# METHODS FOR HIERARCHICAL NETWORK DESIGN

**Tommy Thomadsen**

**LYNGBY 2004**
**MASTER THESIS**
**13/02**

IMM

Printed by IMM, DTU

# Preface

This M.Sc. thesis is the final requirement for obtaining the degree: Master of Science in Engineering. The thesis describe work carried out in the period from 1st of September 2001 to 31st of January 2002. The work was carried out at the Operations Research section at the Institute of Informatics and Mathematical Modeling, DTU. Supervisor is Professor Jens Clausen.

Tommy Thomadsen

Lyngby, January 31th 2002

# Abstract

This thesis investigates hierarchical networks. We start by considering telecommunication networks where hierarchies exists. Telecommunication networks can be modeled as capacitated networks; hence the hierarchical networks are defined based on the capacitated networks.

A mathematical model is set up for a two level version of the hierarchical network problem and the hierarchical network problem is solved to optimality for up to 15 nodes and heuristically for up to 100 nodes.

The optimal solution algorithm is a branch-and-bound algorithm and the heuristic solution algorithm is a simulated annealing algorithm. They both solve the hierarchical network problem by solving a number of capacitated networks and aggregating the results.

Performance is measured for different versions of both algorithms, and the quality of the heuristic solutions are estimated by comparing these with optimal solutions when these can be found.

We conclude, that hierarchical networks using capacitated networks as the underlying network type can be meaningfully described and optimized. When solved heuristically, rather large networks (up to 100 nodes) can be handled easily.

**Keywords:** Hierarchical Networks, Topological Networks, Multilevel Networks, Capacitated Networks, Multicommodity Flow, Network Design, Operations Research, OR, Heuristics

# Contents

# Chapter 1

# Introduction

Previously, the hierarchical network problem has been described as that of finding the least cost, two-level hierarchical network, where the network must include a primary path from a predetermined starting node to a predetermined terminus node [10]. Some articles have followed up on the issue, e.g. [8, 9, 19, 20].

In this thesis an entirely new definition of hierarchical networks is introduced. The definition is motivated by telecommunication networks which are ordered in hierarchies of groups of telephone switches. The hierarchies are generally applicable to networks, however.

The thesis has starting point in the following questions:

- What are hierarchies in networks?
- How can networks containing hierarchies be optimized?

To answers these questions, we start by considering telecommunication networks, where hierarchies exist. Since telecommunication networks can be modeled as capacitated networks, we consider capacitated networks in particular and define hierarchies in this context.

Capacitated networks are notoriously difficult to optimize [12], and since this is the kind of networks we work with, no optimal solution is likely to be attained for large networks. Hence we aim for obtaining efficient heuristics, which give high quality solutions. For problems with few nodes, though, the problem is solved to optimality.

We start out by describing telecommunication networks (chapter 2), and define hierarchies and the hierarchical network problem (HNP) (chapter 3). The HNP is solved by solving a number of non-hierarchical network problems (NHNP's), i.e. HNP's with no hierarchies. Therefore we describe and solve these problems at first (chapter 4 to 6).

Thereafter we introduce hierarchies in the NHNP to obtain HNP (chapter 7) and solve the HNP optimally and heuristically (chapter 8 to 10). Since solving the HNP is done by solving a number of NHNP's, the major problem considered is that of handling the hierarchies.

Designing telecommunication networks is a strategic planning process, thus we assume that the time allowed to solve the hierarchical network problem is in the order of hours. Real world telecommunication networks have hundreds of telephone switches and are usually divided into 3 or 4 levels. The networks we solve contain up to 100 nodes, involving 2 hierarchies, which correspond roughly to the size of the top two hierarchies of telecommunication networks.

In chapter 11 tools for generating and managing networks which are used when testing the algorithms are descried. The performance is measure for different versions of both the optimal and the heuristic algorithm. The heuristic solutions are compared with the exact solutions when these can be found. This is done in chapter 12.

Implemented code and data used for testing are available from the author on request.

# Chapter 2

# Hierarchical Networks in Telecommunication

A telecommunication network consists of cables (optical or electrical wires) and switching and multiplexing equipment located at telephone switches (or exchanges) connecting subscribers and other switches.

The network is usually divided into three levels (in some cases more levels) - a national, regional and local level. The national level connects regional areas and the regional levels connect local areas. National, regional and local areas contain a number of local switches, and subscribers are connected directly to a local switch. Regional switches are also always local switches.

When a subscriber dials a number to another subscriber, the other subscriber is located, and a path is set up between the two subscribers. Which route to choose is programmed into the switching equipment, and thus setting up a path merely consists of reserving a fraction of the capacity for the call. This is done using signalling paths in the network.

If two subscribers are connected to the same local switch or to local switches, which can connect without using regional switches, such a connection is used, and regional and national switches are not used.

If the subscribers are connected to the same regional switch or regional switches, which can connect without using national switches, the call will only occupy connections to the regional switch and to the subscriber (see figure 2.1).

If subscribers are connected to different national switches a call will have to go through both local, regional and national switches (see figure 2.2).

Much of the traffic in the network is in fact data transmission, but the distinction between local, regional and national transmission is still valid though the traffic pattern may differ.

Cables and the equipment facilitating communication over the cables (in the following just cables) have different costs and capacities. In particular, cables of type STM-1 can carry e.g. 63 2Mbit lines, and lines with more capacity than STM-1 has capacity that grows with a factor 4 as in table 2.1. A good rule of thumb when comparing prices is that setting up a cable with 4 times as much capacity doubles the price, for the equipment.

The price of establishing a connection depends mainly on the cost of digging down a cable and the price of the equipment facilitating communication. The physical cable used for different capacities are usually the same.

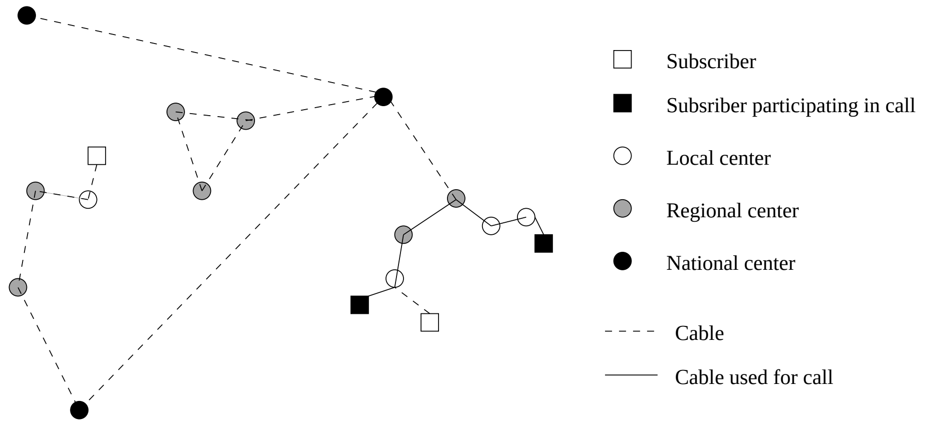


Figure 2.1: *Example of a regional call - some additional switches and connections are shown*

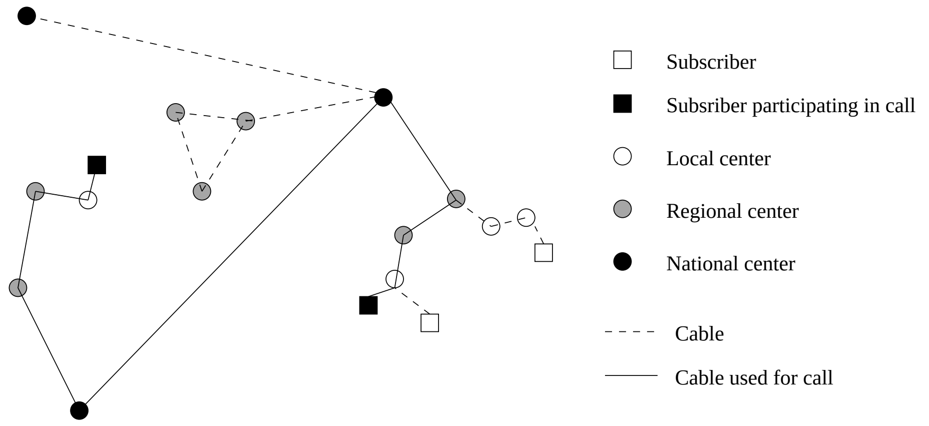


Figure 2.2: *Example of a national call - some additional switches and connections are shown*

| Type | Capacity | “Price” |
|---|---|---|
| STM-1 | 63×2Mbit 155Mbit | 1 |
| STM-4 | 4×155Mbit 620Mbit | 2 |
| STM-16 | 4×620Mbit 2,5Gbit | 4 |
| STM-64 | 4×2,5Gbit 10Gbit | 8 |

Table 2.1: *Cable types and equipment costs*

Usually higher-level switches (e.g. national switches) use connections with high capacities, but there is no direct dependency between switch level and connection type used. Thus high capacity connections can be established between local switches, if e.g. a customer has a need for a particularly high capacity connection between two places.

The network incorporates a high degree of redundancy to protect against failures. There are e.g. always 2 regional switches in a local area. In the local area, the local switches may be connected in some kind of mesh structure. Alternatively they may also be connected on a line, where the circuit allowing one broken connection is established through the regional switches (see figure 2.3).

Location of switches, cables and capacity limits, is in practice historically determined and has been determined and changed as the network evolved. The network continuously evolves, new cables and switches are added to the network, and replacement of existing equipment with higher capacity

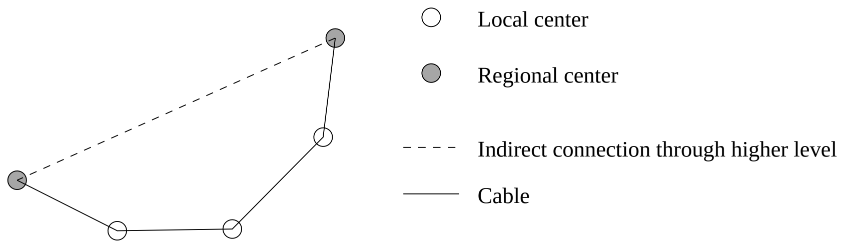


Figure 2.3: *Protection in local areas*

equipment is done frequently.

If a network were to be built from scratch, it would probably look very different from the current network. This is so since the need for capacity has changed over time, and thus an optimal location of switches and cables at some point in time may currently not be optimal.

The location of new switches, new cables and upgrade of switches to increase capacity over cables is of major concern, but also determining how an optimal solution would look like, if starting from scratch, would add information to the decision process.

## 2.1 Why do we have hierarchies?

In telecommunication networks, different cable capacities and hierarchies in part exist in order to allow cheap, low capacity connections where sufficient, while allowing higher capacity cables to be used where required.

In a sense this is not the reason for having hierarchies, since this is what different cable capacities give us, not the hierarchies. What the hierarchies contribute with here is instead an organizational element, that is, it divides the network into areas which can be handled and comprehended by staff who maintains and modify the network.

A related question is: Should hierarchies and edge capacities be tied together? That is, can a cable of e.g. type STM-64 be used between e.g. two local switches, or are cables of this type used between national switches only?

In telecommunication networks, this is not the case. Of course there is a tendency to have high capacity cables in higher levels (e.g. national level), but it is not a must. In some cases high capacity cables are set up if the capacity is needed regardless that it connects e.g. only local centrals.

In this project edge capacities and group levels are tied together. Also protection against failures is not considered. The hierarchical network is defined and described in the next chapter.

# Chapter 3

# Foundation and Definitions

In this chapter hierarchical networks and useful terms regarding the hierarchical networks are defined. The description takes its starting point in telecommunication networks, though differences exist. The most notify worthy is the inclusion of a flow-cost, which is not present in telecommunication networks. Flow-cost is, however, a well-known and commonly used concept in network modeling in general and hence has been included.

## 3.1 Data

The hierarchical network problem consist of a number of matrices giving demand, cost for setting up an edge and cost for using the edge. The costs are denoted setup-cost and flow-cost respectively. The different types of levels of edges are numbered from 1 to $L$ (indicated by $l$) 1 is the highest level (corresponding to national level in telecommunication networks). The matrices are shown in the following table:

| | |
|---|---|
| $D$ | Demand for each pair of nodes. |
| $CS_l, 1 \leq l \leq L$ | Cost for setting up an edge of level $l$ between any pair of nodes. |
| $CF_l, 1 \leq l \leq L$ | Cost per unit flow of a level $l$ edge between any pair of nodes. |

Thus for each level we have a setup-cost matrix and a flow-cost matrix, and thus for each edge a setup-cost and a flow-cost exists for each level.

Demand, setup-cost and flow-cost are all undirected. The demand is described with an origin-destination matrix and is in some contexts denoted commodity. Basically it is a request for capacity between node $i$ and node $j$ of the given value.

The capacities, which are the same for edges of the same level, are:

| | |
|---|---|
| $Cap_l$ | Capacity of level $l$ edge. |

Some demands or a setup-cost and flow-cost pair may be excluded - e.g. if there is no demand between two nodes or if it is not relevant to consider an edge between two nodes.

The hierarchical network design problem is that of finding the minimum cost subset of the possible edges and assigning a path to each demand, such that the capacities of each edge is not violated. The solution should be a “hierarchical network”, which will be defined shortly in section 3.2.4. To define a hierarchical network, we need some additional concepts.

## 3.2 Definitions

### 3.2.1 Node Level

The level of a node is the highest level (lowest number) of any edge incident to the node.

### 3.2.2 The Group

Level $l$ groups are the sets of connected components of the subgraph induced by level $l$ edges and nodes of higher levels than $l$, (i.e the level number $\leq l$).

Usually we only speak of a group if it has more than a single node, but the definition does not require it to be so. In hierachical networks, groups of single nodes exist, if a node of level $l$ ($1 \leq l < L$) has no nodes of level $l - 1$ connected.

For an example of the concepts see figure 3.1. All groups except the level 1 group is depicted. The level 1 group contains exactly the three level 1 nodes.

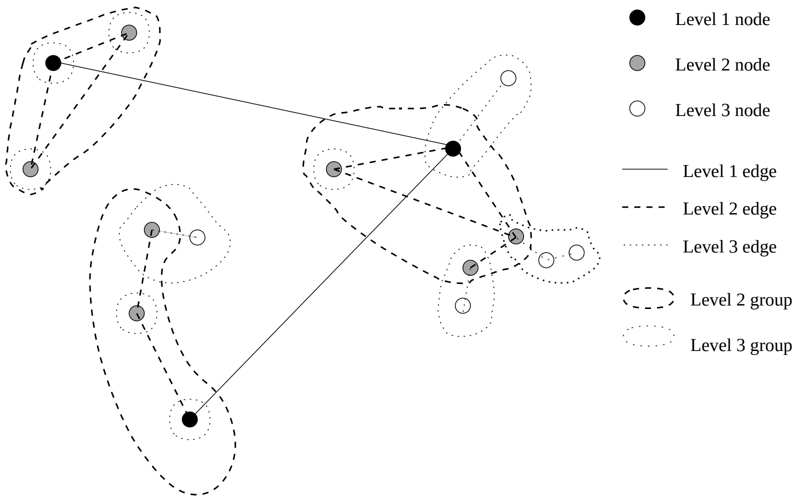


Figure 3.1: *Example of node level and groups.*

### 3.2.3 The Concentrator Node

We say that a node is concentrator node for a group, if it is in the group and has higher level than the group.

A node can be concentrator node in more groups, but if so, the groups have different levels. An example of this is the top left level 1 node in figure 3.1. This node is concentrator node for a level 2 and a level 3 group (which contains the concentrator itself only).

### 3.2.4 The Hierarchical Network

A hierarchical network is a subset of edges, such that:

- There is at most one edge between two nodes.

- The network is connected, hence there must exist a path from any node to any other node. The path may use edges of all levels.
- There are no edges with a lower level than both of its endpoints.
- A group has exactly one concentrator node, except the highest level group which has none.

The last implies that a path from a node to any higher level node will have to pass through one particular node, namely the concentrator node of the group to which the node belongs.

### 3.2.5 The Hierarchical Network Problem

The hierarchical network problem is defined as that of finding the minimum cost hierarchical network, which allows for routing demand (as defined in the demand matrix, $D$) in the network without exceeding the capacity limits of the edges. This includes finding paths for each demand, since this is not in general simple.

The Hierarchical Network Problem is abbreviated HNP. A Non-Hierarchical Network Problem, which is a HNP with one level only is abbreviated NHNP.

The NHNP is similar to the "Capacitated Network Design Problem" described in e.g. [12] and [14].

## 3.3 Demand Paths

Given an edge selection, the path each demand takes should be determined. Given an edge selection, and the fact that the solution is a hierarchic network, the sequence of groups traversed by an acyclic path between two nodes is, however, unique. We say that hierarchical networks are trees with respect to groups, where the highest level group may be considered the root.

That the sequence of groups traversed by an acyclic path between two nodes is unique can be seen from the following:

Starting from the lowest layer (highest numbered), the level $L$ groups are induced by level $L$ edges, and nodes of higher level, i.e. all nodes. Hence we have a division into sets, and each set contains exactly one higher level node (since it is a hierarchical network), i.e. the concentrator node. The concentrator nodes all have level $L-1$ or higher.

Level $L-1$ groups are induced by level $L-1$ edges and nodes of level $L-1$ or higher. Hence the nodes which are a subset of the concentrator nodes of level $L$. Each level $L-1$ group hence connects as many level $L$ groups as there are nodes in the group, and each path in this part of the network is unique with respect to the groups traversed, since each group contains exactly one concentrator.

