of inequalities such as Gomory cuts and Knapsack cuts one can improve the bound in general cases and this is the approach used in our branch-and-

bound algorithm, DIP-cut.

Algorithm 3.5.1: DIP-CUT(ILP)

$n \leftarrow 1$
$p_o \leftarrow$ INITIALFORMULATION(ILP)
$P \leftarrow \{p_0\}$
$\bar{z} \leftarrow max\_value$
$z_0 \leftarrow min\_value$
while $P \neq \emptyset$
do
- $p_i \leftarrow$ select problem from P
- $P \leftarrow P - p_i$
- **repeat**
  - $\{x_i, z_i\} \leftarrow$ SOLVE(p_i)
  - $p_i \leftarrow p_i +$ FINDCUTS(p_i, x_i)
- **until** no more cuts found
- **if** $z_i < \bar{z}$
- **then**
  - **if** $x_i \in Z^l$
  - **then**
    - $\bar{z} \leftarrow z_i$
    - $x_b \leftarrow x_i$
    - $P \leftarrow \{p_j \in P | z_j < \bar{z}\}$
  - **else for each** suitable branch variable $v \in p_i$
  - **do**
    - $p_{n+1} \leftarrow p_i + \{v = 0\}, z_{n+1} \leftarrow z_i$
    - $p_{n+2} \leftarrow p_i + \{v = 1\}, z_{n+2} \leftarrow z_i$
    - $P \leftarrow P + p_{n+1} + p_{n+2}$
    - $n \leftarrow n + 2$

return (x_b)

Here $\bar{z}$ is the value of the objective function in the currently best solution x_b and therefore an upper bound on the optimal value, z_j is the optimal value of the solution x_j to the LP relaxation of problem j. The basic algorithm selects a problem p_i to solve from the pool of potential nodes. It solves the linear relaxation of p_i and tries to identify one or more violated cuts. If any cuts are found, they are added to the formulation and the process is repeated until no more cuts can be found. If the last solution is integer and the value is less than the previous best value, a better solution has been found. In this case all potential nodes with lower bound higher than z_i can be discarded. If the value is less but the solution is fractional the algorithm branches on some variables, forcing them to 0 and 1. The resulting problems are added to the pool of problems and the process is repeated until there are no problems left or a timeout is reached.

The algorithm is applied to the ILP formulation of the UFP instance.

$$\text{minimize} \sum_{r \in R} \sum_{a \in A} x_{\text{var}(a,r)} + \sum_{r \in R} (u(r) + C) d_r$$

subject to

$$\begin{array}{ll} \sum_{i \in \text{en}(k,r)} x_i - \sum_{j \in \text{ex}(k,r)} x_j + f(k,r) d_r = f(k,r), & \forall k \in V, \forall r \in R \\ \sum_{r \in R} u(r) x_{\text{var}(a,r)} \leq c(a), & \forall a \in A \\ x_{\text{var}(a,r)} \in [0,1], & \forall a \in A, \forall r \in R \end{array}$$

where

$$f(k,r) = \begin{cases} -1, & \text{if } k = s(r) \\ 1, & \text{if } k = d(r) \\ 0, & \text{otherwise} \end{cases}$$

Each variable stand for a combination of arc and request. They represent the decision to allocate the request on the specific arc. In addition to the arcs present in the graph, one dummy arc is added for each request, going from its source to its destination with unlimited capacity, represented in the formulation as the variable d_r. The dummy arcs represents the choice to reject a request, and enable the algorithm to return a solution even in the case when not all requests can be allocated simultaneously. These dummy arcs are given a high cost C so that the algorithm only use these arcs if there is no better solution. In order to prioritize requests with higher requested capacity, $u(r)$ is also added to the cost of a dummy arc. $\text{en}(k,r)$ and $\text{ex}(k,r)$ denote the variables that represent allocating request r to the arcs entering and exiting node k. $\text{var}(a,r)$ denotes the variables that represent using arc a for request r.

While this algorithm does significantly more work than the SSP or MSPF, it should still be a considerable improvement over the simple branch-and-bound since the LP relaxation gives a much better bound. However, the amount of work depends highly on the structure of the problem, and in a problem that requires a lot of branching this could still quickly grow beyond what can be efficiently solved. By limiting the time available for the algorithm it is possible to find solutions in some cases, even though they are not proven optimal. This may happen if there are still nodes left to explore in the search tree when the time limit is reached, but an integral solution has been found during the search. In these cases one can use the lower bounds of the remaining nodes to give a bound on how far from optimal the solution is. In our case we set the time limit to 30 minutes.

3.5.2 Branch and Price and Cut

Another approach to solving large integer programs is branch-and-price. Branch-and-price is similar to branch-and-cut but instead of generating inequalities (rows in the ILP formulation) one generates nonzero variables (columns in the ILP formulation) [10]. The main observation in branch-and-price is that in any solution, most of the variables in

the ILP formulation is zero, and will not affect any of the constraints. This means that we can leave those variables at zero and still get the same result. It is not obvious how to come up with the variables that should be part of the solution. The variables are identified by finding negative reduced costs, calculated with help of the duals to the master problem. The identified variables can then be added to the basis and the result reoptimized.

DIP supports this form of algorithm in the integrated form called branch-price-cut. Here pricing and cutting are used together in order to obtain stronger bounds. The pricing is based on a decomposition where some of the constraints are used as subproblems, in our case the conservation of flow constraints. The rest of the constraints form the master problem. The algorithm first solves the master problem obtaining both a solution as well as the values of the dual variables. It then alternates between solving the subproblems using the dual information for improving variables and generating cuts from the solution. If neither of the approaches gives any new information the branching takes place. Since this is a very simple decomposition without any specialized solvers for the subproblems, it is possible that the added work will only slow down the performance compared to the branch-and-cut algorithm. But using DIP the workload to implement the branch-and-price-and-cut algorithm given a branch-and-cut implementation is so small we decided to investigate it anyway.

3.6 Branch and Price and Cut for Integer Multicommodity Flow

There exists specialized versions of branch-and-price where one do not have to know of all the variables in the formulation. This is achieved by starting with a small subset of the variables, called the restricted master problem(RMP), and generating variables that could improve the solution when needed. This is called delayed column generation. The trick lies in identifying good variables without explicitly looking at them all.

Barnhart et al. uses this approach together with cut generation in [9]. In the algorithm presented by Barnhart two alternative ILP formulations are used, the node-arc formulation and the path formulation. The description below shows the path based formulation and is adapted to our specific needs, for the original version we refer the reader to [9]. In the path formulation, each path a request may use through the network is represented as a variable. Here $P(r)$ represents all such path variables for request r, and $p(a, r)$ represents all path variables for request r using arc a. In the same way as before we add dummy arcs to ensure that the instance is feasible.

$$\text{minimize} \sum_{r \in R} \left(\sum_{i \in P(r)} p_i \right) + (u(r) + C) d_r$$

subject to

$$\begin{array}{ll} \sum_{r \in R} \sum_{i \in p(a,r)} u(r) p_i \leq c(a), & \forall a \in A \\ \left(\sum_{i \in P(r)} p_i \right) + d_r = 1, & \forall r \in R \\ p_i \in [0,1], & \forall r \in R, \forall i \in P(r) \end{array}$$

The two formulations are equivalent except that the path formulation trades fewer constraints for a larger number of variables. In the path formulation we have one variable for each path between a request's source and destination. Now we do not need the flow conservation constraint since it is implied by our use of path variables. We instead have a constraint that says that the sum of all path variables for a request is one, ensuring together with the integrality constraint that each request only uses one path.

In each node of the search tree the current RPM is first optimized and a set of duals $-\pi_a^r$ and ρ^r are obtained, corresponding to the capacity constraints and the single path constraints. From the duals the reduced cost of each column can be calculated as $\bar{c}_p^r = \sum_{a \in A}(1 + \pi_a^r) - \rho^r$. Finding the smallest reduced costs for a request r is then easy using a suitable shortest path algorithm in the network where the arcs have costs $1 + \pi_a^r$. If the cost of the path for request r is smaller than ρ^r then we have found a possibly improving path. The reduced problem is then augmented with the improving paths and the problem is re-optimized repeatedly until no more improving paths can be identified. In a branch-and-price algorithm this is the termination condition of the price stage, and if the solution is not integral the next step is to branch.

One issue to consider with branch-and-price is that some branching strategies may destroy the structure of the pricing problem, leading to more complex pricing strategies. Barnhart avoids this by branching on the variables in the arc-node formulation instead of branching on the path variables, in each branch a set of arcs are forbidden for a commodity. Which arc sets to forbid is selected by choosing the commodity with highest requested capacity among those with fractional values, and then choosing the two paths with highest fractions. The paths are followed to the node where they diverge from each other and all arcs leading out of that node is divided into two sets, each including an arc used by one of the paths. As Barnhart et al. describes in [9], this branching alone can lead to poor performance due to the symmetry inherent in the problem. If one commodity is forced to use only one path, another may be split instead. The problem is relieved by using cuts in addition to the pricing. Whenever the pricing phase terminates, an attempt to find cuts is performed. The generated cuts are of type lifted cover inequalities, LCIs, which limits the number of requests using a particular arc. A discussion of how to generate the cuts is beyond the scope of this report, for more information see [22]. The cuts are found in the arc-node formulation but can easily be translated to the path

formulation. If any cut is found the resulting problem is reoptimized. First when no more cuts can be found the branching commence.

4. Evaluation Framework

In order to evaluate algorithms we developed an evaluation framework in C++. The objectives for this application were that it should be fairly easy to implement an algorithm in it and use the framework to test the algorithm. It should give measurements of how well an algorithm performs and it should be possible to let the algorithms be tested on a large number of different problem instances. In this chapter we first describe the functionality of the evaluation framework, then we give an overview of the design of it and finally we describe the test scenarios that we have used to evaluate the algorithms.

4.1 Functionality

The evaluation framework is controlled by an ini-file that allows a user to set up a range of different test scenarios for the implemented algorithms. To execute an algorithm a problem instance consisting of a graph and a request set should first be created. A graph can be acquired by reading it from an XML-file, by using a hard coded graph or by letting the program generate a graph. A request set can be generated or, if a graph is read from an XML-file, it can also be read from a matching XML-file. Then the problem instance is solved by a set of algorithms, which algorithms that should be used is also specified in the ini-file. The result of each algorithm is displayed at the command line. The amount of information displayed at the command line can be adjusted from the ini-file. This program flow is illustrated in figure 4.1.