In general any level $l-1$ groups are induced by level $l-1$ edges and nodes of level $l-1$ or higher. Assuming nodes of levels lower than or equal to $l$ can be reached from exactly one of the nodes of level $l-1$ or higher, paths are unique with respect to group and since there is only one concentrator in each group, connecting the groups by level $l-1$ groups will maintain that the groups traversed between two nodes are unique.

Hence recursively, paths in a hierarchical network are unique with respect to groups.

Within a group several paths may exist between two nodes. An example of this can be seen for the top left level 2 group in figure 3.1. Hence determining a path between two nodes can be done by finding the unique sequence of groups to be traversed and for each group determining the path to take in this group.

## 3.4 Capacity

Ensuring that capacity limits are not exceeded can be done for each group in turn as described in the following.

For a particular group, we remove all edges within the group, i.e. edges with both ends incident to nodes in the group. Then we will have exactly as many components as we had nodes in the group, and each component contains exactly one node from the group. As an example, the right-most level 2 group in figure 3.1 is considered. In figure 3.2 the group edges have been removed.

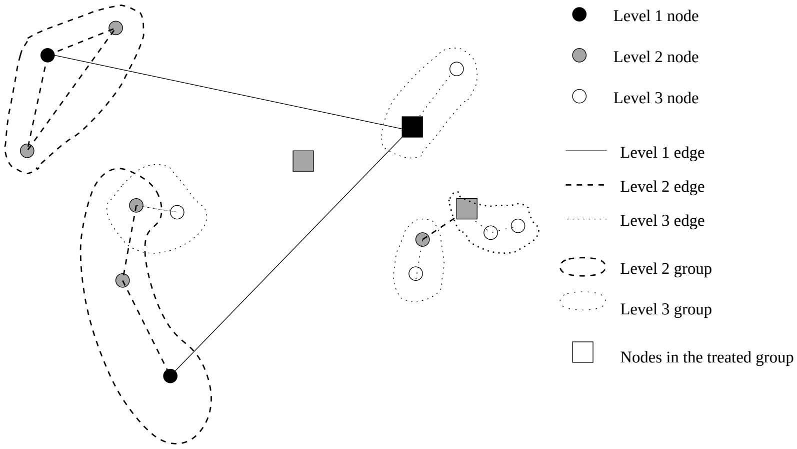


Figure 3.2: *Edges removed from right-most level 2 group.*

A demand matrix for the group can then be built by considering each pair of nodes in the group (the squares in figure 3.2) in turn. For a given pair of nodes, the total demand between them is calculated by first identifying the components they are in. The total demand between the two nodes is then the sum of demand between all pairs of nodes, where one node is in one of the identified components and the other node is in the other component.

Thus for each group we now have a NHNP, which can be solved and capacities can be checked. If done for each group, and no capacities are exceeded, the hierarchical network is feasible.

The solution of the NHNP is the subject of the following 3 chapters.

# Chapter 4

# The Non-Hierarchical Network Problem

In this chapter a mathematical model defining the Non-Hierarchical Network Problem is described. The definition is motivated by telecommunication networks, but not limited to these. E.g. flow-cost is introduced, though it is not used in telecommunication networks.

The NHNP problem solution is used as a building block in the solution to the HNP problem. This description should ease understanding of the mathematical model for the HNP, which follows in chapter 7.

As mentioned earlier, the Non-Hierarchical Network Problem is similar to the "Capacitated Network Design Problem" described in e.g. [12] and [14]. These problems are in general NP-hard due to the capacities [12], and hence this is also the case for the NHNP.

## 4.1 Mathematical Model

### 4.1.1 Definitions

| | |
|---|---|
| $V$ | Set of all nodes. |
| $E$ | Set of all edges. |
| $i, j, k, l$ | $\in V$ - Nodes. |
| $ij, i \in V, j \in V, i < j$ | $\in E$ - Undirected edges. |

We need the concept of a cut in a network. It is denoted (following [6]) $\delta(A)$, where $A \subseteq V$, and is defined as the set of all edges which have an endpoint in $A$ and an endpoint in $V \backslash A$.

### 4.1.2 Data

| | |
|---|---|
| $cs_{ij}, i < j$ | Cost of setting up an edge between $i$ and $j$. |
| $cf_{ij}, i < j$ | Cost per unit flow on the edge between $i$ and $j$. |
| $d_{ij}, i < j$ | Undirected demand between $i$ and $j$. |
| $cap$ | Capacity of edges (the same for all edges). |

$cs_{ij}$ is denoted the setup-cost and $cf_{ij}$ is denoted the flow-cost.

We assume that the data are demand-connected, that is no cut exists with demand equal to zero. Hence to fulfill demands, the solution must be connected as well. For most real world networks and telecommunication networks in particular, this is a reasonable assumption.

### 4.1.3 Decision Variables

| | |
|---|---|
| $x_{ij} \in \{0,1\}$<br>$(i < j)$ | 1 if there is an edge between $i$ and $j$,<br>0 otherwise. |
| $f_{ijkl} \geq 0$<br>$(i < j, k < l)$ | Amount of flow on edge $i$ to $j$ resulting from<br>demand between nodes $k$ and $l$. |

### 4.1.4 Objective Function

The cost of a network is the sum of costs of setting up edges and the sum of all flow through edges:

$$\min \sum_{i,j,i<j} cs_{ij} \cdot x_{ij} + \sum_{i,j,i<j,k,l,k<l} cf_{ij} \cdot f_{ijkl} \tag{4.1}$$

The first part of the objective function is denoted the total setup-cost and the second part is denoted the total flow-cost.

### 4.1.5 Constraints

#### Tree Constraint

When solving optimally we require solutions to be trees, this is done to ease the solution process. When solving heuristically this constraint is relaxed.

Since connectivity is assumed, the following is enough to ensure tree solutions:

$$\sum_{i,j,i<j} x_{ij} = |V| - 1 \tag{4.2}$$

When the only solutions considered are trees, the flow-variables (i.e. $f_{ijkl}$) can be uniquely determined since paths are unique. In this case they are bookkeeping variables and not decision variables. If the tree constraint is relaxed, the flow-variables *are* decision variables.

#### Flow Constraints

Recall that $f_{ijkl}$ is the amount of flow on the edge between $i$ and $j$ resulting from demand between node $k$ and node $l$. A flow is an assignment of values to the $f_{ijkl}$ variables, which does not violate any constraints in this subsection.

The constraints are introduced one type at a time.

First of all, if an edge is used (i.e. $f_{ijkl} > 0$), then there should be an edge between $i$ and $j$ (i.e. $x_{ij} = 1$). $f_{ijkl}$ is required to be nonnegative, and assuming $M$ is larger than any possible assignment to $f$, the following constraints can be used:

$$\forall i, j, k, l, i < j, k < l : M \cdot x_{ij} \geq f_{ijkl} \tag{4.3}$$

The amount of flow incident to node $i$ and node $j$ resulting from demand $ij$ must equal the value of the demand between $i$ and $j$:

$$\forall i, j, i < j :$$

$$\sum_{k,i<k} f_{ikij} + \sum_{k,i>k} f_{kiij} = d_{ij} \tag{4.4}$$

$$\sum_{k,k<j} f_{kjij} + \sum_{k,k>j} f_{jkij} = d_{ij} \tag{4.5}$$

For all demands between node $k$ and node $l$, the total flow incident to other nodes $i, i \neq k, i \neq l$ resulting from demand $kl$ should equal zero (if $i$ is not on the path between $k$ and $l$) or two times the required demand (if $i$ is on the path between $k$ and $l$). That is, either of the following 2 constraints must hold:

$$\forall k, l, k < l, i \in V \backslash \{k, l\} :$$

$$\sum_{j \in V, i<j} f_{ijkl} + \sum_{j \in V, j<i} f_{jikl} = 0 \tag{4.6}$$

$$\sum_{j \in V, i<j} f_{ijkl} + \sum_{j \in V, j<i} f_{jikl} = 2d_{kl} \tag{4.7}$$

This means that flow resulting from a demand between a fixed pair of nodes, is on one path, i.e. the flow is not split. For trees this is trivially so, since for a pair of nodes there is exactly one path.

Introducing either-or-constraints is computationally expensive if solving the mathematical model directly (e.g. by CPLEX), since this introduces a binary variable for each pair of constraints. Hence this is one of the reasons, the problem cannot be solved directly for more than a few nodes.

### Capacity Constraints

Capacity constraints ensure, that no edge has more flow than its capacity allows. This can be expressed as:

$$\forall i, j, i < j : cap \cdot x_{ij} \geq \sum_{k,l,k<l} f_{ijkl} \tag{4.8}$$

This makes equation 4.3 unnecessary, since if for a given $i$,$j$ equation 4.8 holds, then so does equation 4.3.

Notice that, since we assume there exists no cut with demand equal to zero, equation 4.8 ensure the network is connected.

An alternative formulation that does not use the calculated flow but instead use cuts is the following:

$$\forall S, \emptyset \subset S \subset V : \sum_{ij \in \delta(S)} d_{ij} \leq \sum_{ij \in \delta(S)} x_{ij} \cdot cap \tag{4.9}$$

The formulation requires that flow can split, and is not suitable for implementation, since the number of constraints grows exponentially with the number of nodes.

# Chapter 5

# Optimal Solution of the NHNP

In this chapter an algorithm for solving the NHNP to optimality is presented. The algorithm is used in solution strategies when solving the HNP, and thus understanding and being able to solve the NHNP is crucial when solving HNP.

## 5.1 Branch and Bound

The optimal solution strategy used is a branch and bound scheme based on Kruskal's minimum spanning tree (MST) algorithm [7]. The described solution strategy will allow trees only. Trees have in general lower setup-cost than non-trees, hence trees are among the cheap solutions for a given NHNP. Allowing trees only does limit the solution space, hence a feasible tree solution may not exist though a non-tree solution exists. However, solving the NHNP for trees shows to be computationally demanding and since we suspect it to be even more computationally demanding for non-trees, it will be solved heuristically instead.

Notice that the NHNP without capacity limits and no flow-cost is in fact the MST problem. If we allow for capacity limits, but no flow-cost, a MST is a lower bound on the solution value. In fact if a MST is feasible, it *is* the optimal solution. Thus it seems reasonable to base a solution algorithm, and the branching order in particular, on an MST algorithm.

Kruskal's MST algorithm considers all edges ordered in increasing order of cost, and choose either to include or exclude the current edge from the MST. The edge is included in the MST, if it does not introduce a cycle, otherwise it is excluded. Since the network is undirected, this is equivalent to determining if the two endpoints of the edge are in the same component. If they are not in the same component they are included in the MST, and the two components are merged.

The idea of the branch and bound process is to imitate this processing, except that when the MST algorithm choose to include the edges, two cases are created, one in which the edge is included in the solution and one in which it is not.

## 5.2 Representation

The NHNP consists of some data which are not changed, that is setup-cost, flow-cost and demand for all pairs of nodes which are represented as matrices. Also we need the number of nodes, edge capacity and initially we create a list of edges, which is sorted in increasing order of setup-cost, giving the order in which edges are considered.

A particular instance of a partial solution is called a `Net` object. The `Net` object contains, for each edge, a specification of whether the edge is included in the solution. In fact three cases exist: The edge is included in the solution, the edge is excluded, or it is undecided. This is maintained in a matrix so that the state of an edge can be easily checked and updated. Initially all edges are undecided.

Since we require solutions to be trees we maintain disjoint sets representing components (as Kruskal's MST algorithm does). In this way it is easy to check if an edge can or cannot be used, since if the two edge endpoints are in the same component it cannot be used. This is used when branching.

## 5.3 Branch

Solving a particular `Net` object is done by branching on an undecided edge, hence creating two new `Net` objects, one where the edge is included in the solution, and one where the edge is excluded. The edge to branch on is the minimum setup-cost edge with endpoints in two different components. Finding this edge is done by iterating through the list of edges calculated initially. The iterator position is recorded and given to the newly created `Net` objects, such that iteration can be continued from the point where the edge is found.

The order in which the solution instances are considered is by depth-first, and of the two possibilities, we chose first the one where an edge is included in the solution. Doing the exact opposite, that is choosing first the one where an edge is excluded from the solution works much worse, and the usual idea of choosing first the one with the lowest bound value, does at least not work better.

Measuring the number of branches needed for a few examples using the "included edge first" and "lowest bound first" -strategies, indicates that the "included edge first" -strategy works marginally better, since it require marginally fewer branches. In the current implementation, there is also an overhead from handling the "lowest bound first" -strategy, hence the "included edge first" -strategy is used.

When adding an edge, this results in an immediate cost increase, namely the setup-cost of the edge. The total cost of all edges in the current solution is kept so that it can be used when calculating the bound. Hence the cost is updated when an edge is added, i.e. the cost is calculated incrementally.

The same goes for flow-cost, since if an edge is added this result in an immediate cost increase resulting from demands, both through the added edge, but also through other edges. If a path exists between node $i$ and $j$ this is the path, which will be used for the final solution, since solutions are trees only. Hence the flow-cost can be calculated when a path exists and reused in following branches. This is described in the next section.

## 5.4 Calculating flow-cost & detecting exceeded capacity

We keep the value of the total flow-cost and update it each time an edge is added. When updating the total flow-cost, we traverse a path for each demand if a path exists and if it has not been traversed previously. Hence when $V - 1$ edges has been added all flow-costs for each demand has been calculated once and the path traversed once.

We keep a flow-matrix giving the amount of flow on each edge, in order to be able to check if the capacity of any edge is exceeded. Since we traverse each demand path exactly once, this matrix is updated at the same time and the capacity is checked. If the capacity is exceeded an infeasible solution has been generated, and the branch is fathomed.

To speed up path processing, we keep a successor-matrix (or predecessor-matrix since the network is undirected), which gives for any pair of nodes $i$ and $j$ the next node $n$ on the path from $i$ to $j$, if such one exists. When traversing a path, this information is used recursively, so that the the following node on the path from $i$ to $j$ is the next node on the path from $n$ to $j$ until the node reached is $j$.

Figure 5.1 gives an example of a network which has been partially connected, and indicates an edge which is to be added.

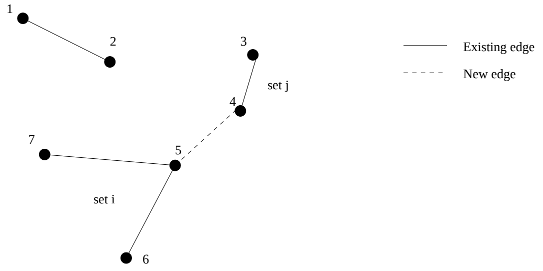


Figure 5.1: *Example network used to exemplify how the successor-matrix and the flow-matrix is updated.*

Figure 5.2 gives the successor-matrix for the example network. Updates resulting from the new edge is indicated in bold.

| **Succ** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|
| **1** | 1 | 2 | | | | | |
| **2** | 1 | 2 | | | | | |
| **3** | | | 3 | 4 | **4** | **4** | **4** |
| **4** | | | 3 | 4 | **5** | **5** | **5** |
| **5** | | | **4** | **4** | 5 | 6 | 7 |
| **6** | | | **5** | **5** | 5 | 6 | 5 |
| **7** | | | **5** | **5** | 5 | 5 | 7 |

Figure 5.2: *Successor-matrix for the example network*

If no path exists from node $i$ to $j$, then the corresponding entry in the successor-matrix is empty. In the example, e.g. entry $[1, 3]$ is empty since no path exists between node 1 and 3.

When an edge $kl$ is added, the two components containing $k$ and $l$ denoted *set k* and *set l* are recorded. Then the successor-matrix is updated by updating all entries corresponding to nodes $i$ and $j$, where either $i \in$ *set k* and $j \in$ *set l* or vice versa.

Assume $i \in$ *set k* and $j \in$ *set l*, the symmetric case is handled the same way.

The update use the fact, that for two nodes in the same component, the path between the two nodes is known. Hence finding the path from node $i$ to node $j$ consist of the path from node $i$ to node $k$, edge $kl$ and the path from node $l$ to node $j$. The two paths are both known from the existing successor-matrix, and may consist of no edges if $i = k$ or $j = l$.

Hence an update is done as follows: If $i = k$, then entry $[i, j]$ in the successor-matrix is updated to $l$. Correspondingly if $j$ is $l$, $[j, i]$ is updated to $k$. In the example this corresponds to, when edge 4-5 is added, updating e.g. entry $[5, 3]$ to 4.

On the other hand if $i \neq k$, then entry $[i, j]$ in the successor-matrix is updated to the value of entry $[i, k]$, and when $j \neq l$, entry $[j, i]$ is updated to the value of entry $[j, l]$. In the example this corresponds to, when edge 4-5 is added, updating e.g entry $[7, 3]$ to 5, which is the value of entry $[7, 5]$.

Figure 5.3 shows the flow-matrix for the example network. $d_{ij}$ is demand between node $i$ and $j$, and added demand is indicated in bold. The flow is only calculated in the lower left half of the matrix, since the flow is undirected, so less than half of the flow matrix is used.

| **Flow** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|
| **1** | | | | | | | |
| **2** | $d_{12}$ | | | | | | |
| **3** | | | | | | | |
| **4** | | | $d_{34} + \mathbf{d_{35}}$<br>$+\mathbf{d_{36}} + \mathbf{d_{37}}$ | | | | |
| **5** | | | | $\mathbf{d_{35}} + \mathbf{d_{36}}$<br>$+\mathbf{d_{37}} + \mathbf{d_{45}}$<br>$+\mathbf{d_{46}} + \mathbf{d_{47}}$ | | | |
| **6** | | | | | $d_{56} + d_{67}$<br>$+\mathbf{d_{36}} + \mathbf{d_{46}}$ | | |
| **7** | | | | | $d_{57} + d_{67}$<br>$+\mathbf{d_{37}} + \mathbf{d_{47}}$ | | |

Figure 5.3: *Flow-matrix for the example network*

When an edge $ij$ is added, all demands for which a path did not exist previously, but now exists because of edge $ij$ are added to the flow-matrix. This is exactly all demands represented by all pairs of nodes where one is in *set* $i$ and the other one in *set* $j$.

For each of these demands the value of the demand is added along the path they use. The path is found using the successor-matrix.

In the example edge 4-5 is added, hence the demands to update are 3-5, 3-6, 3-7, 4-5, 4-6 and 4-7. For e.g. demand 3-5 the demand value $d_{35}$ is added along the path from 3 to 5, that is edges 3-4 and 4-5. The same is done for the other demands.

## 5.5 Bounding

The bound consists of two parts, the setup-bound and the flow-bound. The bounding procedure returns either a lower bound on the solution value, an exact solution, or it notes that the current edge selection result in an infeasible solution. If the solution is feasible and $|V| - 1$ edges has been selected, the solution value is immediately given, since as described above, the solution value is calculated incrementally each time an edge is added. This incremental value is used for both the setup-bound and the flow-bound. If the solution is feasible, and less than $|V| - 1$ edges has been selected, the bound is calculated.

### 5.5.1 Setup-cost Bound

The setup-bound part consist of the setup-cost of edges which has already been added and the minimum sum of setup-costs required to connect the remaining components. The bound is calculated by finding an MST with respect to setup-costs using edges between components only. The capacity limits and the flow-cost are ignored. If the remaining edges cannot result in a connected network, the solution is infeasible, which is detected.

### 5.5.2 Flow-cost Bound

The flow part of the incrementally calculated solution value correspond to those demands $ij$, where a path exists between node $i$ and $j$, since we look for tree solutions only. Hence for those demands the exact solution value is used. For demands $lk$ where no such path exists, we bound by calculating the minimum flow-cost path from node $l$ to $k$ using edges, which have not been explicitly excluded from the solution.

An example of the flow-bound is shown in figure 5.4.

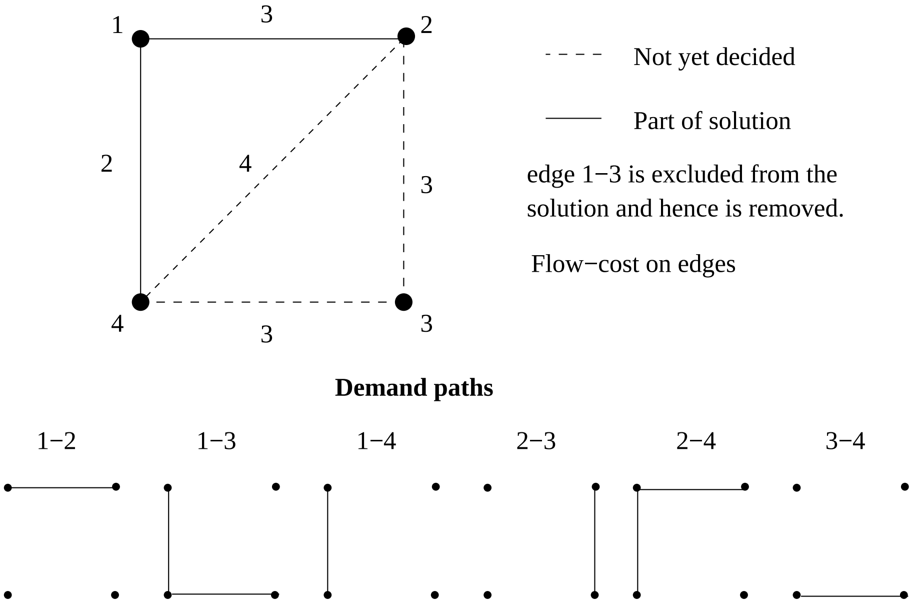


Figure 5.4: *Example on flow-bound*

In the example edge 1-3 is excluded from the solution, edge 1-2 and 1-4 are included in the solution, and for the rest of the edges, it has not been decided whether they are in the solution. Demands where a path exists using edges which are in the solution use this path, i.e. demands 1-2, 1-4 and 2-4. For the rest of the demands a shortest path is found and used as bound, where the path use edges that are in the solution, and edges that are yet undecided. Hence demand 1-3 use the path 1-4-3, demand 2-3 use path 2-3 and demand 3-4 use path 3-4.

As mentioned branching consists of either including an edge or excluding an edge from the solution. In the case where an edge is included in the solution, the flow-bound does not change, since the same edges are available when bounding. However, since an edge is added, some demands are now included in the incrementally calculated exact solution value, and hence should be excluded from the bounding. Thus the calculated bounds for each demand are saved, and reused if the demand is not included in the incrementally calculated exact solution value.

### 5.5.3 Preventing Cycles

The calculated flow-bound does in some cases use edges, which can immediately be seen to introduce cycles. Since it is a bound calculation, this does not invalidate the bound, but on the other hand the bound could be improved by disallowing this.

For a given branch some edges are included and some are excluded. Consider a shortest path used as bound for a demand in this branch. If adding all edges on the path results in a cycle, then the bound can be improved (assuming the shortest path is unique). This can be done, since cycles are not allowed in the final solution. Thus for this branch, a final solution will not contain all the edges of the bound path, hence an alternative bound path should be used.

Each time a bound is calculated, this should be checked and a minimum cost alternative path which does not introduce cycles should be found. In general this is not possible to do fast, and hence no gain can be foreseen. In the case where one edge introduce the cycle, it can be done fast. An example of this is shown in figure 5.5.

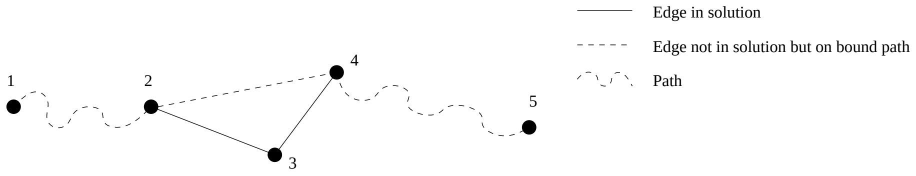


Figure 5.5: *Example on a path containing an edge, which alone introduce a cycle*

Assume the path used to bound a flow-cost for the demand 1-5 is the path indicated by the dashed lines in the figure. If the edges represented by the dashed lines were added to the solution, a cycle would be introduced (2-3-4-2), hence a better bound on the flow-cost can be attained by disallowing this path.

In this case the cycle is introduced by one edge namely edge 2-4, which connects two nodes which are already connected, and hence the nodes are in the same component. Information on components are available, since disjoint sets are maintained representing the components. Hence when building the path used for bounding, and at some point edge $ij$ is to be added to the path, we check that node $i$ and $j$ is not in the same component. If $i$ and $j$ is in the same component, an alternative edge is found.

This interfere with the reuse of the flow-bound described in section 5.5.2. The reuse of the flow-bound required that adding an edge could not change the value of the flow bound. The flow-bound can change now, since adding an edge may invalidate the choice of other edges, hence the flow-bound can be improved.

Whether it is worth updating the flow-bound or reusing it is unclear, but no matter what, adding this functionality seems to slow the processing down. Since introducing checks for cycles in general (i.e. that multiple edges introduce cycles) would be more time-consuming, and the quality of the bound cannot be expected to improve accordingly, this has not be tried.

## 5.6 Extensions

The most important extension is to get rid of the tree-requirement. Solving to optimality using the above scheme could not be done immediately, since cycles would be allowed in the network. Hence demand paths are not unique, and which path to take would have to be decided. Building a branch-and-bound algorithm to do this could be done by, in addition to branching on whether an edge is in the solution or not, include branches on whether an edge is used on the path for a given demand.

This would increase the computation time compared to allowing solutions to be trees only. Since the computation time is high already, this will not be pursued when solving to optimality. Instead the problem is solved for non-trees heuristically. This is described in the following chapter.

# Chapter 6

# Heuristic Solution of the NHNP

In this chapter a heuristic solution algorithm for the NHNP is presented. The algorithm is guaranteed to find a feasible solution, if one exists, but the found solution is usually not optimal. The solution algorithm is used when solving the HNP and will be run many times, hence it has to be rather fast.

The algorithm is outlined in figure 6.1.

```
Check that a feasible solution exists
Find an initial tree solution which has low setup-cost
while solution is infeasible do
    Add edge
    Find demand paths
end while
Run local search
```

Figure 6.1: *Heuristic solution algorithm for the NHNP*

The major difference from the optimal solution is that we allow non-trees. Doing this, we will have to determine for each demand which path it takes, i.e. the $f_{ijkl}$ variables are decision variables.

The phases in the figure are described in the following sections.

## 6.1 Checking whether a Feasible Solution Exists

Note that we assume a complete network, and that flows cannot split. Hence checking that a feasible solution exists can be done by checking that no demand exists, which is larger than the edge capacity. If this is the case, then the solution containing all edges between nodes and with each demand $ij$ taking the path consisting of exactly edge $ij$ is feasible.

On the other hand, if a demand larger than the edge capacity exists, this demand cannot even leave the node, since no edges exist with high enough demand, and we do not allow demands to split.

## 6.2 Finding an Initial Tree Solution

The initial solution found is an MST with respect to setup-costs. For networks where the setup-costs contribute significantly to the objective function value compared to the flow-cost, this is a reasonable choice. Telecommunication networks can be modeled reasonably without any flow-cost, hence at least in this case it is reasonable, in fact as mentioned in section 5.1, if there is no flow-cost and the found MST is feasible, then the MST is an optimal solution.

## 6.3 Finding Demand Paths

Given an edge selection, an assignment to the flow variables is sought which is feasible and has a low total flow-cost. The variable assignment is found (if one exists) indirectly by finding the path each demand takes.

This is equivalent, since if paths are assigned for each demand, the flow variable assignment can be found by for each demand and for each edge in the path, to set the corresponding flow variable equal to the value of the demand. On the other hand, given a flow variable assignment, each path for each demand can be found by considering the values of the flow variables for the corresponding demand. One variable for each edge exists, and given the equations 4.4, 4.5, 4.6 and 4.7, the positive variables correspond to a path.

Calculating the optimal assignment is computationally expensive, and since this is done many times a heuristic solution approach is used.

The demand paths are assigned by considering demands in decreasing order of value. This is done this way, since high value demands are more expensive to reroute than low value demands. For each demand $ij$, the shortest feasible path between $i$ and $j$ is found. This is done by first finding the shortest path between $i$ and $j$.

If this path is not feasible, the first edge where the edge capacity would be exceed if the demand were added is identified. The path should not use this edge, hence it is temporarily removed from the network, and the shortest path between $i$ and $j$ is found again. This is continued until either the demand can be added along the found path or no path exists between $i$ and $j$. In both cases the temporary edges are added again.

If a feasible path was found, demand is added, and processing is continued for the rest of the demands. If no feasible path was found, the edge with exceeded capacity found in the original network is recorded. This edge will be relieved by adding another edge as described in the following section.

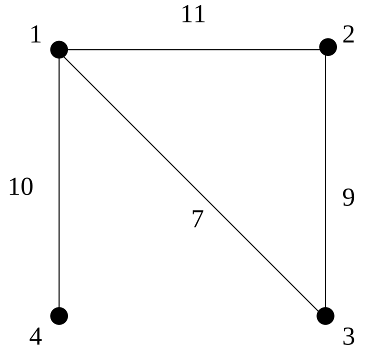
## 6.4 Finding an Edge to Add

The edge to add should relieve the edge with exceeded capacity found above.

The edge to add is found by considering all edges which are not part of the solution in increasing order of setup cost. For each edge $ij$ we check if the current path for demand $ij$ uses the edge with exceeded capacity, if so this is the edge we add.

There is no guarantee that such an edge exists, a situation which arises in practice. An example of the situation, where no edge is immediately found is shown in figure 6.2.

Demands are ordered according to value, depicted in the left column in the figure. One edge with exceeded capacity exists, edge 1-2. Edge 1-2 is used by demand 1-2 and demand 1-3 only. The



| Demand | Value | Path |
|---|---|---|
| 3−4 | 7 | 3−1−4 |
| 1−3 | 6 | 1−2−3 |
| 1−2 | 5 | 1−2 |
| 1−4 | 2 | 1−4 |
| 2−3 | 2 | 2−3 |
| 2−4 | 1 | 2−3−1−4 |

Edge capacity is 10

Total demand on edges

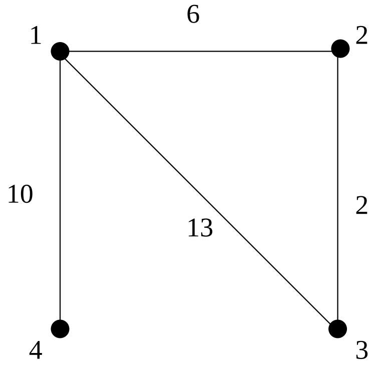
Figure 6.2: *Relieve-edge cannot be immediately found*

corresponding edges 1-2 and 1-3 are both part of the solution, hence it is not immediately possible to find an edge, which will relieve 1-2.

We can, however make sure that such an edge exists, by requiring that if edge $ij$ is in the solution then demand $ij$ uses the path consisting of edge $ij$ only. Consider the edge with exceeded capacity $kl$. Since no demand exists with value larger than the capacity, at least two demands use this edge, one is the $kl$ demand, and assume the other demand is $mn$. Then edge $mn$ is not in the solution, otherwise demand $mn$ would use edge $mn$, and hence adding edge $mn$ would relieve edge $kl$.

Hence what we will do if no edge can be found, which relieves the edge with exceeded capacity is to first assign the path consisting of edge $ij$ to demand $ij$ for which edge $ij$ exists. Then the above path assignment scheme is used, and an edge is found. In fact doing this may result in a feasible solution without adding any edges, simply by considering the paths in another way.

Using this algorithm would for the example result in the paths depicted in figure 6.3.



| Demand | Value | Path |
|---|---|---|
| 1−3 | 6 | 1−3 |
| 1−2 | 5 | 1−2 |
| 1−4 | 2 | 1−4 |
| 2−3 | 2 | 2−3 |
| 3−4 | 7 | 3−1−4 |
| 2−4 | 1 | 2−1−4 |

Edge capacity is 10

Total demand on edges

Figure 6.3: *Relieve-edge is guaranteed to exist*

In fact the edge with exceeded capacity identified is no more edge 1-2 but edge 1-3, and since demand 3-4 use edge 1-3, and edge 3-4 is not in the solution, edge 3-4 is the edge to add.

We might be tempted to simply use this scheme all along, instead of waiting till the first scheme fails. The thing is of course, that in general the first scheme is better than the second one in the sense that it has a higher chance of finding a feasible assignment, and that it also in general finds lower valued path assignments. This is supported by experiments in section 12.4.2

## 6.5 Local Search

When a good initial solution is found, we run a local search algorithm modifying the solution if an improvement is immediately possible. The algorithm has been run with two neighbourhoods, a simple and an extended neighbourhood.

The simple neighbourhood consist of solutions which can be reached by either adding or removing an edge. The extended neighbourhood allows in addition a swap of edges, i.e. one edge is added and another one is removed.

For the simple neighbourhood, the local search runs as follows: For all possible edges try to add the edge, if the edge is not part of the solution, and remove the edge if it is part of the solution. Test if the solution is feasible and better than the current solution. If so the neighbour-solution becomes the current solution, and all neighbour-solutions are checked for this new current solution. This continues until no feasible and better solutions exist.

The effect using this neighbourhood is mainly to remove unnecessary edges added in earlier steps of the algorithm.

The extended algorithm is used the same way, only now swaps of edges are considered as well.

Using the simple and especially the extended neighbourhood increase the runtime severely (see section 12.4.1). To avoid getting into problems with runtime, the extended neighbourhood is used only for networks with up to 10 nodes, and the simple neighbourhood is used for networks with up to 15 nodes.

This is a way of controlling the time used but a very inflexible one, in the sense that it is not possible to control exactly how much time is spent on each group. As an example, solving a network containing 10 nodes takes approximately 1 second using the extended neighbourhood whereas solving a network containing 11 nodes takes approximately 0.1 seconds using the simple neighbourhood (see section 12.4.1). Also the solution quality drops accordingly. Hence it may be beneficial to have groups with nodes 10 compared with having groups with 11 nodes, simply because an algorithm finding better solutions is used for solving groups with 10 nodes.

Another possibility could be to use a simulated annealing approach using the simple neighbourhood, but allowing for accepting marginally worse solutions. This way the runtime could be controlled better, and more solutions may be reachable.

# Chapter 7

# The Hierarchical Network Problem

## 7.1 Mathematical Model

In the following the mathematical model defining the hierarchical network problem is described. The model involves only two levels referred to as the primary (highest level - level 1) and secondary level.

The solution network will consist of one primary group and a number of secondary groups. All groups mentioned in this chapter are secondary groups, and thus this is not always explicitly stated. Also concentrator nodes in the resulting network are exactly the primary nodes, and thus these terms can be used interchangeably.

Solving the HNP in polynomial time would make the NHNP solvable in polynomial time as well, since the NHNP is a special case of the HNP ("divide" a network into 1 group). Since the NHNP is NP-hard (see chapter 4) so is the HNP.