When a graph is generated you specify how many nodes and arcs it should contain and you also specify in what range the arc capacities should be. The default behaviour when generating a graph, is to add one node at a time by randomly selecting which of the previously added nodes it should be connected to. When all nodes are added we have a tree structure. The next step is to, for all the remaining arcs, randomly select two nodes and connect them by the arc. Each arc is also given a capacity randomly chosen within the specified capacity range. The arcs are directed, but since we assume that all links are symmetrical duplex links, for each arc added, an identical arc is also added between the same nodes but in the opposite direction. Figure 4.2 shows an example graph that was generated.

It is also possible to generate a graph that is somewhat more similar to the RBS interconnect network. In this case the number of desired RECs and REs is specified. First the RECs and any extra arcs are added just like before. Next, the REs are added

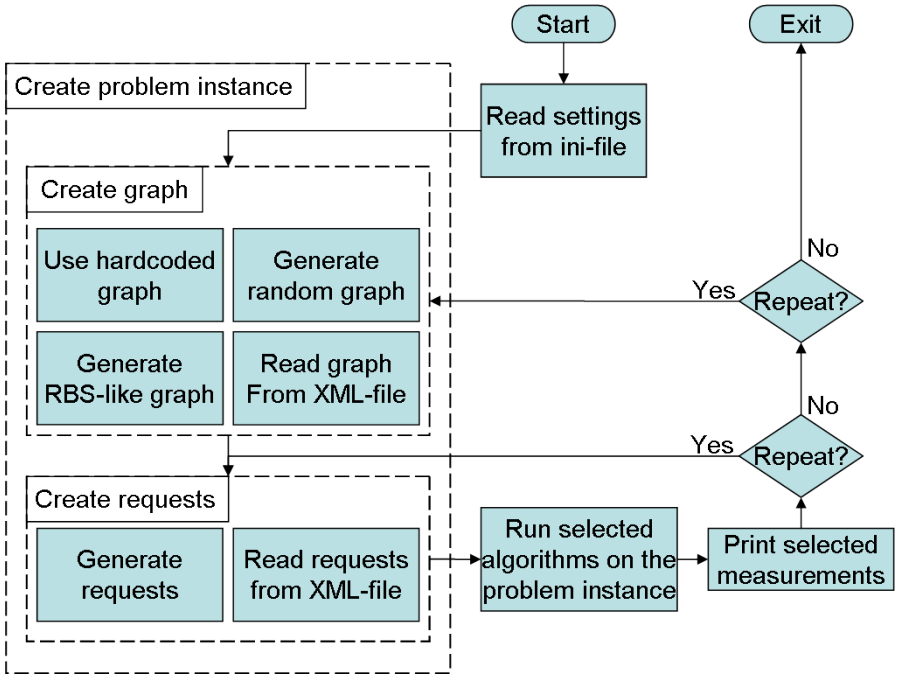


Figure 4.1: The program flow of the evaluation framework.

one by one by randomly selecting a previously added REC or RE. If the selected node is a REC an arc is just added between them. If on the other hand the selected node is an RE, a chain of REs is created out from the connected REC.

A request set is generated by, for the specified number of requests, randomly selecting two nodes where one becomes the source node and one the destination node. The requested capacity is also randomly chosen within a specified range. In the case of the graphs with RECs and REs all requests are set up between a REC-node and an RE-node. To simulate that paths in both directions should be allocated each generated request can be mirrored so that there is a pair of identical requests with the only difference that they have swapped the source and destination nodes. However, when the algorithms are allocating requests there is no rule that forces either both or none of the paths in a pair to be allocated.

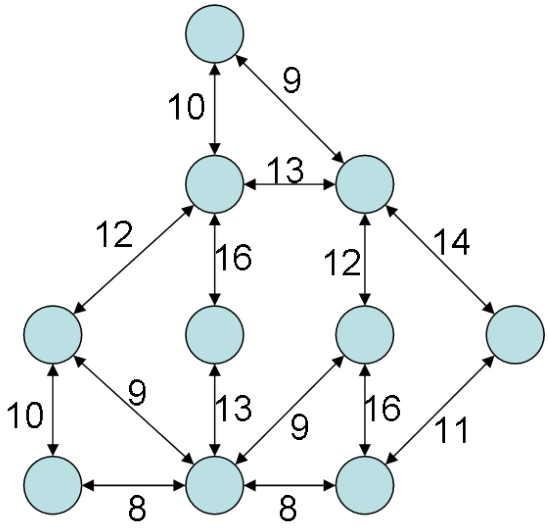


Figure 4.2: An example of a generated graph with 10 nodes and 15 arcs. The numbers specify the capacities of the arcs. The arcs are directed but doubled with one arc in each direction so the real number of arcs is 30.

When an algorithm has been run on a problem instance it returns data structures with information about how it has allocated the requests. If desired, this information can be displayed. The program also makes a number of measurements of each solution in order to give numerical results that can be easily compared. The measurements include:

- the time it took for the algorithm to find the solution
- the number of requests allocated
- the total allocated capacity
- the total arc capacity used
- the number of arcs used
- the average load on the arcs
- the total number of jumps
- the length of the longest path.

The most interesting numbers to look at are the number of requests and the total allocated capacity since these reflect how much that has been allocated. The other measurements are most interesting when comparing solutions that have similar results on the number of requests and total allocated capacity.

The evaluation framework can be configured to create and test many problem instances in sequence. For each graph multiple request sets to test with can be generated,

or read, and it is also possible to read and generate many different graphs. All instances are solved by the desired algorithms and the output can be used to analyse and evaluate the algorithms.

4.2 Design

The evaluation framework can roughly be divided into three different parts plus external code as shown in figure 4.3. First is the main program that handles the reading of parameters, generation or reading of graphs and requests, the execution of algorithms and the collection and printing of data. To read input parameters we have used an open source library called SimpleIni that is used to read parameters from an ini-file [2]. The XML-files that have been used are based on files supplied by Ericsson. The XML parsers for these files are also from Ericsson, but adapted to construct a model for our evaluation framework. The parsers are built using the open source library Expat [1].

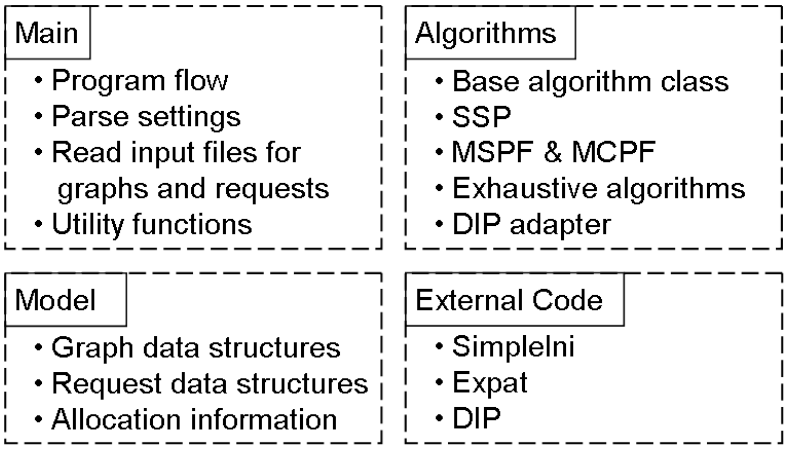


Figure 4.3: A rough division of the design of the evaluation framework.

The second part is an internal model of a network and requests along with utility functions for manipulating and viewing this model. Nodes and arcs are represented in data structures linked together with pointers. Arcs also have information about their timeslots in order to keep track of their available capacity.

The third part consists of the algorithms. This part is object oriented so that it is fairly easy to implement new algorithms and run them in the evaluation framework. New algorithms should be subclasses of a base algorithm class and implement certain methods in order to be executable in the framework. The algorithms are given one problem instance at a time and return data structures containing information about how the requests have been allocated.

4.3 Test scenarios

To evaluate the algorithms the evaluation framework has been used to create a number of scenarios, or problem instances. Since SSP, MSPF and MCPF are greedy algorithms that run in polynomial time these algorithms are capable of handling problems of much larger size compared to what is feasible for DIP-Cut. Therefore a number of test scenarios were made to compare these three algorithms separately. Then a number of test scenarios have also been used to test all algorithms, including DIP-Cut. These include both randomly generated graphs and fixed graphs with both fixed and random requests. All scenarios are described below and the results are presented in the next chapter. Of course there are many more problem settings other than those described here that could be interesting to evaluate but due to the limited time we decided not to add any more test scenarios.

For the greedy algorithms eighteen different scenarios were made. All are randomly generated graphs with random requests. For all scenarios the arc capacities are randomly chosen between eight and sixteen for each arc. The requested capacities are randomly chosen between one to eight which means that the no-bottleneck constraint holds. For each scenario a graph is first generated, then ten different request sets are generated. The mean values of the results from each scenario are used in order to get a more robust result.

The different scenarios can be divided into a number of groups. Nine different classes of graphs are generated and for each class two types of request sets are generated. The first, scenario 1 to 9, has an abundance of requests where only a fraction of all the requests can be allocated simultaneously. The second, scenario 11 to 19, has a more suitable number of requests where all or almost all requests can be allocated simultaneously. The nine graph classes can be grouped in three groups of different sizes. There is a small class with only 10 nodes, a medium class with 50 nodes and a large class with 500 nodes. For each class the number of arcs also vary in three steps. The idea is that these tests should show the differences between the algorithms when applied to different classes of problem instances. A summary of the scenarios is presented in table 4.1.

Table 4.1: Graphs in scenario 1-19 have arc capacities of 8-16. Requested capacities are 1-8.

Scenario	nodes	arcs	average degree	requests
Scenario 1 / 11	10	15	3	80 / 20
Scenario 2 / 12	10	20	4	80 / 30
Scenario 3 / 13	10	25	5	80 / 40
Scenario 4 / 14	50	75	3	1000 / 50
Scenario 5 / 15	50	100	4	1000 / 100
Scenario 6 / 16	50	250	10	1000 / 400
Scenario 7 / 17	500	750	3	10000 / 300
Scenario 8 / 18	500	1000	4	10000 / 500
Scenario 9 / 19	500	2500	10	10000 / 3000

To further investigate the performance of the greedy algorithms a number of additional tests are created where the number of nodes, the number of requests and the average degree are systematically varied according to table 4.2. All are randomly generated graphs with random requests. For all the combinations the arc capacities are randomly chosen between eight and sixteen for each arc. The requested capacities are randomly chosen between one to eight which means that the no-bottleneck constraint holds. For each combination a graph is first generated, then five different request sets are generated. The mean values of the results from each combination are used in order to get a more robust result.