### 7.1.1 Definitions

| | |
|---|---|
| $V$ | Set of all nodes. |
| $E$ | Set of all edges. |
| $i, j, k, l$ | $\in V$ - Nodes. |
| $ij, i \in V, j \in V, i < j$ | $\in E$ - Undirected edges. |
| $h, 1 \leq h \leq G$ | Groups. |

### 7.1.2 Data

| | |
|---|---|
| $G$ | Number of groups in the network. |
| $cs1_{ij}, i < j$ | Cost of setting up a primary edge between $i$ and $j$. |
| $cs2_{ij}, i < j$ | Cost of setting up a secondary edge between $i$ and $j$. |
| $cf1_{ij}, i < j$ | Cost per unit flow of a primary edge between $i$ and $j$. |
| $cf2_{ij}, i < j$ | Cost per unit of of a secondary edge between $i$ and $j$. |
| $d_{ij}, i < j$ | Undirected demand between $i$ and $j$. |
| $cap1$ | Capacity of primary edges. |
| $cap2$ | Capacity of secondary edges. |

As for the NHNP, we assume that the data are demand-connected (no cut exists with demand equal to zero), and thus to fulfill demands, the solution must be connected as well.

Usually (e.g. for telecommunication networks), the setup-cost for primary edges are higher than for secondary edges, i.e. $cs1_{ij} > cs2_{ij}$ and the flow-cost for secondary edges are higher than for primary edge, i.e. $cf1_{ij} < cf2_{ij}$. The model does not require it to be this way, but all tested networks have these characteristics.

### 7.1.3 Decision Variables

$x1_{ij} \in \{0,1\}$ $(i < j)$ 1 if there is a primary edge between $i$ and $j$, 0 otherwise.

$x2_{ij} \in \{0,1\}$ $(i < j)$ 1 if there is a secondary edge between $i$ and $j$, 0 otherwise.

$t_{ik} \in \{0,1\}$ 1 if node $i$ is concentrator node in group $k$, 0 otherwise.

$g_{ik} \in \{0,1\}$ 1 if node $i$ is in group $k$, 0 otherwise.

$f1_{ijkl} \geq 0$ $(i < j, k < l)$ Amount of flow on edge $i$ to $j$ resulting from demand between nodes $k$ and $l$ on a primary edge.

$f2_{ijkl} \geq 0$ $(i < j, k < l)$ Amount of flow on edge $i$ to $j$ resulting from demand between nodes $k$ and $l$ on a secondary edge.

The group ($g$) and concentrator ($t$) variables are in a sense bookkeeping variables, though a selection of edges ($x$) and flow assignment ($f$) does not give a unique assignment to the group and concentrator variables. But if edges are selected and flow assigned, it as matter of selecting which groups are assigned which numbers, and this is unimportant. In fact the objective value is determined from the flow and setup variables only, hence the assignment to group and concentrator variables does not matter as long as a feasible assignment exists.

### 7.1.4 Objective Function

The cost for a given network is the total cost of setting up edges, and the sum of all flow through edges:

$$\begin{aligned} \min \sum_{i,j,i<j} & cs1_{ij} \cdot x1_{ij} + cs2_{ij} \cdot x2_{ij} + \\ & \sum_{i,j,i<j,k,l,k<l} cf1_{ij} \cdot f1_{ijkl} + cf2_{ij} \cdot f2_{ijkl} \end{aligned} \tag{7.1}$$

As for the NHNP, the first part of the objective function is denoted the total setup-cost and the second part is denoted the total flow-cost.

### 7.1.5 Group & Concentrator Constraints

Between two nodes there can be one edge only, either a primary or a secondary edge but not both:

$$\forall i, j, i < j : x1_{ij} + x2_{ij} \leq 1 \tag{7.2}$$

A node is in exactly one group:

$$\forall i : \sum_{1 \leq h \leq G} g_{ih} = 1 \tag{7.3}$$

Each group has exactly one concentrator node:

$$\forall h (1 \leq h \leq G) : \sum_i t_{ih} = 1 \tag{7.4}$$

A node can only be concentrator in one group:

$$\forall i : \sum_h t_{ih} \leq 1 \tag{7.5}$$

If there is a primary edge between $i$ and $j$ ($x1_{ij} = 1$), then node $i$ and $j$ are both primary nodes ($\sum_h t_{ih} = 1, \sum_h t_{jh} = 1$).

$$\forall i, j, i < j : x1_{ij} \leq \sum_h t_{ih} \tag{7.6}$$

$$\forall i, j, i < j : x1_{ij} \leq \sum_h t_{jh} \tag{7.7}$$

If $i$ is a concentrator/primary node ($\sum_h t_{ih} = 1$), then there is a primary edge incident to $i$ ($\sum_{j,i<j} x1_{ij} + \sum_{j,i>j} x1_{ji}$).

$$\forall i : \sum_h t_{ih} \leq \sum_{j,i<j} x1_{ij} + \sum_{j,i>j} x1_{ji} \tag{7.8}$$

If a node is concentrator/primary in a group, then it is in the group as well.

$$\forall i, h : t_{ih} \leq g_{ih} \tag{7.9}$$

If a node $i$ can be reached from $j$ via a secondary link then $i$ and $j$ are in the same group.

$$\forall i, j, h, i < j : x2_{ij} + g_{ih} \leq g_{jh} + 1 \tag{7.10}$$

$$\forall i, j, h, i < j : x2_{ij} + g_{jh} \leq g_{ih} + 1 \tag{7.11}$$

As mentioned earlier, some solutions are equal, i.e. have the same edges and flows selected and hence the same objective function value. The concentrator selection and group division is also given, the only difference is the group numbers, which are different. We do not care what group numbers are and do not distinguish between those solutions.

### 7.1.6 Tree Constraint

So far the model allows solutions, which are not trees (they have to be connected though). When finding flow variables it is, however, a major advantage to consider trees only, since given a solution, the flow can be determined uniquely. When solving optimally, we will only do it for tree solutions, or equivalently (since connectivity is required), the number of edges should be $|V| - 1$:

$$\sum_{i,j,i<j} x1_{ij} + x2_{ij} = |V| - 1 \tag{7.12}$$

When heuristically this constraint is relaxed, and hence when solving heuristically, it is not enough to find the edge-variables ($x_1$ and $x2$), flow-variables ($f1_{ijkl}$ and $f2_{ijkl}$) will also have to be determined.

### 7.1.7 Flow Constraints

We consider at set of constraints, which determines the amount of flow on edges.

A flow is an assignment of values to the variables $f1_{ijkl}$ and $f2_{ijkl}$, which does not violate any constraints mentioned in this subsection (7.1.7). Recall that $f1_{ijkl}$ is the flow on a primary edge between $i$ and $j$ resulting from demand between $k$ and $l$, likewise for $f2_{ijkl}$.

First of all, if an edge is used (i.e. $f1_{ijkl} > 0$ or $f2_{ijkl} > 0$), then there should be a primary edge (if $f1_{ijkl} > 0$) or a secondary edge (if $f2_{ijkl} > 0$) between $i$ and $j$ (i.e. $x1_{ij} = 1$ or $x2_{ij} = 1$ respectively).

$f1_{ijkl}$ and $f2_{ijkl}$ should be positive or zero, and assuming $M$ is larger than or equal to any possible assignment to $f1$ and $f2$, the following constraints can be used:

$$\forall i, j, i < j :$$

$$\forall k, l, k < l : M \cdot x1_{ij} \geq f1_{ijkl} \tag{7.13}$$

$$\forall k, l, k < l : M \cdot x2_{ij} \geq f2_{ijkl} \tag{7.14}$$

The amount of flow incident to node $i$ and node $j$ resulting from demand between $i$ and $j$ must equal the demand between $i$ and $j$:

$$\forall i, j, i < j :$$

$$\sum_{k, i<k} (f1_{ikij} + f2_{ikij}) + \sum_{k, i>k} (f1_{kiij} + f2_{kiij}) = d_{ij} \tag{7.15}$$

$$\sum_{k, k<j} (f1_{kjij} + f2_{kjij}) + \sum_{k, k>j} (f1_{jkij} + f2_{jkij}) = d_{ij} \tag{7.16}$$

For all demands between node $k$ and node $l$, the total flow incident to other nodes $i, i \neq k, i \neq l$ resulting from demand $ij$ should equal zero (if $i$ is not on the path between $k$ and $l$) or two times the required demand (if $i$ is on the path between $k$ and $l$). That is either of the following 2 constraints must hold:

$$\forall k, l, k < l, i \in V \backslash \{k, l\} :$$

$$\sum_{j \in V, i<j} (f1_{ijkl} + f2_{ijkl}) + \sum_{j \in V, j<i} (f1_{jikl} + f2_{jikl}) = 0 \tag{7.17}$$

$$\sum_{j \in V, i<j} (f1_{ijkl} + f2_{ijkl}) + \sum_{j \in V, j<i} (f1_{jikl} + f2_{jikl}) = 2d_{kl} \tag{7.18}$$

As for the NHNP formulation, this means that flow resulting from a demand of capacity between a fixed pair of nodes, is on one path, i.e. the flow is not split. For trees this is trivially so, since for a pair of nodes there is exactly one path. In fact $f1_{ij}$ and $f2_{ij}$ are bookkeeping variables as opposed to decision variables, when the tree constraint is used. If non-trees are allowed, $f1_{ij}$ and $f2_{ij}$ are in fact decision variables, but with this formulation, the flow still cannot split.

If solved directly by e.g. CPLEX, the either-or formulation can be handled as follows: Introduce a binary variable for each new pair of constraints *temp#*. Then the following constraints are equivalent with equation 7.17 and equation 7.18:

$$\forall k, l, k < l, i \in V \backslash \{k, l\} :$$
$$\sum_{j \in V, i<j} (f1_{ijkl} + f2_{ijkl}) + \sum_{j \in V, j<i} (f1_{jikl} + f2_{jikl}) = 2d_{kl} \cdot temp\# \tag{7.19}$$

Introducing binary variables are always computationally expensive, thus this formulation cannot be expected to be useful for solving large instances of the HNP.

### 7.1.8 Capacity Constraints

Capacity constraints ensure, that no edge has more flow than its capacity allows. This can be expressed as:

$$\forall i, j, i < j : cap1 \cdot x1_{ij} \geq \sum_{k,l,k<l} f1_{ijkl} \tag{7.20}$$
$$\forall i, j, i < j : cap2 \cdot x2_{ij} \geq \sum_{k,l,k<l} f2_{ijkl} \tag{7.21}$$

This makes the constraints 7.13 and 7.14 redundant, since if 7.20 and 7.21 are fulfilled then so are 7.13 and 7.14.

The capacity constraints also ensure connectivity, since we assume there exists no cut where the amount of demand crossing the cut is equal to zero.

An alternative formulation, that does not use the calculated flow is:

$$\forall s, \emptyset \subset s \subset V : \sum_{ij \in \delta(s)} d_{ij} \leq \sum_{ij \in \delta(s)} x1_{ij} cap1 + x2_{ij} cap2 \tag{7.22}$$

This formulation require the constraints 7.13 and 7.14.

As for the NHNP, this formulation requires that flow can split, and is not suitable for implementation, since the number of constraints grows exponentially with the number of nodes.

In the following implementations the first formulation is used (i.e. equation 7.20 and 7.21), i.e. flow is calculated to ensure that capacities are not exceeded.

## 7.2 Number of Edges

The number of primary and secondary edges are given by:

$$\sum_{i,j,i<j} x1_{ij} = G - 1 \tag{7.23}$$
$$\sum_{i,j,i<j} x2_{ij} = |V| - G \tag{7.24}$$

Assuming the tree constraint holds.

This can be seen from the following:

Equation 7.4 say that each group has exactly one concentrator node, and since there are $G$ groups, there are $G$ concentrator or primary nodes.

Since the graph as a whole is a tree ($V - 1$ edges are selected (equation 7.12) and the graph is connected (equation 7.20, 7.21 and network is demand-connected)), the subgraph consisting of primary nodes and edges connecting the nodes is a forest (including the tree case). Thus there can be no more than $G - 1$ edges.

On the other hand assume there were less than $G - 1$ edges. If so, all primary nodes cannot be directly connected, thus the subgraph contains at least two trees (i.e. the subgraph is a forest but *not* a tree). In the original graph the two trees must have been connected via a secondary edges, since the graph as a whole is a tree. But equation 7.10 and 7.11 state that if a secondary node connects two nodes, then they are in the same group. Thus we have two primary nodes in the same group, which cannot be the case because of equation 7.4. Thus there are exactly $G - 1$ primary edges, and the remaining $|V| - G$ edges are of course secondary.

An important fact, which follow from the proof is that the subgraph consisting of primary nodes is a tree.

## 7.3 Solving Directly

Obviously the problem cannot be solved directly for more than a few nodes, since the number of constraints grows exponentially in the number of nodes. In order to make sure that the mathematical model stems with what is our understanding of hierarchical networks, (i.e. as described in chapter 3), it is however solved for problems with few nodes. This is done by generating an input file to CPLEX, which contains all constraints.

A program generating input files to CPLEX from a definition file has been implemented. Solving for small networks (up to about 8 nodes and 3 groups) gives reasonable results, i.e. fitting our understanding of hierarchical networks, hence groups are divided, edges are selected within groups and not between groups and so on. Also since comparing results (i.e. the optimal solution found) with solutions found by other solution algorithms described in the following gives sufficing results, the mathematical model seems to describe the desired hierarchical network problem.

Both versions of the capacity constraints are used, and test samples with 7 and 8 nodes seems to indicate that they perform similarly, though more constraints are generated in the cut formulation (i.e. equation 7.22). Since the number of cuts grows exponentially with the number of nodes, whereas the number of constraints using the flow version (i.e. equation 7.20 and 7.21) grows only linearly with the number of edges, the flow version is expected to perform best.

## 7.4 Extensions

The mathematical model as described in this chapter allows two levels only. Hence an extension would be to allow an arbitrary number of levels. This could be done by introducing a level index to the data variables, setup-cost($c$), flow-cost($cf$), capacity($cap$) and number of groups ($G$). That is the number of groups at each level would have to be specified. At the moment we only require the number of secondary groups to be specify, since the number of primary groups is trivially one. The number of level $l + 1$ nodes must be $\geq$ the number of level $l$ nodes.

The decision variables for edges ($x$), flow ($f$), concentrators ($t$) and groups ($g$) should be indexed by a level as well. There should be a number of group variables for each level corresponding to how many groups are required, defined by $G$.

The objective function should sum over the level variable instead of having two separate terms for the primary and secondary edges. Tree-, flow- and capacity-constraints can be extended by summing over the level variable in the same way.

In general the group- & concentrator-constraints can be extended by modifying constraints such that they should hold for groups of level l or concentrators of level l, which should e done for all levels. Hence this will result in roughly the number of levels times as many constraints.

Another extension would be to allow $G$ to be unknown, that is the mathematical model should be formulated such that any number of groups is allowed. This could be done by having a number of group variables equal to the number of nodes (instead of exactly $G$ group variables) and then allow groups to be empty.

The extensions are not included, first of all because the mathematical model described above is hard enough to solve as it is. If the mathematical model mentioned above can be solved, then the extensions could be added.

# Chapter 8

# Solution Strategies for the Hierarchical Network Problem

In this chapter two solution strategies are presented. The first one, "the branch and bound strategy", is built directly on the branch and bound solution algorithm to NHNP described in chapter 5. It works with edges, and operates only implicitly with groups, concentrator nodes and hierarchies.

The second solution strategy, "the phase divided strategy" divides the process into phases. This allows for conceptual abstraction of the problem and division of the solution development, so that phases could potentially be improved one at a time. The phases are of course interrelated, so changing one phase affect others. This strategy shows to be particularly suited for heuristic solution algorithms.

Thus the first strategy is potentially the most efficient, since it looks at the problem from an overall perspective, but it is very hard to work with. On the other hand the second strategy is easier to work with, but may be less efficient because of abstraction. In following chapters we will work with the second strategy, that is the phase divided strategy.

## 8.1 Branch and Bound Strategy

This strategy generalize the branch and bound solution algorithm for the NHNP described in chapter 5. It selects edges, so a solution is an assignment to the binary $x1_{ij}$ and $x2_{ij}$ variables. As for the NHNP solution algorithm paths and flow is calculated incrementally, and disjoint-sets are maintained representing components.

The strategy generate solutions as done for the NHNP, only now there is two passes of the edges, where the first pass selects the primary edges, and the second pass selects the secondary edges. The generated solutions all comply with all constraints defining the HNP, except for the capacity which is checked separately.

The strategy first selects $G - 1$ primary edges, including update of the disjoint-sets representing components and find paths and flow. Each selection of an edge is a branch, hence to sub-cases are created as for the NHNP algorithm.

As stated in section 7.2, the primary edges must form a connected subgraph. This can be ensured as follows.

Record which nodes have a primary edge incident to them and thus are primary, and record the number of primary components (i.e. components containing a primary node). Also record the number of selected edges. Each time an edge is added and components are merged, the recorded info is updated.

The number of primary components is updated as follows. If two primary components are merged, the number of primary components is decreased. If two non-primary components are merged, the number of primary components is increased, and finally if a primary and a non-primary component is merged, the number of primary components is maintained.

When $G-1$ primary edges have been added, the number of primary components shall be 1, since then all primary edges are in the same component and thus they form a subtree.

Since the number of primary sets can only decrease with one for each new edge, a generated solution must comply with:

$$G - 1 - \texttt{numberOfPrimaryComponents} \geq \texttt{numberOfSelectedEdges}$$

Hence it makes sure, that if at some point an edge is added edges can be selected such that the final primary edge selection is a connected subgraph. If adding an edge results in a solution which does not comply with this inequality, the solution it is not generated.

When the primary edges has been generated, the process continues from beginning again, only now we select secondary edges, and edges which are selected as primary are not considered.

The setup-cost and flow-cost is calculated incrementally as for the NHNP, except that the value and capacity of an edge depend on whether it is a primary or secondary edge.

Choosing to select primary edges before secondary edges, has lower runtime than the opposite. This may be because of the primary edges having a higher setup-cost, and thus placing primary edges gives a better (higher value) bound.

Finally we note that the group numbers are not uniquely determined, but as said earlier, we do not care what number each group has.

The strategy has been implemented in Java with success. It is, however, not suitable for a heuristic implementation, and since this is what we aim for, it is not used any further. Solutions found has been compared with solutions found by the optimal algorithm based on the following solution strategy, i.e. it is made sure that the same optimal solutions are found.

## 8.2 Phase Divided Strategy

The phase divided strategy divides the solution process into 4 phases:

1. Divide into groups
2. Choose concentrator nodes
3. Select edges in each group (including primary group)
4. Assign paths to demands

One way to divide into groups and choose concentrator nodes is simply to try all possibilities. When solving optimally this is what we will do. This works for smaller networks, but for larger networks the amount of possibilities gets so large, that it is not possible to do this.

It is also possible to find a division of nodes into groups and selection of concentrator nodes heuristically. Measuring the quality of a group division and concentrator selection is hard though, since the objective function value does not depend directly on how groups are divided but on which edges are selected, and how much flow is on each edge. Thus it would be nice to have an evaluation function, which gives an estimate on the quality of the group division and concentrator selection. Ultimately the evaluation function should equal the objective function, but calculating the exact value is slow for large networks.

The third phase is to select the edges, such that the objective value is minimized. Each group can be optimized in turn (including the primary group), i.e. we have to solve G+1 NHNP's with data derived from the HNP. The data can be derived as initially described in section 3.4 and further specified in section 9.2.1 .

The fourth phase involves assigning a path to each demand. This is not relevant in the optimal case, since we consider trees only. When solving heuristically, the assignment has already been described in chapter 6 for the NHNP. Assigning paths to demands one group at a time actually solves the problem, since paths are unique with respect to groups, and hence the paths in the network is made uniquely from the paths in the groups (see section 3.3 and 9.2.2).

The optimal solution algorithm described in chapter 5 to solve the NHNP problems can only be used for networks of sizes up to about 7-8 nodes in a reasonable time. Thus if solving networks of larger sizes, i.e. groups exist with more than 7-8 nodes, heuristics will have to be applied as well. The heuristic used finds an MST, and if this MST is not feasible, the MST is modified (i.e. edges are added) until the MST is feasible.

The phase divided strategy will be used to solve the HNP both optimally (for small networks) and heuristically in the two following chapters.

Solving the HNP this way, allows for replacing the NHNP solution algorithm easily, since there is a clear distinction between phases. Hence if an alternative solution algorithm where developed or adopted this could be immediately done using this strategy.

# Chapter 9

# Optimal Solution of the HNP

This chapter describes an algorithm, which solves the HNP to optimality. It is based on the phase division strategy, and thus have four phases. The first and second phase are simple, namely try all possibilities of group division and concentrator selection.

The third and fourth phase use the solution algorithm for the NHNP (described in chapter 5) to solve each group to optimality. This can be done since the optimal solution of the HNP is the union of the optimal solutions of the groups.

## 9.1 Representation

A solution is represented as two arrays of length |V|, a group array and a concentrator array. The group array is an integer array giving group numbers for each node and the concentrator array is a binary array where entries are true if the node is concentrator otherwise false. The group numbers are 0 to $G-1$, and at least one node has to be in each group. Also each group has exactly one concentrator; thus G entries in the concentrator array are true, and the nodes which are concentrator nodes have different group numbers.

The group-numbers do not matter so given this representation some solutions are symmetrical - two solutions are symmetrical, if the groups are divided the same way, and the same nodes are concentrator nodes, but group-numbers differ. To avoid generating symmetrical solutions we require, that if node number $i$ has group-number $h$, then for all groups with numbers $0 \leq g < h$, at least one node $j$ exist with group-number $g$, such that $j < i$. In this case the group number assignment is valid.

This ensures that no symmetrical solutions exists, since the group number can be uniquely determined. This can be done by assigning group numbers to nodes in order of node number. The node number 0 is given group number 0. Recursively node number $i$ is given number $g$ if a node $j$ with number $j < i$ is in the same group as node $i$ and $j$ has group number $g$.

If no node with number $< i$ exists which is in the same group as $i$, then $i$ is assigned group number $h$ where $h$ is the least group number larger than all group numbers assigned to nodes with number $< i$. $i$ cannot be assigned a group number which is smaller than $h$, since it is not in the same group as any of the nodes with number $< i$, and if assigned an unassigned group number less than $h$, the group assignment would not be valid.

On the other hand $i$ cannot be assigned a larger number, since if so either there is not enough group numbers or a group number assigned later is smaller than this, and hence is not a valid group number assignment. Thus the group numbers are unique.

The search space is all possible valid assignments to the group- and concentrator-arrays.

This representation does not allow for some of the nodes to be either not assigned to a group or undecided whether they are concentrators. The representation can easily be extended though, by simply allowing a special marker to be assigned as either group number or concentrator which indicated that it is yet undecided.

## 9.2 Solution Algorithm

All possible group divisions and concentrator selections are generated. For each group division and concentrator selection, the determination of the solution value is done by determining the solution value for each group in turn. Solving the groups is done as described in chapter 5, hence the only thing left is to determine how data are computed for the groups, and how the solutions to the groups are aggregated again to attain the solution for the entire network.

### 9.2.1 Calculating NHNP data from HNP data

Given a group-division and a concentrator-selection, most of the data can be immediately extracted from the HNP problem - the number of nodes is the number of nodes in the group, edge capacity, setup-costs and flow-costs depend only on the group being primary or secondary.

The demand-matrix cannot be determined immediately, since for the primary group, the demand is a sum of demands between nodes, and likewise for secondary groups, a demand between a node in the group and a node outside the group is replaced with demand between the concentrator and the node. This is described in detail in the following.

The demand matrix is generated by considering all pairs of nodes in the HNP. An example of a network where group division and concentrator selection is known, is shown in figure 9.1. The figure is used to exemplify the following discussion.

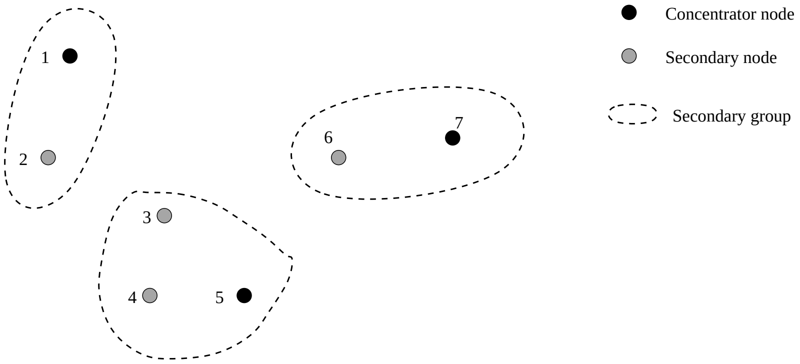


Figure 9.1: *Example of a network where groups are known and concentrators are selected*

The primary group consists of all concentrators, hence an entry in the demand matrix exists for each pair of concentrator nodes. The values of the demand matrix is determined by considering all pairs of nodes. If two nodes are in different secondary groups, the concentrators of the two groups are identified. The demand between the two nodes is then added to the entry in the demand matrix corresponding to the two concentrator nodes.

E.g. the demand between node 2 and 3 in figure 9.1, is added to the demand between concentrator nodes 1 and 5, since the path from 2 to 3 consists of the subpaths from 2 to 1, from 1 to 5 in the primary group, and finally from 5 to 3.

If nodes are in the same group, their communication does not go through the primary group, and thus is not added to the demand matrix. The resulting demand matrix for the primary group is shown in figure 9.2. Demands for a node pair $i, j, i < j$ is denoted by $d_{ij}$, and since demand is undirected less than half of the demand matrices are used.

| | **1** | **5** | **7** |
|---|---|---|---|
| **1** | | | |
| **5** | $d_{13} + d_{14} + d_{15}$<br>$+d_{23} + d_{24} + d_{25}$ | | |
| **7** | $d_{16} + d_{17}$<br>$+d_{26} + d_{27}$ | $d_{36} + d_{37} + d_{46}$<br>$+d_{47} + d_{56} + d_{57}$ | |

Figure 9.2: *Demand-matrix for the primary group*

The demand matrix for secondary groups has an entry corresponding to each pair of nodes in the group. Each pair of nodes is considered, and demand is added, if one or both nodes are in the group, otherwise it is not. If both nodes are in the group, the demand is added between the two nodes. If one node is in the group only, the demand is added from this node to the concentrator node, since this is part of the entire path.

The resulting demand matrices for the secondary groups are shown in figure 9.3(a), 9.3(b) and 9.4.

| | **1** | **2** |
|---|---|---|
| **1** | | |
| **2** | $d_{12} + d_{23} + d_{24}$<br>$+d_{25} + d_{26} + d_{27}$ | |

(a) *Top left group*

| | **6** | **7** |
|---|---|---|
| **6** | | |
| **7** | $d_{16} + d_{26} + d_{36}$<br>$+d_{46} + d_{56} + d_{67}$ | |

(b) *Top right group*

Figure 9.3: *Demand-matrix for the two top groups*

| | **3** | **4** | **5** |
|---|---|---|---|
| **3** | | | |
| **4** | $d_{34}$ | | |
| **5** | $d_{13} + d_{23} + d_{35}$<br>$+d_{36} + d_{37}$ | $d_{14} + d_{24} + d_{45}$<br>$+d_{46} + d_{47}$ | |

Figure 9.4: *Demand-matrix for the bottom middle group*

If we consider the demand between node 2 and 3 again, this demand is added in both the top left group between node 1 and 2, and in the bottom middle group between node 3 and 5. For e.g. demand 34, the demand is added to edge 34 only, and since neither node 3 nor 4 is concentrator this is the only demand on this edge.

The nodes are renumbered in the groups, such that iterating through the nodes of a group is simple. In practice two arrays are built and kept, one which maps the node number in a group to a node number in the full network and one which maps the other way. Hence a demand matrix is built, which contains exactly the necessary nodes, i.e. the nodes in the group.

Since the flow- and setup-costs are used often, it may make sense to copy them, such that no mapping between group numbers are necessary. On the other hand the mapping is so simple that it ought not be necessary to copy them. Experiments show that there is a marginal difference only, but pre-copying the data tend to be fastest, hence this is done.

### 9.2.2 Aggregating NHNP Solutions to attain Solution for HNP

When the solutions to the NHNP's are found, the solution to the HNP is feasible if all NHNP's are feasible, and the solution value is simply attained by adding the solution value for each of the groups. The selection of edges is immediately attainable from the edges which are selected in the groups.

Finding the flow is done by finding the paths to each demand (this is equivalent - see section 6.3). The path for a demand $ij$ consist of up to three parts. Assume $I$ is concentrator of the group $i$ is in and $J$ is concentrator for $j$ and $I \neq J$, then the path consist of the path from $i$ to $I$, the path from $I$ to $J$ and finally the path from $J$ to $j$. This information is simply extracted from the solution to the NHNP's if the paths has to be found. If $I = J$, i.e. nodes are in the same secondary group, then the path is simply taken from the solution to the particular NHNP.

## 9.3 Reducing Search Space

The algorithm basically has the following problems:

- The number of different groups increases exponentially with the number of nodes.
- The solution time for a group (a NHNP) increase exponentially with the number of nodes, because the number of constraints increase exponentially with the number of nodes.

Thus when adding a node to a problem, the number of different groups at least doubles, and the new groups are larger and thus more difficult to solve.

Measurements show that though there are many more small groups, the large groups take most time to solve. Thus, the size of networks, which can be handled, can be increased by limiting the size of groups allowed. This is done by checking that each group has sizes less than a predetermined value.

Doing this does not help much in it self, since now the first issue comes into play - now it is the number of different groups that is the problem. This is considered in the following section.

## 9.4 Reusing Secondary Group Solution Values

Since all group divisions and concentrator selections are tried, a lot of secondary groups are calculated more than once. In figure 9.1 if we change the concentrator node in the group consisting of nodes 1 and 2 from 1 to 2, the solutions of the two other secondary groups are not changed. Thus the solution value for a group and selected concentrator can be saved when it has been calculated, so that it can be reused later.

This is done by maintaining an array of floats, where each possible secondary group has an entry. There are $2^{|V|}$ possible subsets of a size $|V|$ network, but the solution to a secondary group also depends on the selection of concentrator. Thus there are less than $|V| \times 2^{|V|}$ possible solutions.

It is important that the solution value can be found fast if it has been calculated, thus there have to be a simple mapping from a secondary group description to an entry in the array. The mapping used is to take a binary array of length $|V|$, where element i is 1 if node i is in the set, otherwise zero. This binary array is then read as a binary number, and multiplied with $|V|$ and the node number of the concentrator node is added to obtain the index in the array.

Thus each time a secondary group is to be calculated, we check the array to see if the value has previously been calculated, and if so, we simply use the value, otherwise we calculate it and record the solution value. If the group is infeasible, this is recorded as well.

A lot of the entries in the array are not used, since some indexes represents groups which are not generated (e.g. the group consisting of all nodes) or a group with invalid concentrator selection (i.e. the concentrator is not in the group). For small networks (e.g. about 10 nodes) it is not a problem, since the total allocation for 10 nodes is $2^{10} \cdot 10$, i.e. 10Kbytes times the size of a float. The size grows exponentially though, and for 20 nodes the allocation required is 20Mbytes times the size of a float. We cannot solve such large networks to optimality anyway, so it is not a problem until heuristics are applied, where hashing will be used to reduce the amount of memory required.

It should be remarked that keeping results for primary groups makes no sense, since they cannot be reused (at least not entirely), which can be seen from the following. If the secondary groups are not exactly the same between instances, the minimum cost is not the same for sure. If the only difference between two solutions is the selection of concentrators, then the amount of flow between groups is the same, but the cost between concentrators is changed, hence the total cost changes.

Combined with limiting the allowed size of groups, this does in fact give a very high improvement in performance - see section 12.3.3.

## 9.5 Using the solution value for the HNP as bound in the NHNP

Since solving in the third and fourth phase consist of optimizing independent groups, it may be the case that the sum of solution values of some of the groups have such a high solution value, that the complete solution will not be optimal. In this case there is no need to solve the remaining groups.

This can be generalized, such that when one or more groups are solved we sum the solution-values and compare with the best known solution value for the HNP. This difference is used as initial bound for a remaining group, since if no better solution value can be attained for this group, the current group-division and concentrator-selection is not optimal.

This strategy may interfere with the above “reuse of calculated group-data”-scheme, but only if a solution is not found. In this case, we cannot be sure whether a solution exist, since we did not allow for investigating all possibilities, but if a solution is found it is optimal and can be reused as usual.

In the current set up, where all possibilities are tried in a random order (i.e. nothing is done to built promising groups), a substantial speedup cannot be expected, since good solutions are usually not found at first. Hence bounding should be used on partially divided groups and/or concentrator selections, since this can guide us towards a good solution initially.

The solution time for NHNP’s increase exponentially with the number of nodes in the network (see section 9.3), hence it may be beneficial to consider groups with few nodes first, since they can be computed fast.

Another strategy is to calculate the solution values for the secondary groups, and then use the bounding on the primary group only, since this solution value is never reused anyway. If this is done, a small performance improvement is achieved. This can probably be improved even more, if the groups with few nodes were calculated first, and the bounding on partial group-divisions and/or concentrator-selection were used. Possibly it is not necessary to bound, but only make sure that group divisions which look promising are generated first.

## 9.6 Non-Trees

Allowing for non-trees to be solutions increases the number of feasible solutions, and all previously feasible solutions are still feasible. Hence HNP's which had no feasible solution previously, may now have a feasible solution.

Allowing for non-trees to be solutions would not alter anything for the HNP part of the solution algorithm. Hence if a new solution algorithm for NHNP's, which allow for non-trees were developed, HNP's could be solved immediately as well.

# Chapter 10

# Heuristic Solution of the HNP

In this chapter a heuristic solution algorithm for the HNP is presented. The algorithm is based on simulated annealing (SA) and the solution of NHNP's. For an introduction to SA see e.g. [17].

## 10.1 The Algorithm

The basic idea of the algorithm is to generate an initial solution, and then modify the initial solution to obtain better solutions using SA. The SA algorithm is characterized by accepting solutions which have higher values than the currently best known solution (denoted worse solutions). As a generally accepted rule of thumb, between 30% and 70% of the worse solutions should be accepted when the algorithm is started (and temperature is high), and no worse solutions should be accepted when the algorithm finishes (and the temperature is low).

The SA algorithm is sketched in figure 10.1.

```
$current$ = Initial feasible solution
$best = current$
$t$ = Initial Temperature
do
   $t$ = update($t$)
   do
      $nbh$ = neighbour($current$)
   until feasible($nbh$)
   if eval($nbh$) < best($current$)
      $best = nbh$
   if eval($nbh$) < eval($current$)
      $current = nbh$
   elseif accept($t$, eval($nbh$), eval($best$))
      $current = nbh$
until stop-criteria
```

Figure 10.1: *SA algorithm for solving HNP*

The algorithm contains an inner loop which finds a feasible neighbour-solution and an outer loop which loops until the stop-criteria is true. The algorithm keeps two solutions, namely the best and the current solutions. The best solution should not be necessary, since the current solution should preferably end with the best solution anyway. This is however not always the case, usually because the selected parameters are not advantageous, sometimes also because of coincidences. Hence the best solution is used as a guideline for finding good parameters. The solution value of a solution is determined using the eval function.