Table 4.2: Graphs in the performance scenarios have arc capacities of 8-16. Requested capacities are 1-8.

Nodes	10	50	100	150	200	250	300	350	400	450	500			
Requests	10	20	40	80	160	320	640	1500	3000	4500	6000	7500	9000	10000
Average degree	2	3	4											

To evaluate the DIP algorithms additional scenarios are used. Again random graphs and request sets are used but the sizes of these are considerably smaller compared to those previously described. The random scenarios, 21 to 25, are summarized in table 4.3. In addition, nineteen fixed topologies, supplied by Ericsson, representing different configurations of RBS interconnect networks are used. These graphs are both used with fixed request sets from XML-files, also from Ericsson, as well as randomly generated requests. In the fixed request sets the requested capacity is one for all requests and in the generated request sets the requested capacity range from one to eight. For the fixed requests there is no point of running the scenarios more than once for each algorithm since the problem instances does not change. The generated requests in scenario 51 to 69 are generated in pairs with one request in each direction but with equal requested capacities. Table 4.4 shows the scenarios with the graphs from Ericsson. Scenario 31 to 49 correspond to the fixed request sets and scenario 51 to 69 use random requests. Even though some rows may look identical in the table there are topological differences in the graphs.

Table 4.3: Requests in 21-25 have a requested capacity of 1 to 8.

Scenario	nodes	arcs	average degree	requests
Scenario 21	5	8	3.2	30
Scenario 22	5	8	3.2	40
Scenario 23	20	32	3.2	30
Scenario 24	20	32	3.2	40
Scenario 25	40	64	3.2	50

During early tests of the DIP algorithms they were observed to have difficulties with some problem instances. In some cases they ran for a long time without finding a solution and eventually they allocated so much memory so that the program was stopped. To

Table 4.4: Graphs in scenario 31 to 69 have arc capacities set to 16. Requests in 31 to 49 have requested capacity of 1 and in 51 to 69 the requested capacity is 1 to 8.

Scenario	nodes	arcs	average degree	requests
Scenario 31 / 51	16	18	2.25	96 / 20
Scenario 32 / 52	16	18	2.25	96 / 34
Scenario 33 / 53	28	30	2.14	128 / 26
Scenario 34 / 54	16	16	2	88 / 20
Scenario 35 / 55	16	16	2	96 / 30
Scenario 36 / 56	28	16	2	128 / 30
Scenario 37 / 57	16	15	1.88	56 / 20
Scenario 38 / 58	16	15	1.88	72 / 34
Scenario 39 / 59	28	27	1.93	84 / 20
Scenario 40 / 60	24	29	2.42	128 / 30
Scenario 41 / 61	24	29	2.42	96 / 30
Scenario 42 / 62	24	29	2.42	160 / 34
Scenario 43 / 63	24	30	2.5	128 / 30
Scenario 44 / 64	24	30	2.5	128 / 30
Scenario 45 / 65	24	30	2.5	256 / 40
Scenario 46 / 66	24	31	2.58	128 / 34
Scenario 47 / 67	24	31	2.58	128 / 30
Scenario 48 / 68	24	31	2.58	256 / 40
Scenario 49 / 69	144	158	2.19	294 / 50

deal with this a time limit of 30 minutes has been added so that if the DIP algorithms are not finished within this limit they are aborted. In many, but not all, cases feasible solutions that are not proven to be optimal are found within this time limit. In the early tests the DIP-Price-and-Cut algorithm was also found to be much slower than DIP-Cut and when solutions were obtained DIP-Cut reached the same result in less time. Due to the time it takes to run all scenarios multiple times it was decided not to include DIP-Price-and-Cut in the test scenarios.

For some request sets in some scenarios the DIP algorithms causes the program to crash. Whether this is due to bugs in the DIP framework or in the adaptations between the evaluation framework and DIP we do not know. To deal with this problem, and the problem that DIP-Cut sometimes times out before it has found a feasible solution, the test scenarios that suffers from this have been rerun until ten feasible solutions have been found.

5. Results

In this chapter the results obtained from the different test scenarios described in the previous chapter are presented. The tests have been performed on 32-bit Windows workstations with 2 GB of memory and an Intel Core i5-520 CPU at 2.40 GHz. We begin with a discussion of the measurements used followed by the results grouped in sections for the different groups of scenarios.

5.1 Selection of Measurements

The measurements we have used were chosen either because they were related to the objective function of our algorithms or related to how efficiently the allocated requests are using the available capacity.

5.1.1 Allocated Capacity vs Number of Requests

In the case when not all requests can be allocated simultaneously the allocated capacity or the number of allocated requests are probably the most important measurements that can be used to compare different algorithms. Which of the measurements to use is not obvious and no general answer exists for all cases. If the number of requests is prioritized it can be expected that an RBS will have a better coverage. On the other hand, if allocated capacity is prioritized, the RBS should be able to serve more users and give higher bandwidths.

This trade off between bandwidth and coverage is a choice the operators should make and it can vary between different sites. In either way, both goals are possible to express in an algorithm for UFP with profits by a suitable selection of profits. When optimizing the number of requests the profit for each request is simply one and when optimizing for maximum allocated capacity a suitable choice of profits is the requested capacity of each request. It is also possible that a completely different prioritization can be used by giving each request an arbitrary profit depending on its importance.

5.1.2 Comparing solution efficiency

While the total allocated capacity and number of requests are obvious measurements of the solution quality the relevance of the other measurements appears when comparing solution efficiency. This is mainly applicable when different solutions have the same

number of allocated requests and amount of allocated capacity. Then, the number of used arcs, used arc capacity and total number of jumps can be seen as values for how efficiently the requests have been allocated. A lower number of used arcs means that some links in a network can be shut off and energy can be saved. A higher amount of used arc capacity means that there might be less options if more requests should be allocated in the future as well as a greater energy consumption. The number of jumps is similar but also says something about the latency that can be expected for the requests. If certain latency bounds should be followed the length of the longest path is also of interest. Which of these measurements that are most important is hard to say anything about and can probably differ in different situations. However, we found that the average number of jumps reflected the used arc capacity as well as it said something about the expected latencies and therefore we were satisfied with only presenting this measure in the results.

5.1.3 Measuring algorithm execution time

When measuring the time used by the algorithms we have deliberately chosen to take a crude wall clock measurement. This is in part because of the inherent difficulty of accurately measuring time in a complex operating system that at any time has many different tasks running but it is also a reflection of the importance the exact measurement has. As Ahuja et al. writes in [3] the running time measured depends on many factors including such as choice of language and compiler, the skill of the programmer, the test environment and especially other programs and processes competing for the same CPU.

To address this issue Ahuja et al. present an alternative to measuring CPU time. They argue that performance should instead be measured by counting representative operations. The operations should be selected so that they take a constant time, not depending on the problem size, to complete and in such a way that if one knows how many such operations that are performed the CPU time required may be estimated. This has several advantages as Ahuja et al. points out. Since representative operations that are fundamental to the algorithm are counted the measure is more independent of language, style and the workload of the computer. It is also possible to compare measurements run on different computers with different computing capabilities more easily. In the case of the greedy shortest path based algorithms we chose to count the number of iterations over the request sets as well as the number of arcs scanned as this gives a good indication of how much work that is done.

5.2 Scenario 1 - 9

Figure 5.1 shows results from scenario 1 to 9. Recall that these scenarios have an abundance of requests and only test the three greedy algorithms SSP, MSPF and MCPF. The results of MSPF and MCPF are divided by the result of SSP so that the graphs show the relative performance of the algorithms for each scenario.

Figure 5.1a shows the relation of the total number of requests that each algorithm

could allocate. Summed over all the scenarios SSP could allocate 19 percent of all requests, MSPF managed 30 percent and MCPF 34 percent. As can be seen in the graph the performance of MSPF ranges from 10 to 100 percent more requests compared to SSP while MCPF manages to allocate 10 to 200 percent more requests. Especially in scenario 4, 5, 7, and 8 MSPF and MCPF show a great improvement compared to SSP. These scenarios have a relatively large number of nodes but a low connectivity.

If we sum the requested capacity of all allocated requests we get what we call the total allocated capacity. The relative performance of the three greedy algorithms in terms of total allocated capacity is shown in figure 5.1b. As can be seen MSPF has 10 to 100 percent more and MCPF has 2 to 50 percent more than SSP. This reflects the result of the number of requests but also that MCPF chooses requests with low requested capacity.

A third measurement, see figure 5.1c, is the average number of jumps per request. This reflects the average path length of the allocated requests. Shorter paths are of course better since this reduces latency and required capacity. Compared to SSP, MCPF is 15 to 30 percent better while MSPF is 15 to 50 percent better. The greatest differences are again in scenario 4, 5, 7 and 8 which helps to explain how MSPF and MCPF can allocate more requests and more total allocated capacity.

The average execution times for the different algorithms and scenarios found in table 5.1d show that the algorithms can handle fairly large problem instances before the execution times are noticeable. Only for scenario 7 to 9 MSPF and MCPF begin to take time while SSP still finishes within seconds. To get a better view of the differences in execution time consider figure 5.1e and 5.1f. Figure 5.1e shows the number of iterations each algorithm runs. Recall that SSP run one iteration by definition while MSPF and MCPF recalculate the shortest paths of all remaining requests as soon as they encounter a request that can not be allocated with its currently calculated path. Each such recalculation is one iteration. For the small graphs about eight iterations is needed by MSPF and MCPF. This increases up to 140 and 100 iterations respectively for the largest problem instance. MSPF continuously has the highest number of iterations.

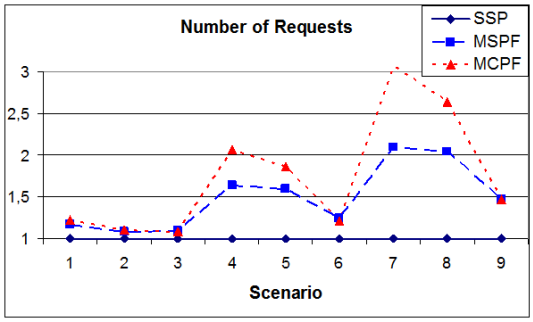
During the breadth first search for finding the shortest paths the algorithms investigates arcs to see if they should be followed or not. Figure 5.1f shows how many such investigations that are performed. Again we see that MSPF performs more such investigations than MCPF although the difference is comparatively small. Compared to SSP both the other algorithms perform between four times more investigations, for scenario 1, to almost four hundred times more for scenario 7. The billions of investigations that are made for scenario 9 explains why the execution times for even the greedy algorithms begin to grow.

5.3 Scenario 11-19

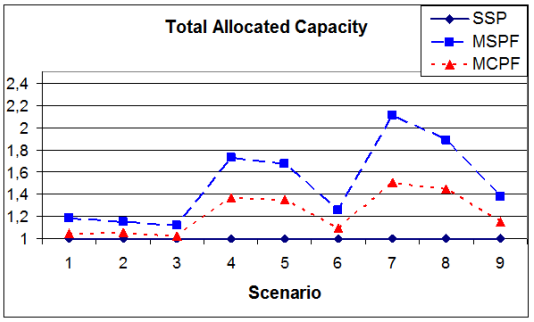
Scenario 11 to 19 are identical to scenario 1 to 9 except that the number of requests are decreased to a level where most requests can be expected to be allocated. In these settings SSP manages to allocate 92 percent of all requests summed over all scenarios, MSPF reaches 94 percent and MCPF 91 percent.

Figure 5.2a shows the number of allocated requests and as can be seen the large differences between the algorithms seen in figure 5.1a are no longer present. MSPF is still slightly better than SSP by 0.5 to 6 percent. MCPF on the other hand is now only better in two scenarios and have otherwise up to 2 percent fewer allocated requests.

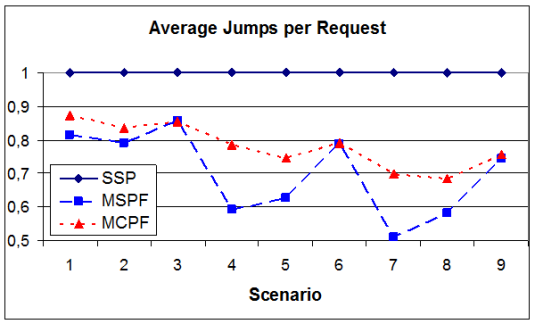
Similar results are achieved for the total allocated capacity in figure 5.2b. MSPF is a few percent better than SSP but MCPF is in most cases a few percent worse. From



(a) Relative amount of allocated requests.



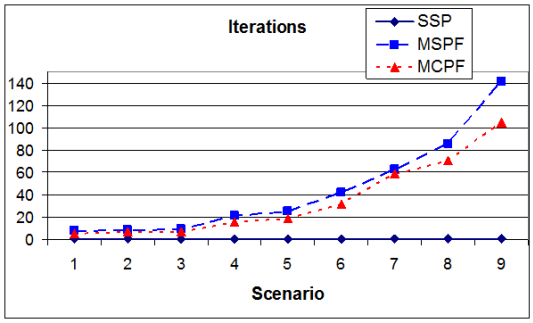
(b) Relative amount of total allocated capacity.



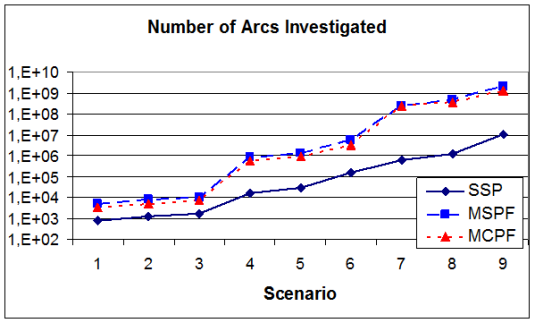
(c) Relative average path lengths of the allocated requests.

Scenario	SSP	MSPF	MCPF
1	$< 1s$	$< 1s$	$< 1s$
2			
3			
4			
5			
6		2 s	2 s
7		2 min	2 min
8		4 min	3 min
9	4 s	14 min	8 min

(d) Average execution times.



(e) Average number of iterations performed.



(f) Average number of arcs investigated.

Figure 5.1: Results from scenario 1 to 9.

figure 5.2c the path lengths of MSPF and MCPF are fairly equal, about 5 to 15 percent shorter compared to SSP. The execution times are found in figure 5.2d. Since there are less requests to process the execution times are faster compared to scenario 1 to 9 but the relative efficiency of the algorithms remain. This is also reflected by the required number of iterations for MSPF and MCPF as seen in figure 5.2e. The large difference between SSP and the other algorithms in investigated arcs, figure 5.2f, is however decreased considerably.

5.4 Scenario 21-25

In scenario 21 to 25, and the remaining scenarios, the DIP-Cut algorithm is also evaluated and compared to the three greedy algorithms. Out of all the requests in scenarios 21 to 25 SSP allocates 77 percent, MSPF 79 percent, MCPF 80 percent and DIP-Cut 84 percent. From figure 5.3 it is clear that DIP-cut gives the best solutions with more allocated requests and larger total allocated capacity. MSPF and MCPF show results similar to scenario 11 to 19 although MCPF allocates more requests than SSP, this may be because the quota of requests that can be allocated is slightly lower in scenario 11 to 19. The average number of jumps per request for DIP-Cut is a few percent lower than for SSP but not as low as for MSPF and MCPF.

For these relatively small graphs the greedy algorithms finish in less than a second, DIP however requires a lot more time as can be seen in figure 5.3d. The times displayed are average times over ten repetitions where DIP-Cut found a solution. It should be noted however that there are four different outcomes of DIP-Cut as described in the list below. The number of times each outcome occurred in the test scenarios is summarized in figure 5.3e.

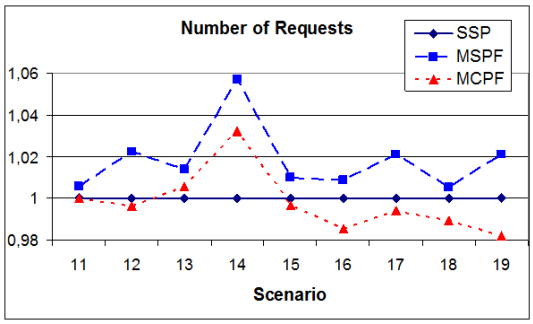
DIP-Cut outcomes

Optimal Efficient cuts can be found and almost no branching is necessary. An optimal solution is found within the time limit, often in less than a second. Sometimes the problem requires more branching and a solution is then given after a few seconds or minutes.

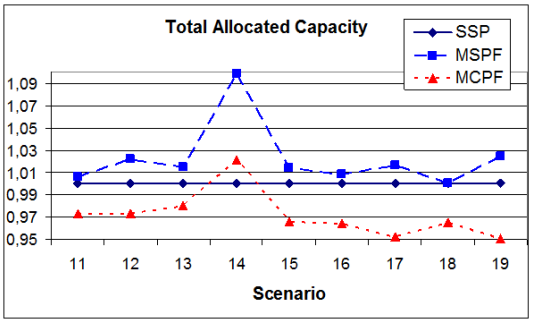
Suboptimal For some problem instances finding efficient cuts is hard and a lot of branching is needed. A feasible solution can sometimes be found but it is not proven to be optimal. When the time limit is reached the best such solution is given.

Timeout Some problem instances are too hard. The algorithm times out before a feasible solution is found. In these cases the scenario is rerun with a new request set.

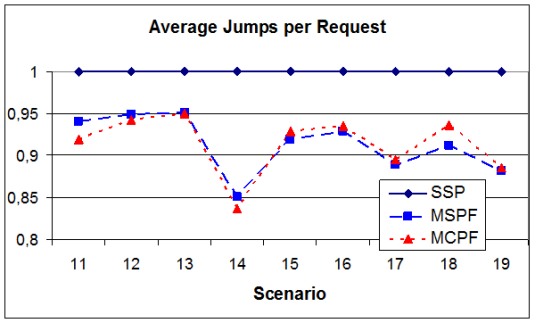
Crash Something causes the program to crash and no solution is obtained. The scenario is rerun with a new request set.



(a) Relative amount of allocated requests.



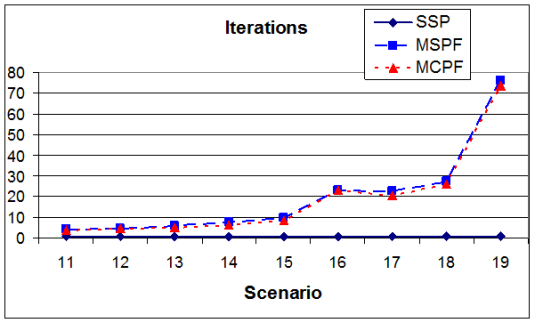
(b) Relative amount of total allocated capacity.



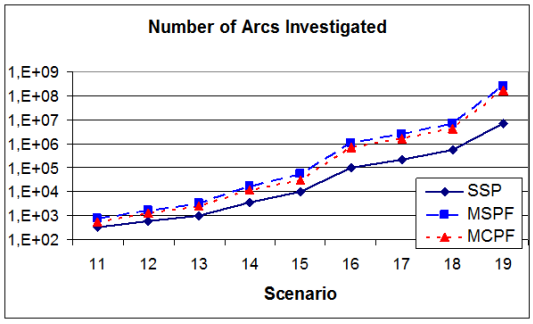
(c) Relative average path lengths of the allocated requests.

Scenario	SSP	MSPF	MCPF
11	$< 1s$	$< 1s$	$< 1s$
12			
13			
14			
15			
16			
17			
18		3 s	2 s
19	3 s	3 min	1 min

(d) Average execution times.



(e) Average number of iterations performed.

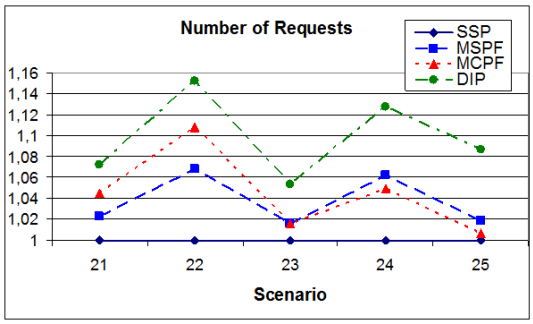


(f) Average number of arcs investigated.

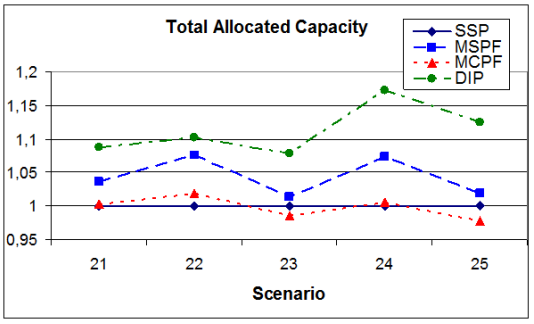
Figure 5.2: Results from scenario 11 to 19.