The initializations consist of determining an initial solution and the initial temperature. The temperature is updated for each iteration using the update function. The temperature affects the chance of accepting a worse solution. This is determined by the accept function, which based on the solution values and the temperature determine if a neighbour-solution is accepted.

The algorithm is described in detail in the following sections.

## 10.2 Representation

A solution is represented as for the optimal case, i.e. two arrays, one giving group numbers and one specifying which nodes are concentrators. The only difference is that there are no limits on the representation involving the group numbers. Since we will not go through all possible solutions, there is no reason to do that.

Given an solution in this representation, $G + 1$ NHNP's can be generated as described in section 9.2.1. The representation contains nothing about flow. Instead information about flow is kept in each of the NHNP's. By saving each NHNP problem, flow information can be reused.

## 10.3 Neighbourhood

Two types of neighbour-solutions are used:

**Concentrator-Neighbour** Select another concentrator in a group
**Group-Neighbour** Move non-concentrator node to another group

The number of neighbour-solutions is large - there are $|V| - G$ concentrator-neighbour-solutions, since this is the number of non-concentrator nodes. The number of group-neighbour-solutions is $(|V| - G) \cdot (G - 1)$, since each non-concentrator node can be moved to any of the other groups. Too many neighbour-solutions exist hence the neighbourhood is limited.

### 10.3.1 Limiting Neighbourhood

To reduce the number of possible neighbour-solutions to the ones that seems most likely to be better, we limit the number of group-neighbour-solutions. The solution possibilities we would like to get rid of are those that have groups, which are scattered in space, since those kind of groups usually have high setup-costs.

For a given group the nodes to move to the group is limited to be only the $\lceil |V|/G \rceil$ nodes, which are closest to the concentrator node of the group, i.e. has the lowest setup-cost to the concentrator. Also the nodes considered are in another group and are not concentrator nodes. This limit the neighbourhood to contain solutions only, which seem to be close and thus reduce setup-cost. Since there are $G$ groups this amount result in up to $|V|/G \cdot G = |V|$ group-neighbours, which is roughly as many as there are concentrator-neigbours. Thus the two types of neighbour-solutions have roughly the same "priority".

This also limits the number of solutions considered, since solutions with secondary edges between widely separated nodes (i.e. high-cost edges in between) will never be generated except if generated as initial solution.

### 10.3.2 Cycling Neighbourhood

The neighbour function returns a neighbour-solution, to the current solution. The returned neighbour-solution depends on the previous invocations of the neighbour function. The neighbour solutions are cycled, i.e. first the concentrator-neighbours are considered and then the group-neighbours are considered, but across invocations of the neighbour function.

In each of the two groups of neighbour-solutions, cycling is also done. The concentrator-neighbours are simply considered in turn starting at the lowest numbered node. If this node is not concentrator it is made concentrator, otherwise the next node is considered until a non-concentrator node is found which is made concentrator. The state is saved such that when the function is invoked the next time, the node considered is the node following the node which was made concentrator. In each cycle a total of $|V| - G$ concentrator-neighbours are considered.

The group-neighbours are cycled in groups, such that one group is considered during one invocation of the neighbour function. Next time the next group is considered an so forth. This is done $|V|/G$ times for each group, such that a total of $V$ group-neighbours are considered in each cycle.

During one invocation, the $\lceil |V|/G \rceil$ nodes closest to the concentrator node of the group and not already in the group are found, and one is picked at random. Picking one at random seems to work better than finding the best valued or closest, which will limit the neighbourhood searched even more.

An alternative to cycling the neighbourhood as described above, is to pick the neighbour solution at random from all the possible neighbour-solutions. Doing this is simpler, the runtime is similar and a small improvement in the solution quality has been measured. Hence this should probably be done instead. The quality of the solutions are measured in section 12.5.5.

### 10.3.3 An Alternative Neighbourhood

Another type of neighbour-solution is to swap nodes between groups, hence maintaining the size of groups. Every non-concentrator node can be swapped with any other non-concentrator node where the two nodes are not in the same group, hence there are $\binom{|V|-G}{2}$ neighbour-solutions except for the pairs of nodes which are in the same group. If groups are of equal size there are $G \times \binom{|V|/G-1}{2}$ of them, which for $|V| = 100$ and $G = 10$ result in 3645 neighbours. This is too many solutions to consider, hence the neighbourhood must be reduced.

This reduction can be achieved by limiting the swaps considered to only one swap for each pair of groups, selecting the nodes which are closest to the other group. This way the number of neighbour-solutions is reduced to $\binom{G}{2}$ and may be lower if one or more groups have size 1, in which case they do not have any non-concentrator nodes.

A modification of the mathematical model allowing for dividing the network into any number of groups, would require us to change the neighbourhood, such that the number of groups could change. This can be achieved by having neighbour-solutions, of groups which were merged or split.

## 10.4 Evaluation Function

The evaluation function used is the objective function value, calculated group-wise, that is the objective value is the sum of values of groups (see section 9.2.2). The feasibility check is also carried out the same way - each group is checked for feasibility, and if all groups are feasible, then so is the entire solution.

The chosen evaluation function does not allow for infeasible solutions. Hence the initial solution should be feasible. It may be a good idea to allow for evaluation of infeasible solutions, calculating the value e.g. as described in the following.

The infeasibility of a solution is due to one or more group solutions being infeasible. Hence feasible groups can contribute to the objective value as usual.

An infeasible group has at least one demand with value larger than the capacity of edges (see section 6.1). For each of those demands $ij$, we require demand $ij$ to use edge $ij$ as the only flow edge and ignore the capacity limit for that edge. Otherwise the group is solved as usual. The exceeded capacity result in a penalty, which relate to how much the capacity is exceeded and to the size of the group.

The cost should be so high that it does not pay off to end up with an infeasible solution but on the other hand so low, that it can be accepted during the SA algorithm, such that other parts of the solution space can potentially be reached.

Doing this would mean that it was not necessary to find an initial solution, since this could be done by the SA algorithm. There may be other benefits that cannot be immediately foreseen, but the modification has not been made. Instead the initial solution is found using algorithms described in the following section.

## 10.5 Initial Solution

In most cases an initial solution can be easily found using one of the methods described in the following. Hence the problem is merely that of finding an initial solution, which imply that the SA algorithm finds a good solution. In general this cannot be done and in particular it does not necessarily imply, that the initial solution should be of low value, but it may be beneficial. Hence this is what we will aim for when building the algorithms.

Finding the initial solution is done in two phases. The first phase finds a low cost initial solution, which ignore capacities, hence it may be infeasible. The second phase makes the solution feasible if necessary. The flow-cost is ignored, so in fact the solution is low cost with respect to setup-cost. If the flow-cost dominates the cost of the network, the described algorithms will probably not perform well.

The first phase is tried in three different variants, described in the following subsections followed by a subsection containing a description on how feasibility is obtained, i.e. the second phase.

The three variants all divide the network into equally sized groups (i.e. groups of size $|V|/G$), since this division tends to have least cost and also has a good chance of being feasible. If not, only few of the groups tend to be infeasible.

### 10.5.1 Random Initial Solution

A random initial solution is built using the algorithm in figure 10.2. The algorithm is mainly used for comparison with the other algorithms to see if there is any benefit from using these.

```
for $grp = 0$ to $|V| - 1$
    Choose one node $n$ at random, which is in no group
    Allocate $n$ to group number ($grp \pmod G$)
```

Figure 10.2: *Finding initial solution - Random*

### 10.5.2 Simple Low Valued Initial Solution

This algorithm builds groups in a greedy way to minimize setup-cost for the group considered at the moment. The algorithm is shown in figure 10.3.

```
for $grp = 0$ to $G - 1$
    Choose one node $c$ at random, which is in no group
    Allocate $c$ to group $grp$ and select $c$ as concentrator
    do
        Find node $j$ closest to $c$ which is in no group
        Allocate $j$ to group $grp$
    until $|V|/G$ nodes are in group $grp$
```

Figure 10.3: *Finding initial solution - Simple*

First a previously unselected node is picked at random and made concentrator. Then the nodes which are closest to the concentrator, are put in the same group as the concentrator. This is not the same as the group having the minimum setup-cost, which would rather be found by building a minimum spanning tree algorithm, but it is a good estimate.

The groups which are built first has the best chance of having a low setup-cost, since they can chose from more nodes to be included in the group. On the other hand, the groups built last may not be able to find close nodes (i.e. nodes with low setup cost to the concentrator), since these may already be allocated to other groups. This is what the next variant of the algorithm addresses.

### 10.5.3 Find Low Valued Initial Solution using Assignment

This version of finding a low valued initial solution, consist of three phases:

- Select (initially) concentrator-nodes as widely separated nodes
- Assign remaining nodes to the concentrators
- Reselect concentrator nodes as close nodes

As opposed to the previous version, we try to minimize the setup-cost of the groups in general, not just for the group we are working with at the moment. The hope is of course that this will give a better initial value,

### 10.5.4 Selecting Initial Concentrators

The nodes selected as concentrators are nodes, which are widely separated. In general it does not make sense to choose concentrators as separated nodes, but initially they are only used to identify a starting point for building the groups. The two concentrators chosen first are the two nodes which are separated the most, i.e. the two nodes that define the diameter of the network. The next node selected as concentrator is the node that has the longest distance to the already selected concentrators on average. This is continued until $G$ nodes have been selected as concentrators

### 10.5.5 Assignment of Nodes to Groups

The selected concentrators represent a group each. Remaining nodes is distributed evenly between the groups, i.e. the groups should contain $|V|/G$ nodes each if $|V| \equiv 0 \pmod g$, or at most $\lceil |V|/G \rceil$ nodes if $|V| \not\equiv 0 \pmod g$. This is denoted the *group-size*.

The problem is solved as an assignment problem, assigning each non-concentrator node to a concentrator (and hence indirectly a group). A standard assignment problem assigns exactly one facility to exactly one location. In order to have equal many facilities and locations we copy the concentrators, such that there is *group-size* $- 1$ number of copies of each concentrator.

There should be equally many non-concentrator nodes and since we have $G$ groups, there should be $G$ times the *group-size* $- 1$ non-concentrator nodes. There may be fewer if $|V| \not\equiv 0 \pmod g$, in which case we add dummy nodes. The dummy nodes can be assigned to any concentrator for free. Since more dummy variables can be assigned the same concentrators, the size of some groups may be much less than the upper bound on the size of groups. This is not a problem though, since feasibility depends mostly on that no large groups are generated, and if groups are made small it is because it pays of.

The prices we use in the assignment problem are the setup-costs between concentrators and nodes. This way the total distance between concentrators and nodes are minimized, and not, as would be preferred, the total cost, i.e. the sum of costs of the MST's of each group.

### 10.5.6 Reselecting Concentrators

Given the group division, we reconsider the choice of concentrators. The initial choice of concentrator was made such that the concentrators were separated. This will have a high setup-cost for the primary group, hence a new selection is made.

The concentrators selected should be close in order to minimize setup-cost, hence we start out by choosing the two nodes, which are closest, but are in different groups as concentrator nodes. Continuing from this selection, we select the node which are not in any of the groups of the already selected concentrators, with the least setup-cost distance to the concentrators on average. This is continued until $G$ concentrators are selected.

### 10.5.7 Modifying Solutions to obtain Feasibility

If a solution is infeasible, at least one group is infeasible. If the primary group is infeasible, nodes with high demand between them are moved such that they are in the same group. This will reduce the demand flowing in the primary group. If a secondary group is infeasible, it may be the case that selecting another concentrator will solve the problem. This is so if a single node has a high demand out of the group, hence making the node the concentrator will relieve the highest loaded edge. If this is not enough to ensure feasiblity of the group, one or more nodes are moved to another group to decrease demand in the group.

The algorithm for making the primary group feasible is shown in figure 10.4.

Given the group division and concentrator selection, the required demand for the primary group is calculated as described in section 9.2.1. Then a node pair in the primary group is identified, which has higher demand than the capacity of primary edges (if such one exists). As mentioned in section 6.1, if the primary group is infeasible such a demand exists and vice versa. Hence checking whether the primary group is feasible and if not, finding a node pair with too high demand is combined and done by checking that all demands in the primary group does not exceed the capacity.

```
while primary group is infeasible
  Find node pair $i$ and $j$ which has a demand that is too high
  $i$ and $j$ represents two secondary groups, $A$ and $B$
  Find the node $k$ in either $A$ or $B$ (assume it is in group $A$),
    which has the highest total demand
    to nodes in the other group (i.e. $B$)
  move $k$ to group $B$.
```

Figure 10.4: *Making primary group feasible*

If a node pair $i$ and $j$ is found in the primary group with demand larger than the capacity, the two secondary groups they represent are identified, denoted $A$ and $B$. The nodes in $A$ and $B$ are considered. For the nodes in group $A$ the required demand to nodes in group $B$ is measured. This is done symmetrically for the nodes in group $B$, and the node with the highest demand is identified. This node is moved to the other group (e.g. if the node is in group $A$ then it is moved to group $B$ and vice versa).

In most cases, this relieve the amount of demand on edges, but the algorithm may loop forever moving the same node back and forth between the same two groups. This has not been a problem, but if it shows to be a problem, this can be easily solved by introducing some kind of randomness in the algorithm, such that it is not necessarily the node with the highest demand which is moved, but one of the nodes with a high demand.

When the primary group is feasible, we try to make the secondary groups feasible if necessary. This algorithm is shown in figure 10.5.

```
while any group infeasible
  Make primary group feasible
  For each node find demand out of secondary group
  Find secondary group which is infeasible
  Find the node with the highest demand out of group - node $i$
  if $i$ is not concentrator
    Make $i$ concentrator
  else
    Find the non-concentrator node with the highest demand
      out of group - node $j$
    Move $j$ to other group selected at random
```

Figure 10.5: *Making secondary group feasible*

Selecting which group to move a node to is chosen at random, but involves the size of the group. The chance of choosing a given group is inverse proportional to its size such that the chance of creating large groups is small.

Each iteration starts out by ensuring that the primary group is feasible. Then demand out of groups is calculated, and a group is identified which is infeasible. In this group the node with highest demand out of the group is considered. If this node is not the concentrator, it is made concentrator. Measuring the demand out of the group is roughly the same as finding the highest demand in the group assuming demand out of the group is in general more significant than internal in group. Hence selecting the concentrator to this node will relieve the group of one of the highest demands.

If the highest demand node is already concentrator, the node with the highest demand, which is not the concentrator is moved to another group chosen at random as described above. This remove the highest demand node pair from the group.

Termination is ensured by limiting the number of changes which are allowed to 500. Running 500 iterations takes fairly short time, and it seems to be enough to find a feasible solution if one exists. If none can be found this is simply reported.

### 10.5.8 Concluding Remarks on the Initial Solution

In fact it does not matter much which scheme is chosen, the price difference is not high on the final solution found using the SA algorithm, though the initial solution values may differ much (see section 12.5.2). The initial solution values are compared with the final solutions in section 12.5.1.

Yet another alternative has in fact also been tried. This algorithm starts out by minimizing the primary setup-cost, and builds groups using minimum spanning trees. The groups built have unequal size, and hence the chance of finding an initially infeasible solution is large. In fact the found solution may be useless since the second phase basically breaks down the good valued solution to obtain feasibility. Hence unless the demand is so low that finding an initial solution is not difficult, this alternative version does not find good initial solutions.

## 10.6 Accept Function

The accept function determines whether a worse solution should be accepted or not. The chance of whether a neighbour-solution is accepted depend on the temperature. The SA algorithm starts out at a high temperature and lowers the temperature as it runs, hence lowering the temperature should lower the chance of accepting a worse solution.

The difference in solution value between the current solution and the neighbour-solution value also influence the chance of accepting a worse solution. If the difference is low, there should be a higher chance of accepting a worse solution than if the difference is high.

In [17] it is suggested that the function in equation 10.1 is used compared with a uniformly distributed number between 0 and 1. If the calculated value is larger than the randomly found number, the neighbour-solution is accepted.

$$p = \frac{1}{1 + \exp(\frac{val}{temperaure})} \tag{10.1}$$

We require *val* and *temperature* to be positive, since this gives function values between 0 and 1. *val* is the calculated difference between the solution values in percent as given in equation 10.2.

$$val = \frac{\mathsf{eval}(nbh) - \mathsf{best}(current)}{\mathsf{best}(current)} \tag{10.2}$$

The difference is calculated in percent of the solution value to avoid dependence on the costs in an instance of the HNP. The calculated percentage does depend on the number of nodes, since neighbour-solutions consisting of moving one node or selecting a new concentrator in one group influences the objective value relatively more in a network with few nodes than in a network with many nodes.

Equation 10.1 has the desired characteristics - if *val* is increased, the function value decrease, and it has a lower chance of being greater than the randomly calculated number and thus accepting a worse solution. Also as temperature is lowered, the function value decrease, and hence there is a lower chance of accepting a worse solution at lower temperatures.

### 10.6.1 Random Generation

The random generation used is the standard `Random.h` library, which if fed with the same seed produces the same sequence of random numbers between runs. The seed used is constant in order to be able to reproduce results which facilitates easy debugging.

## 10.7 Stopping Criteria

The stopping criteria used is a sliding window stopping criteria, i.e. when no new solution has been accepted in 200 iterations, the algorithm is terminated.

In some situations it has been the case that the inner loop in the algorithm (see figure 10.1), which finds a feasible solution cannot find any feasible solution. This situation arises only if the total demand in the network is high, and also seems to require that the group-neighbours are limited as described in section 10.3.1. Given this, the algorithm will never get out of the inner loop, and hence will not terminate.