5.5 Scenario 31-49

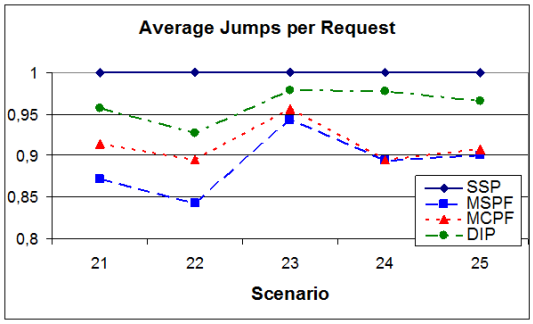
In scenario 31 to 49 fixed topologies with fixed request sets provided by Ericsson are used. The request sets are designed so that it should be possible to allocate all requests. Therefore, in all but one of the scenarios, all algorithms can allocate all the requests. In most of these scenarios the same solutions are obtained but in some scenarios there



(a) Relative amount of allocated requests.



(b) Relative amount of total allocated capacity.



(c) Relative average path lengths of the allocated requests.

Scenario	SSP, MSPF, MCPF	DIP-Cut
21		< 1 s
22		41 s
23	< 1 s	7 min
24		22 min
25		15 min

(d) Average execution times.

Scenario	Total runs	Optimal	Suboptimal	Timeout	Crash
21	10	10	0	0	0
22	14	10	0	0	4
23	11	8	2	0	1
24	15	3	7	4	1
25	15	6	4	4	1

(e) Outcomes of DIP-Cut.

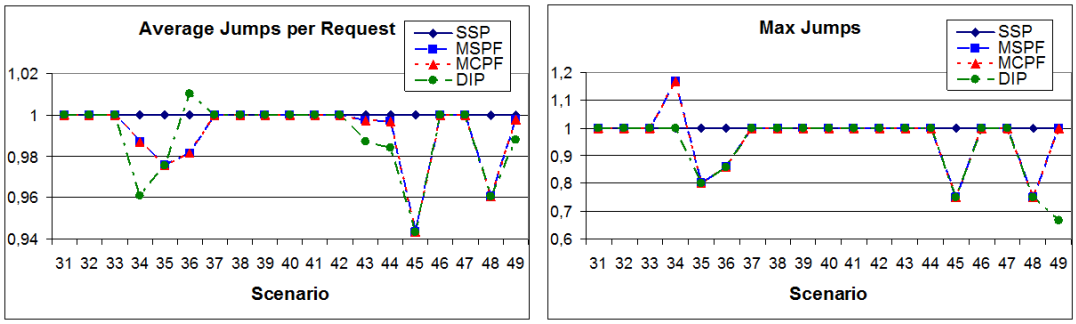
Figure 5.3: Results from scenario 21 to 25.

are differences between the solutions given by the different algorithms. DIP-Cut proves to solve all the scenarios to optimality while the solutions of the greedy algorithms are somewhat worse in some of the scenarios. Since all requests are allocated it is not very interesting to see the number of allocated requests or total allocated capacity. Instead figure 5.4 only shows results for the average number of jumps per request and the maximum length of a path.

As can be observed in 5.4a the algorithms perform equally well in most of the scenarios but there are several scenarios where SSP gives a solution with unnecessarily long paths. MSPF and MCPF show identical results for all scenarios and are often, but not always, as good as DIP-Cut. The reason for that DIP-Cut has a higher average jumps per request

in scenario 36 is that DIP-Cut manages to allocate all the 128 requests which forces it to choose somewhat longer paths. SSP, MSPF and MCPF only manages to allocate 112, 120 and 120 requests respectively. Despite this DIP-Cut still has the shortest maximum path length in the scenario as can be seen in figure 5.4b. Looking at the other scenarios as well it is clear that DIP-Cut gives the best results in all scenarios while MSPF and MCPF keep up in all but two scenarios.

For scenario 31 to 49 DIP-Cut had no problems with finding optimal solutions and it never timed out or crashed. The time required for the greedy algorithms is less than a second for all scenarios since the problem instances are not very big. DIP-Cut also manages most scenarios in less than a second except for the largest scenario which requires just above one second. However, it also requires 17 seconds to translate the problem from our model to the model used by DIP.



(a) Relative average path lengths of the allocated requests.

(b) Relative lengths of the longest paths in each solution.

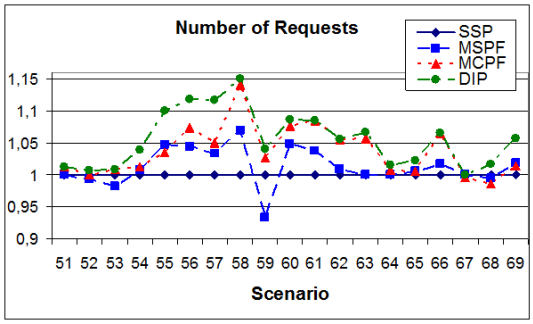
Figure 5.4: Results from scenario 31 to 49.

5.6 Scenario 51-69

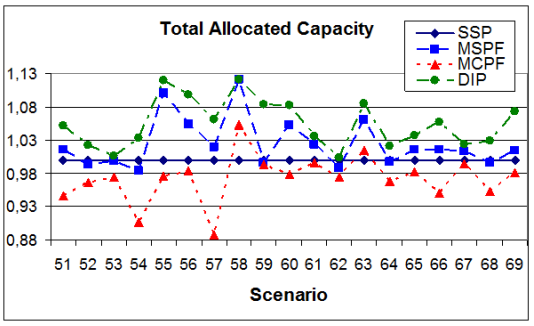
Recall that in scenario 51 to 69 the same graphs are used as in scenario 31 to 49 but this time ten request sets are randomly generated for each graph. This means that all requests may not be satisfiable at the same time but the number of requests is set so that it should be possible to allocate most requests simultaneously. The total percentages of allocated requests over all scenarios are 79.0 for SSP, 80.0 for MSPF, 81.5 for MCPF and 83.0 for DIP-Cut.

The results in figure 5.5 again show that the greedy algorithms gives solutions of less quality compared to DIP-Cut. In figure 5.5a DIP-Cut has allocated up to 15 percent more requests than SSP, MCPF follows tightly in most scenarios and MSPF is in most scenarios better than SSP. Looking at total allocated capacity in figure 5.5b MCPF lose performance even compared to SSP. MSPF is in most cases better or almost as good as SSP but in general not as good as DIP-Cut. Since DIP-Cut allocates more requests and more total allocated capacity it is not strange it does not have the lowest average number of jumps per request. It is however still lower than SSP and almost in line with MCPF.

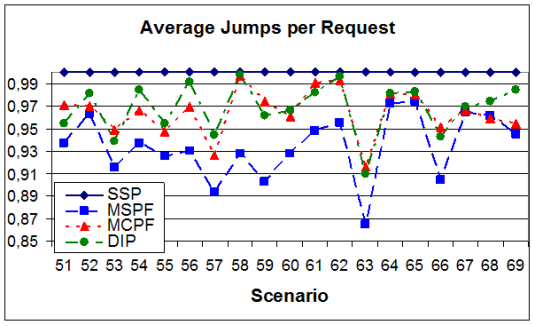
From the times and outcomes given in figures 5.5d and 5.5e we see as usual that the greedy algorithms gives their solutions in a fraction of a second. DIP-Cut is also finished within a second or two in most scenarios. The exceptions are scenario 65 and 69. In 65 one request set caused the algorithm to execute for a few minutes and for another the algorithm found no solution. The other request sets were finished within seconds. For scenario 69 DIP-Cut had two solutions not proven to be optimal which means that the algorithm was aborted after 30 minutes. There were also request sets for which no solutions were found and which caused the program to crash. The optimal solutions were typically found within a few seconds. However, in one case it took a few minutes and in another the optimal solution was found just before the algorithm was going to abort.



(a) Relative amount of allocated requests.



(b) Relative amount of total allocated capacity.



(c) Relative average path lengths of the allocated requests.

Scenario	SSP, MSPF, MCPF	DIP-Cut
51-54, 57, 59, 60, 62-64, 66-68	< 1 s	< 1 s
55, 58, 61		2 s
56		7 s
65		40 s
69		10 min

(d) Average execution times.

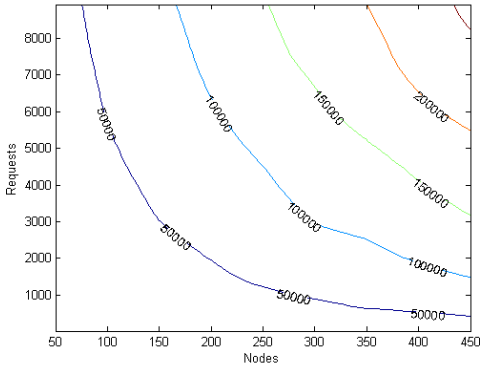
Scenario	Total runs	Optimal	Suboptimal	Timeout	Crash
51-64, 66-68	10	10	0	0	0
65	11	10	0	1	0
69	18	8	2	6	2

(e) Outcomes of DIP-Cut.

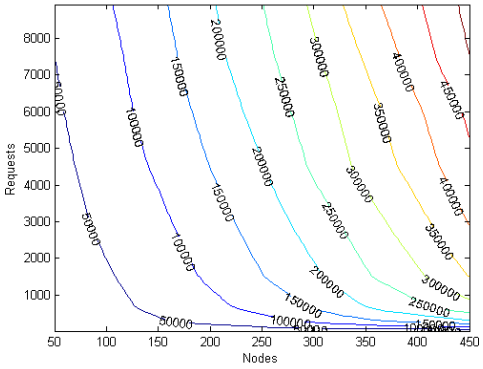
Figure 5.5: Results from scenario 51 to 69.

5.7 Performance scenarios

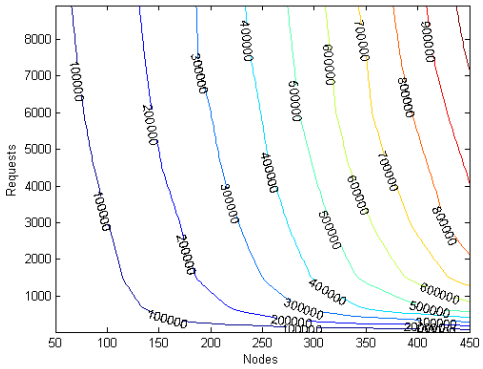
Figure 5.6 and 5.7 shows contour graphs of the number of arcs investigated by the three greedy algorithms when both the number of nodes and the number of requests are varying. The tests have been repeated on graphs with average degrees of two, three and four. Again there is a large difference between SSP and the other greedy algorithms and it is also clear that MSPF investigates more arcs than MCPF. For MSPF and MCPF it is clear that both the number of nodes as well as the number of requests have an impact on the performance. In the case of SSP the number of nodes grows more important as the degree increases. The difference between MSPF and MCPF varies a lot, for some combinations of values MSPF investigates as much as 80-90 percent more arcs than MCPF but for other combinations the difference is negligible.



(a) Investigated arcs for SSP with degree 2.

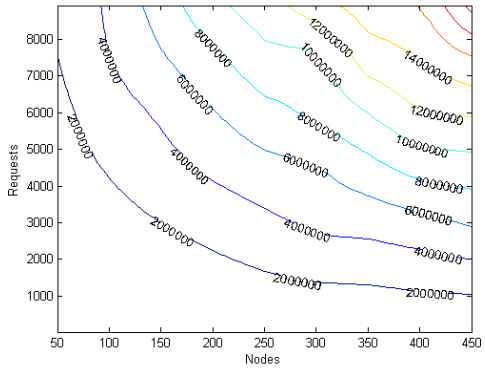


(b) Investigated arcs for SSP with degree 3.

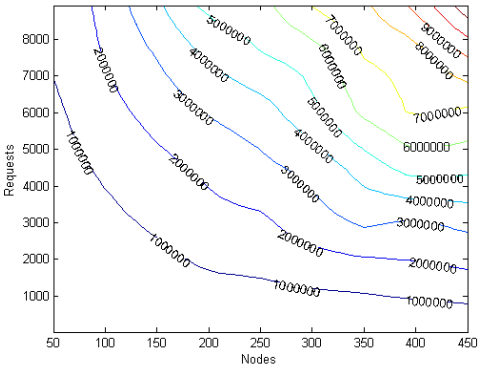


(c) Investigated arcs for SSP with degree 4.

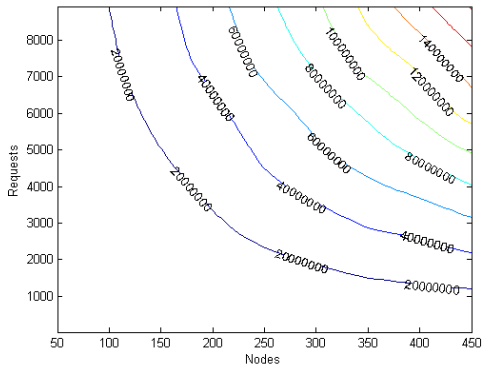
Figure 5.6: Results from the performance scenarios for SSP.



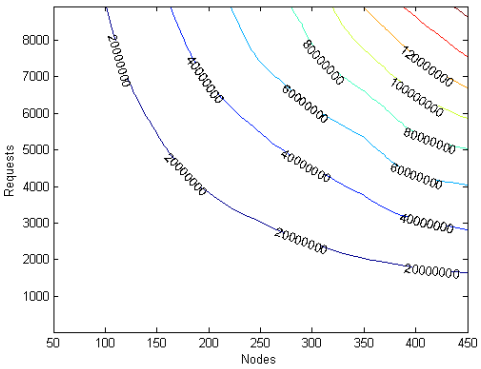
(a) Investigated arcs for MSPF with degree 2.



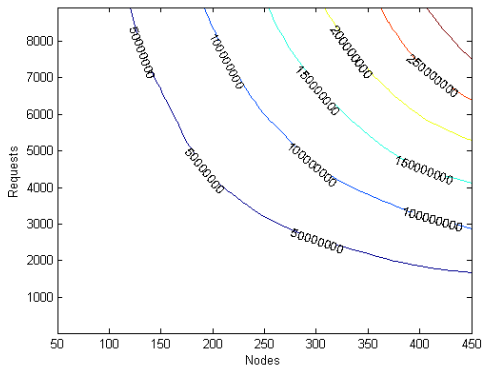
(b) Investigated arcs for MCPF with degree 2.



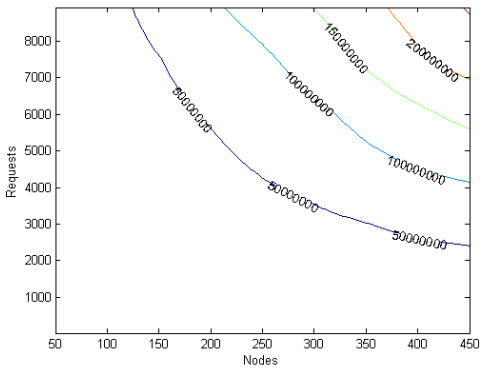
(c) Investigated arcs for MSPF with degree 3.



(d) Investigated arcs for MCPF with degree 3.



(e) Investigated arcs for MSPF with degree 4.



(f) Investigated arcs for MCPF with degree 4.

Figure 5.7: Results from the performance scenarios for MSPF and MCPF.

5.8 Exhaustive Algorithms

Finally a few notes about the performance of the exhaustive algorithms is in place. As expected the naive branching algorithms that were developed can only find solutions in a feasible time for the very smallest problem instances. For example, an instance with only ten nodes, fifteen arcs and five requests can take several minutes to solve while all the other algorithms, even DIP-Price-and-Cut, are finished in less than a second.

6. Discussion

In this chapter we will discuss what the results mean and how relevant they are to the problem we are trying to solve. We begin with discussing our evaluation framework and then continue with the impact of the structures of graphs and request sets. Then pros and cons of different algorithms will be considered. We will also describe some of the problems encountered and how the work was affected by these.

6.1 Evaluation Framework

A large part of the work was put into creating the framework for evaluating the algorithms described in chapter 4. When the algorithms were implemented the development of the framework proved well worth the effort. One key functionality of the framework is the ability to define the input data and run several different algorithms on it, comparing their results. Utilising the framework we could easily construct a few algorithms and compare their solutions on the same set of input data. We will discuss the impact of input data further below.

The framework also helped automate the task of generating random test data. The generation code could possibly be extended to help test the algorithms on more specialized graphs, with a more defined structure or generate request sets with a particular pattern or distribution. With help of the framework, testing a new algorithm and comparing it to the existing solutions is simplified. As soon as the algorithm is implemented it can be run on the exact same input data as in the earlier tests, giving a comparable result. The old algorithms can also easily be tested on new topologies and on real data when available.

Thus the framework should be useful in the future, potentially saving time and effort as new topologies and algorithms are developed and evaluated.

6.2 Impact of graph and request set structure

It is always possible to construct graphs and request sets where greedy algorithms can perform very well or very poor, as an example recall the concentrator topology in figure 3.1. By using different random graphs and request sets we have tried to see how the algorithms behave in a general case and we have also tested graphs provided by Ericsson to see how the structure of the graphs affects the performance. The greatest differences

that we have seen come from when there is an abundance of requests which the offline algorithms can take advantage of but this will be discussed later.

We have not seen any clear differences in the results between the random graphs and the fixed graphs with random requests. Instead differences can mainly be seen between the individual scenarios. When comparing scenario 4, 5, 7, and 8 with the other scenarios in 1 to 9 we see that MSPF and MCPF allocates many more requests and has a larger total allocated capacity than SSP. These graphs are relatively large with many nodes but with a low connectivity which causes SSP to allocate some requests with very long paths that block capacity that could be better utilized by other requests. MSPF and MCPF instead benefit from that they can choose requests with short paths. As seen in scenario 6 and 9 this benefit is somewhat decreased as the connectivity increases since then SSP will find shorter paths for the requests.

For some of the graphs with fixed request sets, scenario 31 to 49, there is only one way to allocate each request. In most of the others, there are multiple ways but a shortest path algorithm is sufficient to find the best allocation. However, there are instances such as scenario 36 where the greedy algorithms fail to allocate all requests and other instances such as scenario 45 where SSP allocates paths of unnecessarily long lengths. The reason for this is often that there are multiple equally short paths but if the wrong paths are chosen some arcs are filled which causes other requests to be forced to take longer paths or to be completely blocked. A different order of the requests would in many cases solve the problem and even SSP could then get the optimal solution. However, no specific ordering gives the best solution in all cases.

Figure 6.1 shows the graph of scenario 36 where none of the greedy algorithms managed to allocate all the 128 requests. There is one request with one in requested capacity from each REC to each RE and the initial order of the requests is REC1-RE1A1, REC2-RE1A1, REC3-RE1A1, REC4-RE1A1, REC1-RE1A2 and so on for each RE. Since our algorithms consider arcs in order they will always check port *1* before port *2*. The optimal way is to only let the requests from REC1 and REC2 go over the arc between REC1 and REC2. This is however not detected by the greedy algorithms which will start to allocate some of the requests from REC3 and REC4 over this arc. Eventually this causes requests from REC1 and REC2 to be blocked. Interesting to note is that if the breadth first search algorithm used by the greedy algorithms had investigated the arcs in opposite order, so that REC3 and REC4 checked the arc on port *2* first, all the greedy algorithms would have managed to allocate all the requests. This is not true in general but it shows how specific structures and characteristics of the problems, if they hold for all instances, can be utilized to improve the algorithms.

Comparing scenarios 51 to 69, the fixed graphs with random requests, shows that for some graphs the results are very similar for all algorithms while for others the results differ more. We believe that this is due to different kinds of bottlenecks that occur. Some bottlenecks limit all algorithms about the same while other bottlenecks can be used more efficiently by DIP. In scenario 57 to 59 each request only has a single possible path to be allocated on, in these cases the difference in the results comes from which of the requests that were allocated.

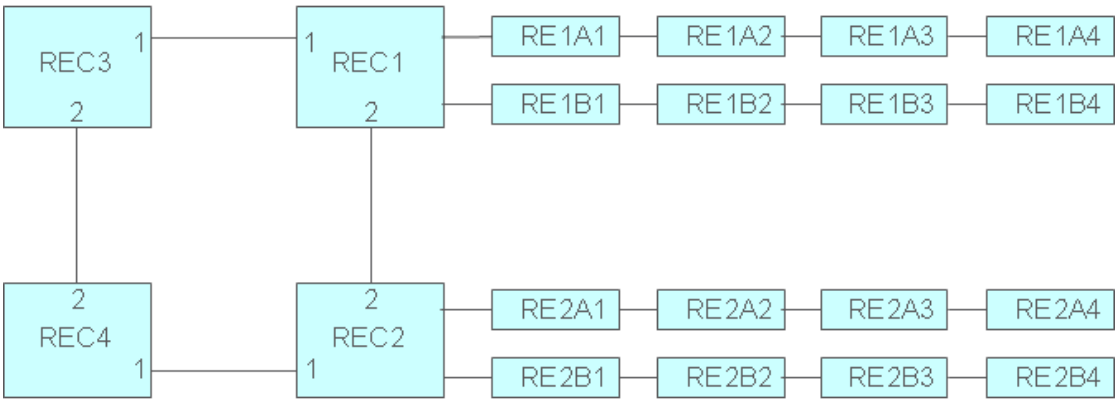


Figure 6.1: Graph used in scenario 36 where the greedy algorithms fail to allocate all requests.