To make sure the algorithm terminates we count the number of infeasible solutions found, and if this exceeds 500, we simply stop the algorithm. An alternative would be to not limit the group-neighbours, in which case more solutions can be reached and hence there is a better chance of finding feasible solutions. This could be done dynamically, if more than e.g. 100 neighbour solutions had been tried and none were feasible. The problem does not arise often though, so this has not been tried in practice. However it would be a good idea to implement and easy it seems.

The total number of iterations (in the outer loop) is counted, and usually it is in the order of 3000.

## 10.8 Initial Temperature

Choosing initial temperature and cooling rate in general which gives good results is not easy. Usually a particular problem instance is investigated and the parameters are adjusted accordingly. Since we test for many instances this is not a possible way to do it, thus we find the parameters for problem instances in general.

Care have been taken to make sure that the choice of temperature does not depend on the edge costs, i.e. if the same factor is multiplied on each edge cost, using the same temperatures should give the same results. This is achieved by calculating the cost difference between the current- and the neighbour-solution in percent rather than real costs in the accept function (see equation 10.2).

The number of nodes in the network, though, influence the temperature, since the number of nodes influence the calculated deviation. For e.g. small networks moving a node from one group to another may affect the objective function value relatively more than moving a node between two groups in a large network, since less of the network is potentially affected.

Thus for large networks, the initial temperature should be lower than for small networks, since the calculated *val* is probably lower for larger networks (see equation 10.1).

As mentioned a general rule of thumb is that at the initial temperature, between 30% and 70% of worse solutions should be accepted, hence this is aimed for. Given the accept function, experiments show that this is the case at approximately the temperature $0,75 \times |V|^{-0.25}$, so this value is used. The initial temperature is depicted in figure 10.6 as function of the number of nodes in the network.

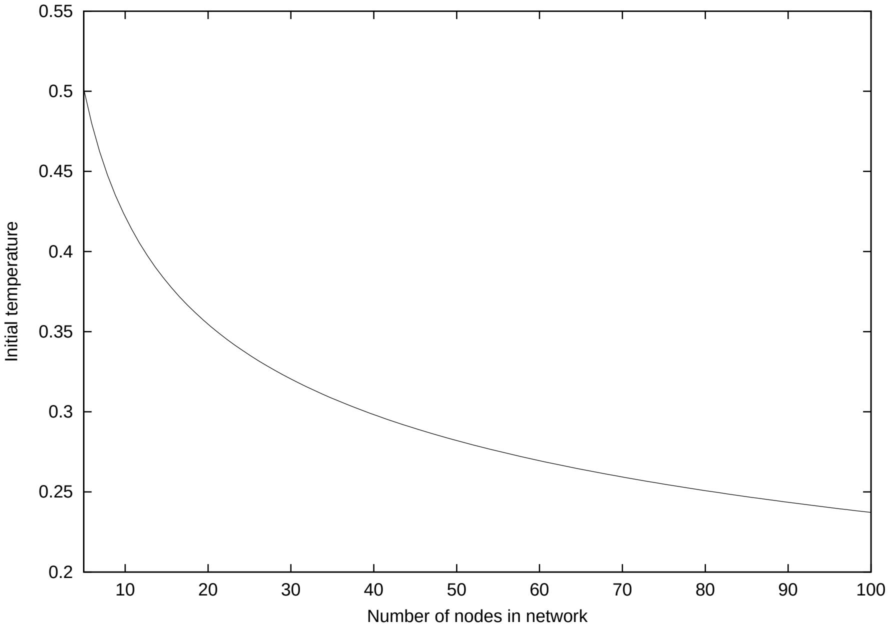


Figure 10.6: *Initial temperature as function of the number of nodes in the network*

The figure shows, that using this function to determine the initial temperature, the initial temperature is lower for large networks as wanted.

Experiments have been carried out modifying the initial temperature, but there does not seem to be any general improvement by lowering or raising the coefficient or the factor in general. If modified dramatically, such that e.g. the temperature is close to zero or very large (in reality local search and random search respectively) the performance drops.

In fact the temperature also depends on the number of groups, but no meaningful dependence have been identified.

## 10.9 Cooling Rate

The cooling rate or update function is chosen to be a factor, by which the temperature from the last iteration is multiplied. The factor should be low such that random walk in the search space is allowed, but also there is no reason for wasting time checking solutions totally at random. The factor is constructed as a function of the number of nodes, since as described above the temperature depends on the number of nodes.

The factor is chosen such that it allows for approximately 1000 iterations to be run when considering 10 node networks, and 3000 iteration to be run when considering 100 node networks. Experiments show that this is approximately at the point where increasing the number of iterations does not give any improvement in solution quality, and if the number of iterations is decreased (at least for the large networks) the solution quality decrease as well.

The factor used is $0.9978 - 0.0015 \times |V|^{-0.5}$. The update factor is depicted in figure 10.7 as function of the number of nodes in the network.

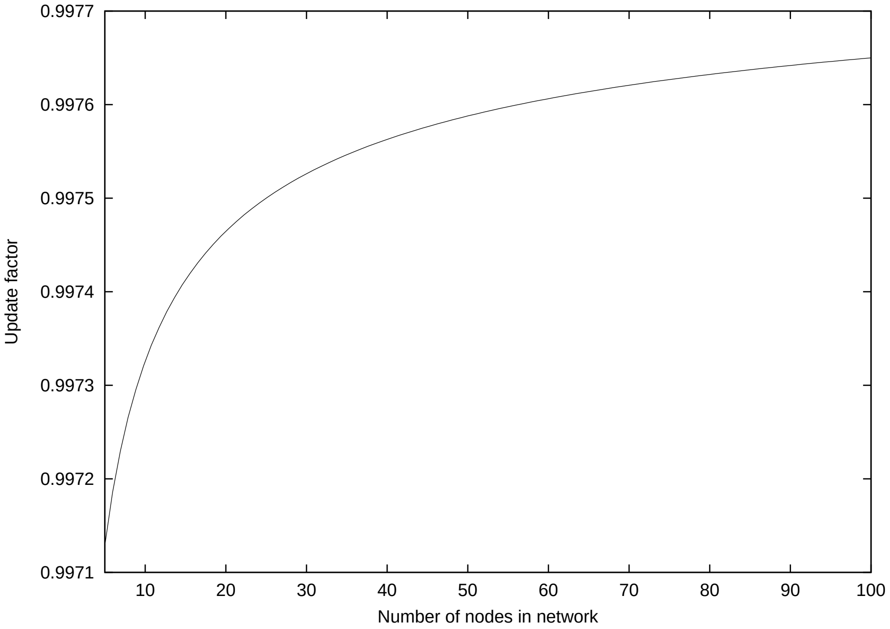


Figure 10.7: *Update factor as function of the number of nodes in the network*

The figure shows, that using this function to determine the update factor, the update factor is larger for networks with many nodes than for networks with few nodes. Hence the temperature decrease slower for networks with many nodes than for networks with few nodes. In fact tests indicate that the temperature decrease for networks with many nodes should be even slower, hence the update factor should be increased for networks with many nodes (see section 12.5.6).

## 10.10 Saving Solutions for Secondary Groups

The solution value of a secondary group depends only on what nodes are in the group and which node is the concentrator, it does not matter how the rest of the nodes are divided, and which nodes are concentrators in other groups. For HNP's which are to be divided into at least 3 groups, the neighbour-solutions of a solution has at least one group in common with the solution. Therefore there is no need to recalculate the solution value of at least this group - the solution value can be reused.

The Simulated Annealing algorithm may also return to previously considered solutions due to the built in possibility of accepting worse solutions. In this case it is also possible to reuse values for calculated groups.

The number of groups generated is fairly small - checking one neighbour-solution will generate either one (in case of concentrator change) or two (in case of node move) new groups, and in some cases the groups are known already. If around 3000 HNP solutions are considered (which is realistic in the current implementation), the maximum number of groups is 6000.

As mentioned in section 9.4 we cannot simply index all possible groups into an array, since the size is too large. Instead hashing is used to save all calculated solution values. Open hashing is used, i.e. a linked list of elements is associated with each index in the hash-table, and since the hash-table it self does not take up much space (each element is a 32 bit pointer), we allocate more than enough space (i.e. 20000 elements) to reduce the chance of hitting the same index.

Space is not everything, it is also important, that the hash-function chosen does not give the same hash-values for the groups considered. Instances are represented as binary arrays of length |V|, which contain 1 if a node is in the group and 0 otherwise. The concentrator, $c$ of the group is also used, i.e. a number between 0 and $G-1$. The hash function used is shown in equation 10.3.

$$n \cdot G + c \pmod{hashtablesize} \tag{10.3}$$

n is the binary array read as a number and c is the concentrator number.

The efficiency of the hash-function has not been measured systematically, but for a few examples the number of conflicts has been measured, and in the examples it works well, even though more than $hashtablesize/2$ elements are put in the hash-table. Few equal hash-values are computed and the hash-values computed more than once, are not generated more than a few times.

# Chapter 11

# Tools and Data-files

Generation of random HNP's, solution of the HNP's by CPLEX or the developed HNP solvers and then finally visualizing the found solutions graphically have required development of tools. The tools work on files containing data for HNP's. The tools and data-files are described in this chapter. In addition to this, tools (scripts and make files) have been used, e.g. for ease of development and performance tests. These tools will not be described any further.

## 11.1 Overview

The files all describe one instance of the HNP, and are identified by their extensions. The files are:

`.ran` Foundation for generation of HNP, i.e. number of nodes and groups and amount of total demand.
`.xy` HNP described by points in a x-y grid and demand between each pair of nodes.
`.net` HNP described by setup-cost, flow-cost and demand between any pair of nodes.
`.lp` Input file for CPLEX, which contains constraints.
`.mst` Solution file output from CPLEX, giving values of each decision variable.
`.sol` Solution file giving objective value, primary and secondary edges and for each edge the flow on the edge.

The developed tools are the following:

`RanToXy` Generate random HNP problem in `.xy` format following specification in a `.ran` file.
`XyToNet` Convert HNP problem described by a `.xy` file to a `.net` file.
`NetToCplex` Convert HNP problem described by a `.net` file to a `.lp` CPLEX input file.
`MstToSol` Convert a solution-file generated from CPLEX to a `.sol` solution file.
`ShowGraph` Given a `.xy` file and a `.sol` file, draw graph in a x-y grid.

The tools are implemented using Java. The data-files-flow and the tools are depicted in figure 11.1.

Data-files are in rectangles and tools are in rectangles with rounded edges. The tools `SolveExact` and `SimAnn` are the developed solution programs, which solve the HNP optimally and heuristically using simulated annealing respectively. CPLEX version 7.0 is used.

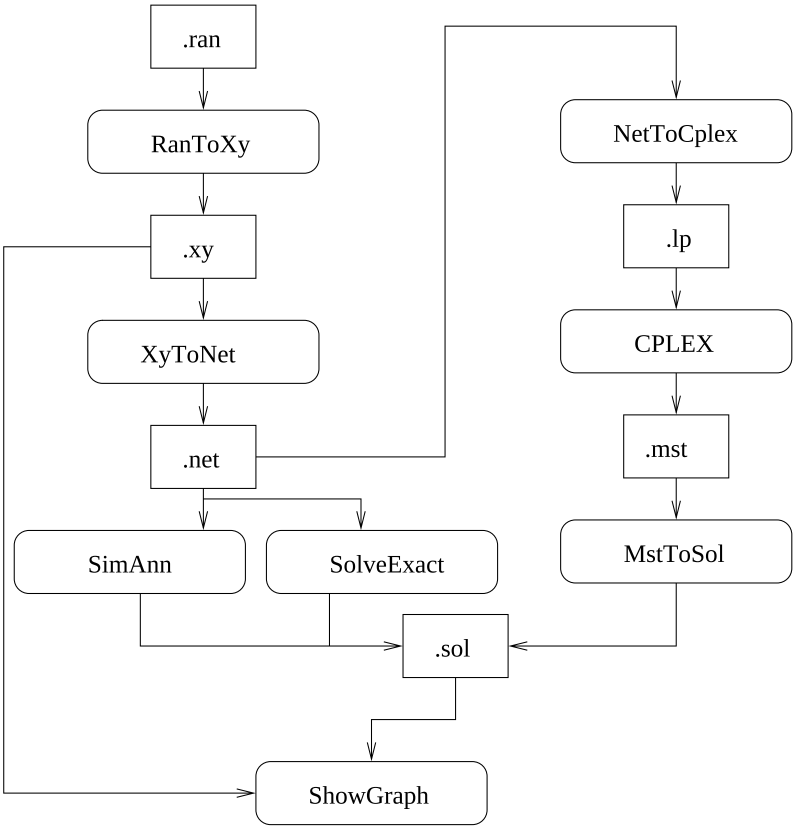


Figure 11.1: *Tools and data-files*

## 11.2 Graph Generator

The `RanToXy` tool generates a HNP instance by setting up a x-y grid, and placing nodes randomly in the grid. Nodes cannot be in the exact same position in the grid, hence if a node is already where a new node is to be placed, a new position is generated.

The total demand in the network is controlled by a parameter(denoted the wished total demand) supplied in the `.ran` file. Initially the demand is generated by first generating a number giving the amount of demand each node generates. Then demands are assigned values equal to the sum of the two endpoint nodes of the demand. The total sum of demand in the network is calculated as well. This calculated total demand is not equal to the wished total demand, hence we multiply each demand by the same factor, such that the total demand is as wished. The factor is the wished total demand divided by the calculated total demand.

Other schemes are possible, e.g. completely random, i.e. each parameter (e.g. secondary flow-cost between two nodes) with no relationship to e.g. secondary flow-cost. Also demands can be generated only for some of the node pairs, but for telecommunication networks it seems fair that all nodes communicate with each other.

The grid based graph generator has mainly been chosen for its simplicity and its ease of visualization. The question is whether the networks generated are representative for the networks which are to be solved for real world applications. Telecommunication networks are switches and cables in a plane, but the setup-cost does not depend only on the distance, i.e. cost of digging down a cable, but also on buying communication equipment, which can handle communication on cables. The price of communication equipment is certainly not linear in distance but would rather be modelled as a fixed cost independent of distance.

For the carried out tests the grid based graph generator fully suffices, regardless it does not model telecommunication networks exactly.

## 11.3 HNP Files

The `.ran`, `.xy` and `.net` files describe a HNP. They are text files, where each line contain a string explaining what data is next and the data. The order of the lines is important, the first string of each line is ignored by the tools reading and writing the files, it is only there to allow humans to read and modify the files.

The three files all contain the number of nodes, number of groups, capacity of primary and secondary edges and the maximum number of nodes in each group. The `.ran` files additionally contains the size of the grid to place the nodes in, minimum and maximum demand and cost for a unit-distance of primary and secondary flow- and setup-cost.

The `.xy` files contain the generated demand and the position of nodes, instead of a minimum demand and a maximum demand.

The `.net` files contain a description of networks, which do not require a grid, instead the setup-cost and flow-cost for primary and secondary edges are specified for each pair of nodes.

The grid description is suitable for visualizing networks, since distances are proportional to the setup-cost and the flow-cost (and primary and secondary costs are also proportional), but they also prevent the description of networks where this proportionality does not exist. In general this is not preferable, hence the solution algorithms do not use the grid description, but the general description.

## 11.4 `CPLEX` Solution

If CPLEX is used to solve HNP's, `.lp` files, listing objective function and constraints are required. `.lp` files and hence the constraints are generated by `NetToCplex`.

`CPLEX` allows for outputting a `.mst` file, which gives the value of each decision variable. This file can be converted to a `.sol` file, which contain an extract of the `.mst` file giving objective value and primary and secondary edges. This is done by `MstToSol`.

## 11.5 Solution Files

Solution files `.sol` contain objective value, primary and secondary edges the amount of flow on each edge. Since the path for each demand is not stored in the file, the values of the flow-variables cannot in general be determined. If the solution is a tree, the paths can be easily found, since paths are unique in trees. If the solution is not a tree, additional information is needed to construct the paths. At runtime paths are constructed for each group, hence this information can be recorded, in fact at the moment it is possible to get this information for the final found solution written to the screen. From this the full paths can be constructed as described in section 3.3.

## 11.6 Visualizing Solutions

`ShowGraph` is used to visualize solutions. `ShowGraph` requires a `.xy` file giving the coordinates of each node in the grid and a `.sol` file. Hence the solution is drawn by setting up a grid, placing nodes, and drawing primary edges in red, and secondary edges in black. The primary nodes (concentrator nodes) are all nodes with an incident primary edge, and since each group contains exactly one concentrator node, a group can be identified by finding all nodes, which can be reached by following secondary edges starting out at a concentrator node.

# Chapter 12

# Performance Tests

## 12.1 Problem Instances Used

An HNP instance as defined by a `.ran` file determines number of nodes, number of groups, capacity of primary and secondary edges, primary and secondary flow-cost and setup-cost for edges relative to the edge length and a value determining the total amount of flow in the network.

The generated problem instances are mostly based on how telecommunication networks works (described in section 2). But since flow-costs are zero in telecommunication networks, and flow-cost is included in the definition of HNP, flow-cost will be included in the problem instances used. For equipment cost and “digging down a cable” cost in telecommunication networks, it seems that only “digging down a cable” depends on the distance. In the generated problem instances, setup-costs are proportional to the distance.