As we saw in figure 6.1 the topology mainly consists of a network of RECs and then the REs are connected in chains out from one or several RECs. The random graphs that have been generated and used in the test scenarios does not have this structure. Even if the evaluation tool is capable of generating random REC networks and adding chains of REs we decided not to do so since we wanted to see how the algorithms behave in a more general setting and felt that the scenarios with the fixed graphs with random request would be sufficient to cover those test scenarios.

6.3 Impact of the number of nodes, requests and the degree

In this section the results of the performance scenarios are discussed. It is not surprising that increasing the number of requests or the size of the graph causes the algorithms to investigate more arcs.

If we look at the increase of investigated arcs between the different degrees we can see that for SSP this increase is fairly even when going from degree two to three compared to when going from three to four. For the offline algorithms the increase is instead much larger for the first than the second step. When the degree is increased then so is the number of possible paths for the requests. For a degree of two there are in general very few paths, perhaps only one or two. When the degree is increased to three the graph becomes much more connected and there will be many more possible paths. For the offline algorithms this means that a lot of requests, even though they are eventually discarded, are kept for more iterations than previously. Thus there is a drastic increase of investigated arcs for the offline algorithms when going from degree two to three. Since SSP only looks for a shortest path for each request once the increase is not equally drastic. The smaller impact of the jump from degree three to four for offline algorithms is probably related to that even though the number of paths for each request is increased this does not lead to the same increase in iterations as more requests are allocated in each iteration of the algorithm.

Looking at the results from the SSP algorithm the curves of the graphs are much steeper than those for MSPF and MCPF, at least when the number of requests is more than around one thousand. The reason for this is that as soon as the SSP algorithm has filled up a graph it will quickly find that the rest of the requests can not be allocated. Very little work will then be done for the discarded requests. The offline algorithms on the other hand may calculate the shortest paths for all the requests multiple times, even for those that later will be discarded. Thus the workload is always increasing with the number of requests even though no additional requests can be allocated.

6.4 Choice of algorithms

As we have described earlier there are many different algorithms for UFP in the literature, but not all are suitable for the concrete problem at hand. Many algorithms are designed with an infeasible amount of requests in mind, meaning that it is expected that not all of them will be allocated. The problem of deciding which request to allocate, sometimes called call admission, is therefore of great importance in these algorithms. An algorithm without call admission cannot hope to compete since it can be forced to allocate a request even in situations where it would be beneficiary to reject it [20]. While we have tested this scenario as well, one could argue that in the common RBS case all of the requests will fit. If they do not there has probably been a mistake or error that should be corrected by an operator.

Many of the approximation algorithms are shortest path algorithms with some sort of criteria for which requests to accept. Such an algorithm may only outperform a greedy algorithm by actually excluding a request. If that is not necessary then one cannot hope for an improvement. In some cases this could lead to worse performance compared to the greedy algorithms, for example when all requests fit given that a path that does not fulfil such a criteria is used. Most of the algorithms are also concerned with guaranteeing the worst case performance. This means that several of the algorithms divide the set of requests into several parts, and only allocate the best set. This is clearly unsuitable since it guarantees that some requests will be rejected, no matter what.

The algorithm chosen for the RBS interconnect allocation problem should thus be one that aims to allocate all requests. The questions are if it should be an online or offline algorithm and if it can be greedy or should use some more advanced technique such as branch-and-cut.

6.5 Online vs Offline algorithms

As we have seen in our results there is a clear advantage of using an offline algorithm over an online, at least in some scenarios. An offline algorithm can take advantage of the fact that it can consider more than one request at a time. The largest differences appeared in scenario 1 to 9 where there were many more requests than it was possible to allocate. Recall that MCPF had 200 percent more requests compared to SSP in scenario 7 and MSPF had 100 percent more allocated capacity than SSP. The reason for this is

that the offline algorithms can pick requests that are as profitable as possible while the online SSP algorithm has to allocate or reject requests one by one in the order they come no matter how profitable they are. Thus SSP allocates long paths that consumes a lot of capacity while MSPF choose the shortest paths first. MCPF minimizes the used capacity so that short paths and requests with low requested capacity are allocated first.

It is not surprising to see that MCPF in scenario 1 to 9 manages to allocate the highest number of requests since it for each allocated request leaves as much arc capacity left as possible for the following requests. Less obvious is that MSPF achieves the highest amount of allocated capacity. We believe that this is because MCPF first allocates all the requests with short paths and low requested capacity so when it reaches the requests with higher requested capacity these are forced to take longer paths. As a result the remaining capacity is quickly consumed by unnecessarily long paths leading to less allocated capacity. This is also reflected by the measure of *Average Jumps per Request* where MSPF have shorter paths than MCPF.

When there is a more balanced amount of requests the advantage of offline algorithms is much smaller and as we saw in scenario 11 to 19 all the three greedy algorithms perform more equal. There was only up to a 6 percent increase and down to a 2 percent decrease in allocated requests for MSPF and MCPF compared to SSP. For the amount of allocated capacity MSPF had up to 9 percent more and MCPF had down to 5 percent less than SSP. Thus MCPF is not even better than SSP in this setting, probably due to the same reason for which it previously lacked behind MSPF in allocated capacity. When comparing with scenario 51 to 69 we see that MCPF manages to allocate more requests than both MSPF and SSP but usually a lower amount of allocated capacity. The results from scenario 31 to 49, the fixed graphs with corresponding request sets, show that when all requests can be allocated, apart from scenario 36, there is not much difference between SSP, MSPF and MCPF. In some cases the offline algorithms manages to allocate the requests over shorter paths but there are exceptions, as in scenario 34, where the opposite holds.

As previously mentioned, in the RBS setting there will not be an abundance of requests. Instead all requests are supposed to be allocated which means that we do not expect a great advantage for an offline algorithm. The question is then if all requests will be known in advance or not and this question does not have a definite answer. The answer can depend on both how far into the future you look as well as which RBS you are looking at. In some cases, the requests will be known in advance and it would be trivial to make this information known to the algorithms. In other cases the requests arrive at different times but one could choose to collect all requests before allocating the first one. It is also possible that all the requests will not be known in advance but instead arrive one by one, in this case it may not be obvious when all requests have arrived. A possible solution is to set a time limit on how long to wait for new requests before processing them but this could in theory deteriorate to an online algorithm if the time limit is too short, while introducing too large or unnecessary delays if it is set too long. In the case of MSPF and MCPF, if requests would be processed one by one in the order they come, they would simply give the same solutions as SSP would. Thus offline algorithms that deteriorate into online algorithms when requests comes with delays may not be such a

bad alternative. Then, in the worst case they are as bad as online algorithms while in the best case they can use their offline advantages.

6.6 Greedy vs Branch-and-Cut

The results clearly show that the greedy algorithms does not reach all the way up to the optimal solutions obtained by DIP-Cut. For the random graphs in scenario 21 to 25 DIP-Cut allocates about 6 to 15 percent more requests compared to SSP and similar results hold for the allocated capacity. MSPF gives in general better results than SSP but there is still a 3 to 9 percent increase for DIP-Cut. When it comes to the average number of jumps MSPF gives shorter paths than DIP-Cut. However, DIP-Cut will always use the smallest total number of jumps possible for the allocated requests and the reason for that MSPF is better in this measure is that it has allocated less requests and capacity which then could use shorter paths. For scenario 51 to 69, the fixed graphs with random requests, similar results are obtained. A few differences are that in some scenarios all algorithms are more equal and that MSPF often perform almost as good as DIP-Cut.

In many of the scenarios DIP-Cut gives solutions in less than a second and in these cases it is easy to conclude that DIP-Cut is the better algorithm to use. An observation regarding the execution time is that when the greedy algorithms have been able to allocate all requests DIP-Cut has often, but not always, found a solution as well within a few seconds or minutes, sometimes even for graphs with 100 nodes and 60 requests. The problem with DIP-Cut is of course that it does not always give a solution within a reasonable time frame for some combinations of graphs and request sets. It is especially for large graphs where all requests can not be allocated simultaneously that DIP-Cut frequently times out or crashes.

6.7 Branch-and-cut implementation and potential

Our original plan included implementing the branch-and-price-and-cut algorithm described in chapter 3.6. Much time was spent understanding the algorithms and the implementation details. We quickly came to the conclusion that an implementation from ground up was out of the question, implementing an LP solver is a complex activity and obtaining good performance requires both a good grasp on all the technical details as well as a lot of fine tuning. Several frameworks suitable for branch-and-price-and-cut algorithms where investigated and DIP was selected for use in the development. Unfortunately it was found too late that DIP makes certain assumptions regarding branching that does not hold true for the algorithm. This was also found to be true even in other frameworks investigated and the time and effort necessary to circumvent this assumption or develop the algorithm without a framework was deemed to high. Instead we decided to use DIP with the algorithms DIP-Cut and DIP-Price-and-Cut, as described in chapter 3.

It is hard to say for certain what makes DIP-Cut time out without a solution. However, one problem with the algorithm is that the formulation grows very rapidly with the number of requests and arcs. It is possible that the branch-and-price-and-cut algorithm

described in chapter 3.6 could help alleviate this by using the path formulation together with column generation. On the other hand, if we compare the size of the networks and request sets used to test their algorithm in [9] with our scenarios we see that they are approximately the same sizes. They get fairly similar results as we got from using DIP-Cut, where an optimal solution is obtained in less than ten seconds in the majority of the cases, with a few timeouts with solutions that are not proven optimal as well as one case where their algorithm was not able to find any solution. While it is hard to compare their results directly with ours, it could indicate that the performances of the two algorithms are similar, and that the improvements from using their algorithm would not be substantial.

Further development of the branch-and-price-and-cut can only be done by diving deeper into the vast area that is operation research, especially ILP. While much work have been done in the area, it is a highly specialized field and even though many theoretical results are available there are many implementation decisions to be made when implementing the algorithms. Without in-depth knowledge it can be hard to make the necessary trade-offs and come up with the necessary improvements. The algorithm is also tremendously more complex than the simpler alternatives, leading to drastically larger development costs as well as making testing and verification significantly more difficult.

6.8 Alternatives

An alternative to design, implement and fine tune a specialized ILP solver could be to use commercial, general ILP solvers such as CPLEX. As we have stated several times before, producing a state-of-the-art ILP solver is a huge task that requires both a lot of technical know-how as well as resources. There are a lot of tricks and optimizations, from presolving to clever ways to generate cuts. This can lead to situations where the general solvers may outperform the specialized ones just because the specialization looses all those general optimizations. This was the case in the experiments performed by Alvelos in [4] where he found that using CPLEX as a general ILP solver outperformed the specialized algorithms he developed.

Another alternative could be to use several algorithms. You could let them all solve the problem simultaneously and then use the best solution or let them solve the problem one by one until a satisfying solution is obtained. Since SSP is the fastest algorithm, at least of the ones that are evaluated in this thesis, a good approach could be to use it first and if it fails to allocate all requests another algorithm could be used. First, a slightly more complicated algorithm such as MSPF could be tried and if also this fails a complex algorithm such as DIP-Cut could be used. If still no solution with all requests is found within reasonable time it might be best to use one of the non-optimal solutions found so far.

As for SSP, different results may also be obtained simply by handling the requests in different orders. So if some requests were not allocated in one solution, reallocating everything, starting by allocating the ones that were previously not allocated or just handling the requests in a different order, could potentially solve many instances where

SSP otherwise fail to allocate all requests.

It is also possible that there are other heuristics that can improve the algorithms and give good solutions for all instances that show up in practice. These heuristics could for example take advantage of different structures that are frequently present in the network topologies and request sets. However, since we have not investigated any such heuristics we can not say much about them.

7. Conclusions

In this thesis we have modelled the problem of allocating paths between RECs and REs in an RBS interconnect network as the Unsplittable Flow Problem. The main contribution is the development of a framework for implementing and evaluating suitable algorithms for this problem. Within this framework it is possible to generate random graphs and random requests as well as load predefined graph and request sets from XML. All implemented algorithms can then be tested on the same instances. The result can be analysed by the evaluation framework which provides a set of algorithm independent measurements of the quality of the solution. The results in this thesis are based on randomized test sets. This is partly due to the lack of defined future topologies and partly because of the simplicity of the topologies that are actually defined in the present. The differences of the algorithms in these topologies are in most cases negligible. The randomized test data is therefore used in order to help accentuate the differences that exists between the algorithms. When additional test data is available this framework can be used to evaluate the algorithms further as well as facilitate comparisons between new algorithms developed in the future.

UFP is NP-Hard and previously several non-greedy approximation algorithms have been proposed that can give a solution with some guarantee within a reasonable time frame. The major problem with the approximation algorithms are that they focus on the worst case and that can lead to poor performance in other cases, when requests that could have been allocated are rejected instead. The approximation algorithms investigated in this thesis do not reject requests and are based on breadth first search. Another algorithm investigated in this thesis is instead based on branch-and-cut and aims for finding the optimal solution. This algorithm naturally takes much more time and cannot hope to compete with the approximation algorithms in terms of the problem size that can be handled. Instead it can give an optimal solution or, if the time is too short for an optimal solution to be found, it either gives a solution with a measure on how far from optimal it is or it gives no solution at all.

Given the results described in the earlier chapters we see that the algorithms described in this report, SSP, MSPF, MCPF and DIP-Cut, are all capable of solving the problem but their performance differs depending on the characteristics of the given problem instance. SSP is suitable when it is expected that nearly all requests can be allocated. It is also the only one of the implemented algorithms that can be used when the requests need to be allocated online. MSPF is preferable to SSP if we know all requests in advance. It

has the best performance compared to the other algorithms when there is an abundance of requests and the objective is to maximize the total allocated capacity. Similary the performance of MCPF is also good when there are more requests than can be allocated and when the objective is to maximize the number of allocated requests. DIP-Cut is mostly suitable only for small instances. If all requests can be allocated it can often solve problems of moderate size. The objective function can be designed in many ways to change how it should optimize the solution, whether it is to maximize the number of requests, the allocated capacity or something else.

As we can see no single algorithm performs the best under all conditions. Though no algorithm can guarantee an optimal solution to all instances in a reasonable time frame, it is clear that in many instances, the choice of algorithm does not affect the outcome at all. In many cases the time taken by DIP-Cut to solve the problem to optimality is just negligibly longer than the time taken by the other algorithms. Still, in some relatively small instances DIP-Cut times out and that is clearly not desirable in the final algorithm. Hence for DIP-Cut to become a feasible alternative one needs to develop it further or alternatively combine it with a simpler backup algorithm for the cases when it fails to deliver in a reasonable time. On the other hand, SSP is the simplest, fastest and cheapest algorithm. In many cases its performance is comparable to the others', but in some cases it falls way behind. MSPF seems to be a good compromise between SSP and DIP-Cut. It is almost as fast as SSP while still achieving better results most of the time. At the same time it is considerably easier to implement than DIP-Cut. Therefore if any recommendations should be made, out of the algorithms we have evaluated, we would recommend MSPF to be used for the automatic request allocation. However, this is not a clear cut decision and is subject to change with new requirements and priorities.

Bibliography

[1] Expat. http://expat.sourceforge.net/, accessed March 2012.

[2] SimpleIni. http://code.jellycan.com/simpleini/, accessed March 2012.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Upper Saddle River, NJ, USA, 1993.

[4] Filipe Alvelos. *Branch-and-price and multicommodity Flows*. Phd, Universidade do Minho, February 2005.

[5] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive on-line routing. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 32–40, 1993.

[6] Baruch Awerbuch, Rainer Gawlick, Tom Leighton, and Yuval Rabani. On-line admission control and circuit routing for high performance computing and communication. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 412–423, 1994.

[7] Yossi Azar and Oded Regev. Strongly polynomial algorithms for the unsplittable flow problem. In *Proceedings of the 8th Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 15–29, 2001.

[8] Yossi Azar and Oded Regev. Combinatorial algorithms for the unsplittable flow problem. *Algorithmica*, 2006.

[9] Cynthia Barnhart, Christopher A. Hane, and Pamela H. Vance. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. *Operations Research*, Vol. 48(2):318–326, 2000.

[10] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1996.

[11] Alok Baveja and Aravind Srinivasan. Approximation algorithms for disjoint paths and related routing and packing problems. *Mathematics of Operations Research*, 25, 2000.

[12] Chandra Chekuri, Amit Chakrabarti, Anupam Gupta, and Amit Kumar. Approximation algorithms for the unsplittable flow problem. In *Algorithmica*, volume Vol. 47, pages 53–78. 2007.

[13] John W. Chinneck. Practical optimization: A gentle introduction. Carleton University, Ottawa, Canada. Available online at www.sce.carleton.ca/faculty/chinneck/po.html. Accessed February 2012, October 2011.

[14] Ericsson. Traffic and Market Data Report - November 2011. Bussiness report available online at http://hugin.info/1061/R/1561267/483187.pdf, accessed 2012-02-27.

[15] Ericsson. More than 50 billion connected devices, February 2011. White paper available online at http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf, accessed 2012-02-27.

[16] Ericsson. Heterogeneous Networks - Meeting Mobile Broadband Expectations with Maximum Efficiency, February 2012. White paper available online at http://www.ericsson.com/res/docs/whitepapers/WP-Heterogeneous-Networks.pdf, accessed February 2012.

[17] Matthew Galati. DIP - Decomposition for Integer Programming. accessed March 2012.

[18] Matthew Galati. *Decomposition methods for integer linear programming*. Phd, Lehigh University, January 2010. available online at http://coral.ie.lehigh.edu/ ted/files/papers/MatthewGalatiDissertation09.pdf, accessed March 2012.

[19] Matthew Galati and Ted K. Ralphs. *Integer Programming: Theory and Practice*, chapter Decomposition in Integer Programming, pages 57–110. CRC Press, September 2005. available online at http://coral.ie.lehigh.edu/ ted/files/papers/DECOMP04.pdf, accessed March 2012.

[20] Rainer Gawlick. *Admission Control and Routing: Theory and Practice*. Phd, Massachusetts Institute of Technology, June 1995.

[21] Bernard Gendron, Teodor Gabriel Crainic, and Antonio Frangioni. Multicommodity capacitated network design, 1997.

[22] Zonghao Gu, George L. Nemhauser, and Martin W.P. Savelsbergh. Lifted cover inequalities for 0-1 integer programs. *INFORMS Journal on Computing*, 10:417–426, 1998.

[23] Venkatesan Guruswami, Sanjeev Khanna, Rajmohan Rajaraman, Bruce Shepherd, and Mihalis Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In *Journal of Computer and System Sciences*, volume vol. 67, pages 473–496, November 2003 1999.

[24] P. Karlsson, P. Eriksson, and S. Ehnebom. Private communication, 2012.

[25] Richard M. Karp. On the computational complexity of combinatorial problems. In *Proceedings of the Symposium on Large-Scale Networks, Evanston, IL, USA, 18-19 April 1974.*, pages 45–68, January 1975.

[26] Jon M. Kleinberg. Single-source unsplittable flow. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 68–77, 1996.

[27] Petr Kolman and Christian Scheideler. Improved bounds for the unsplittable flow problem. *Journal of Algorithms*, Vol. 61(1):20–44, 2006.

[28] John E. Mitchell. *Handbook of Applied Optimization* , chapter Branch-and-Cut Algorithms for Combinatorial Optimization Problems. Oxford University Press, 2000.

[29] Stefan Parkvall, Anders Furuskär, and Erik Dahlman. Next Generation LTE, LTE-Advanced. *Ericsson Review*, (2):22–28, 2010. Avaliable online at http://www.ericsson.com/res/thecompany/docs/publications/ericsson_review/2010/next-generation-lte.pdf, accessed February 2012.

[30] Matthias A.F. Peinhardt. Integer multicommodity flows in optical networks, Mars 2003.

[31] Iraj Saniee and Daniel Bienstock. Atm network design: Traffic models and optimization-based heuristics. *Telecommunication Systems*, 16:399–421, 2001.

[32] Stefania Sesia, Issam Toufik, and Matthew Baker. *LTE: The UMTS Long Term Evolution: From Theroy to Practise.* John Wiley & Sons, second edition, 2009.

[33] Subhash Suri, Marcel Waldvogel, Daniel Bauer, and Priyank Ramesh Warkhede. Profile-based routing and traffic engineering. In *Computer Communications*, volume Vol. 26, pages 351–365. 2003.

[34] Xincheng Zhang and Xiaojin Zhou. *LTE-Advanced Air Interface Technology.* Auerbach Publications, 2012.