All the generated problem instances use the values in table 12.1 unless otherwise stated.

| **Parameter** | **Value** |
|---|---|
| Primary edge capacity | 400 |
| Secondary edge capacity | 100 |
| Primary setup-cost for unit distance | 400 |
| Secondary setup-cost for unit distance | 200 |
| Primary flow-cost for unit distance | 1 |
| Secondary flow-cost for unit distance | 2 |
| Grid size - X | 100 |
| Grid size - Y | 100 |

Table 12.1: *Parameters for the data set*

First we remark that the two grid parameters only influence the objective value of the network problem not the problem it self. The grid is used for calculating distances, and since all costs are relative to the distance, modifying the grid will for a given solution change the value of the objective function but not the solution. The parameters are set to 100.

The capacity values relate to the demand and the flow-cost. If the capacity is increased by a factor, the demand is increased by the same factor and the flow-costs are decreased by the same factor; the problem only differs by the same factor in the objective value. The ratio between the edge capacities, however is fixed to 4, since this is the case for telecommunication networks. Capacity is set to 400 for the primary edges and 100 for secondary edges.

The setup-costs and flow-costs are chosen, such that they have equal impact on the objective function value. Also, since the ratio between equipment costs on different levels is 2 in telecommunication networks we will use this for the setup-cost as well (primary is highest), and the ratio between the flow-costs is also set to 2, but here the secondary is highest. We set the primary flow-cost to be 1 and the secondary flow-cost to be 2.

Given these values, the setup-cost contributes to the objective value with at least the same amount as the flow-cost. This is so since to use an edge the setup-cost of the edge has to be paid, and the maximum amount of flow-cost to pay is the capacity times the flow-cost, which is exactly the value of the setup-cost in both the primary and the secondary case.

We therefore have three parameters to control, namely the number of nodes, the number of groups and the maximum demand. The number of groups is set to approximately $\sqrt{|V|}$, usually a bit below.

The total amount of demand a network can carry depends heavily on the number of nodes, so it does not make sense to test networks with different number of nodes with the same demand. Instead we create three groups, where the total demand is light, medium and heavy respectively. The groups are found by experimentally generating networks with different number of nodes, and different amount of demand.

The medium group should have a demand amount, such that tree solutions exist, but are not found by the heuristic. For networks which can be solved optimally, the optimal tree solution is found to ensure that one exists. For networks with more nodes, the heuristic solutions are investigated, and the total demand is selected such that heuristic solutions are not trees but relatively few extra edges are in the solution.

The found solutions are plotted and smoothed, which gives the demand for the medium group. The demand of the light group is 50% of the medium group, and the heavy group has a total demand value of 30% more than the medium group for networks with 4-14 nodes, 40% higher for networks with 15-40 nodes and 50% more for networks with 45-100 nodes. The demand is reduced below 50% for networks with few nodes, since otherwise the heuristic cannot find feasible solutions at all.

The test instances generated have between 5 and 100 nodes, and instances are for few nodes generated with gaps between number of nodes of only 1, ending with gaps of 10 at the 100 node instance. More networks with few nodes are generated since they are the ones which can be solved optimally, and for the heuristic they do not take much time to solve.

The total demand for the groups are depicted in figure 12.1.

We test the NHNP algorithm by it self, hence we need to generate NHNP instances. This corresponds to HNP problems where the number of groups is 1. Hence we reuse the above generated data-sets, but require the number of groups to be 1. The generated instances have between 4 and 20 nodes, in gaps of 1. The demand is depicted in figure 12.2.

Using the same demand amounts as for the HNP's may be a problem, since the demand amount were chosen such that network with hierarchies contained trees. Since we do not have any high capacity edges in the tested NHNP problem, the demand may be too high. On the other hand we have more possibilities of edges so a feasible solution probably exists when allowing non-trees but tree solutions are unlikely to exist. The tests we carry out are tests of the heuristic solution algorithm where non-trees are allowed, hence this should not be a problem.

Generating the networks is done by `RanToXy` described in section 11.2.

Usually we generate 5 instances with the same number of nodes, groups and demand, i.e. the same `.ran` file is used. The results reported are averages on the 5 instances.

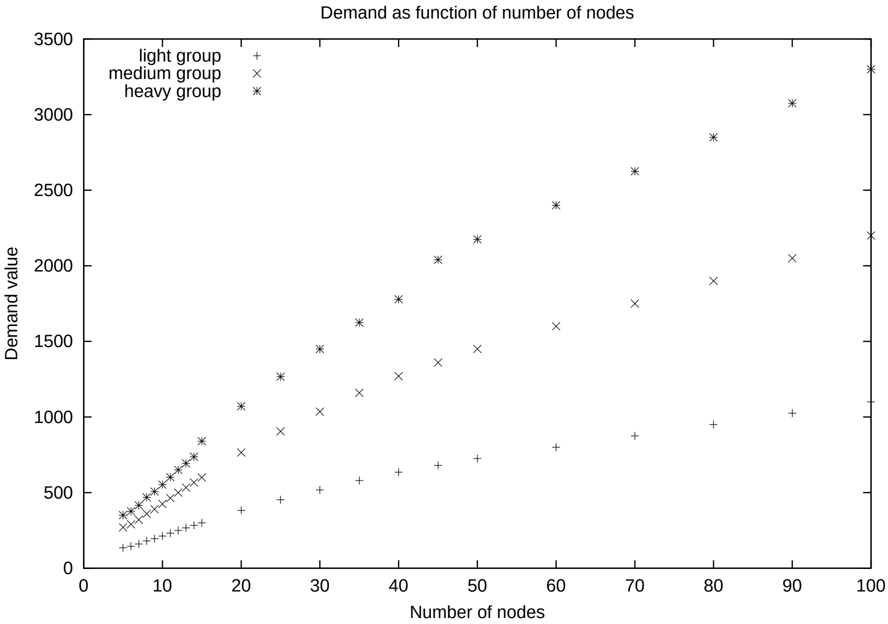


Figure 12.1: *Amount of demand in networks for test of HNP*

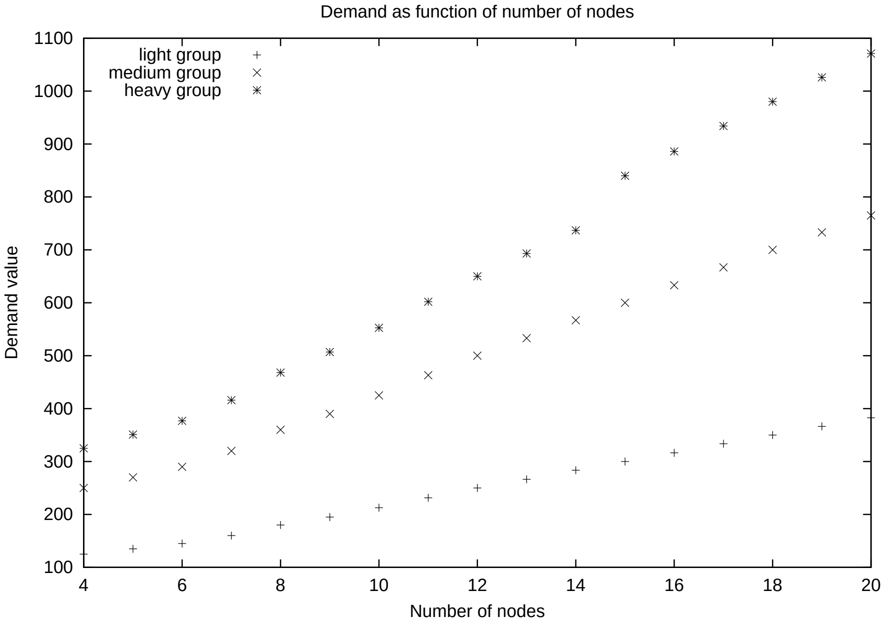


Figure 12.2: *Amount of demand in networks for test of NHNP*

## 12.2 Testing

The tests are divided into four sections, test of the optimal solution algorithm, test of the heuristic NHNP solution algorithm, test of the heuristic HNP solution algorithm and finally comparison of heuristic solution values with the optimal solutions.

The different versions of the algorithms are in most cases implemented by using `#define`'s. If e.g. a particular bound is used in place of another, this can be controlled by defining a single parameter and recompiling.

Times has been measured by calls to `getrusage`, which gives the system time and user time spent by the program as opposed to real time. The user time is reported, which is by far the most significant. The reported time include everything the program does, including output of results and log-generation. Log generation is, however set to a minimum.

Tests are run on a SUN Blade 1000 with 2 750 MHz processors and 2Gb ram or on another SUN machine with 12 processors and 12Gb, which seems to solve problems approximately 5 times slower than the SUN Blade (Exact information is unfortunately not available). For a given test-run the different instances are of course run on the same machine, but results cannot necessarily be compared across test-runs. In most cases the heuristic algorithm has been run on the SUN Blade and the optimal algorithm has been run on the slow machine.

The programs do not use much ram - in most cases below 5Mb. The machines were used by other people while testing, but since the time measured is the CPU time used, this has little (if any) influence.

The tests are run from what is considered normal, i.e. as described in the previous chapters. If deviating from this, it will be stated explicitly.

In most cases we will run different versions of the algorithm, which may and may not find different solutions. The solution with the best value is used as reference point, and comparison is then done with respect to this best found solution. Hence at least one of the versions of the algorithm will have a deviation equal to 0, though better solutions probably exist.

## 12.3 Performance Tests of the Optimal Solution Algorithm

This section contains tests of the optimal solution algorithm. In most cases, the maximum number of nodes in networks are 9 or 10. In practice it is possible to find solutions for networks with more than 10 nodes, but since we run tests on many instances the time required to solve the problems is substantial and does not contribute noteworthy to the discussion, hence it has not been done.

### 12.3.1 Bounds

To measure the effect of using bounds, different versions of the optimal solution algorithm have been run on the standard test-sets. The test instances have been run in the standard setup up, where all bounds are used, and in a version where the setup bound is not used and a version where the flow bound is not used (see section 5.5).

When the tests are run, the number of `Net` objects generated is counted and the runtime is measured. A branch creates up to two `Net` objects, and each is bounded, hence this is the number of times the bound value is calculated. This count indicates how good a bound is, but if the bound takes too much time to compute, the total runtime may increase though the number of `Net` objects decrease.

In figure 12.3 the runtime of the normal algorithm is compared with the runtime of the algorithm without use of the setup-bound.

Surprisingly there is only little improvement. The number of `Net` objects is only reduced with 1%, hence no major improvement is achieved, but the bound is fast to compute so the runtime is roughly the same regardless of if the bound is used.

In figure 12.4 the runtime of the normal algorithm is compared with the runtime of the algorithm without use of the flow-bound.

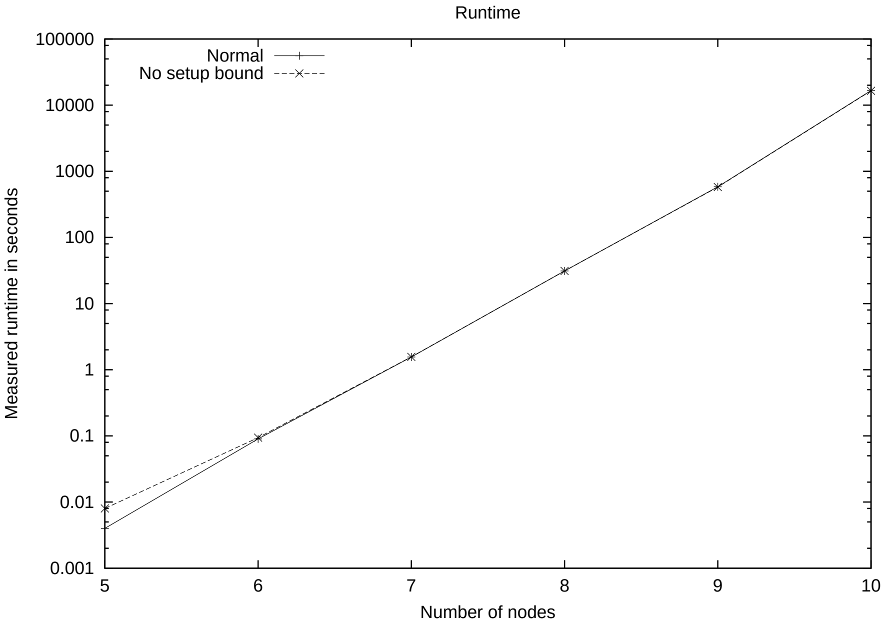


Figure 12.3: *Runtime with and without using setup bound*

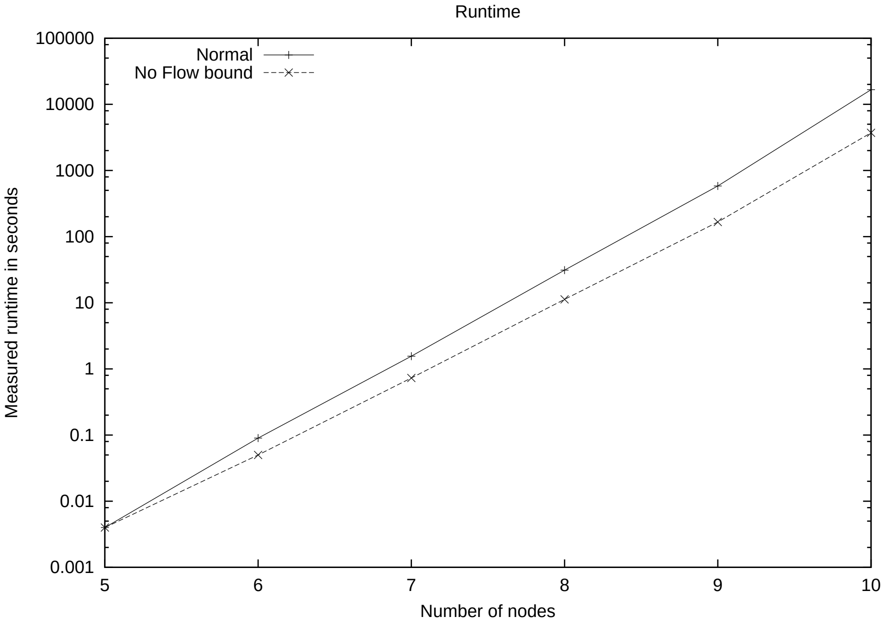


Figure 12.4: *Runtime with and without using flow bound*

Using the flow bound increase the runtime severely. The number of generated Net objects is reduced by only 3%, hence the gain from the reduced number of objects is so low, that it does not even come close to make up for the time spent on calculating the bound. This comes as a surprise - in fact early performance tests showed, that the flow-bound contributed to an improve in the runtime.

### 12.3.2 Reuse of Group Solutions

Reuse of group data are very important for increasing the performance of the algorithm. It reduces the amount of time spent on solving single NHNP's, since already calculated values are reused. The runtimes have been measured for the standard test-set, where demand is medium, for networks of size up to 9. Finding the solution for the 10 node networks took more than 4 hours when reuse of group solutions was used. The solutions are not found without reuse of group solutions, since the expected runtime is 7 hours, calculated from the difference in runtimes for the networks with 9 nodes. Results are shown in figure 12.5.

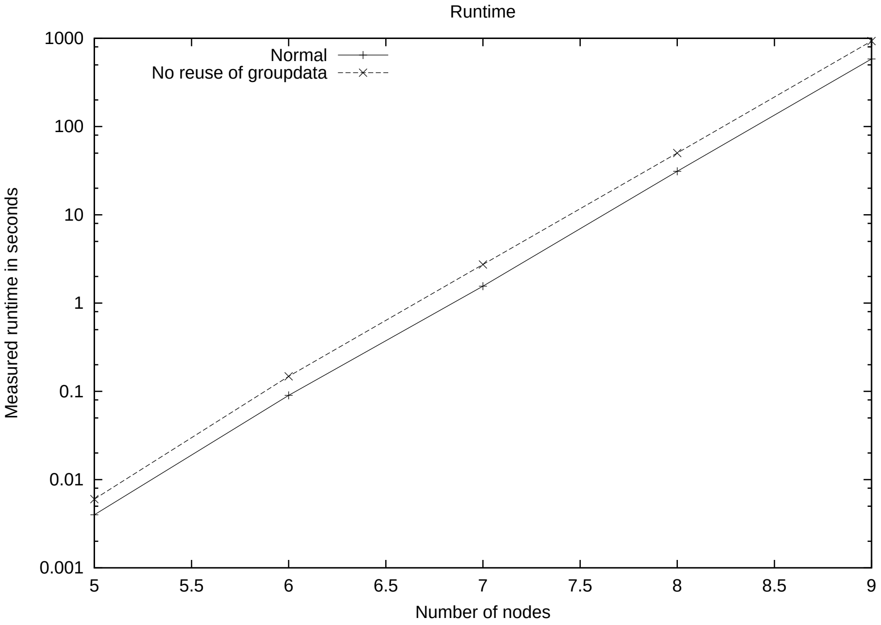


Figure 12.5: *Runtime with and without reuse of group data*

The graph shows that as expected, the algorithm performs substantially faster. The runtime is reduced with approximately 40%.

### 12.3.3 Limit on Group Size

When solving a HNP to optimality, the time spent on solving groups with many nodes is much higher than the time spent on solving groups with few nodes (see section 9.3). This is so though few group-divisions exist where one group is large and also few possible large groups exist compared with the number of medium sized groups. Nevertheless much time is spent on the large groups, and also this statement is true regardless of whether reuse of group data are used or not.

In order to be able to find solution values for larger networks and in order to speed up processing, limiting the maximum size of groups is considered. In some cases it may even be a natural part of the specification of a problem instance, since it may be the case that too large groups are not interesting. Also it is often not the case that the optimal solution is among the solutions were large groups exist, but it cannot be known for certain.

We run tests using the medium demand test set and find solutions using max group size on 5, 6 and 7 nodes. 5 instances of each network is solved. The result is shown in figure 12.6 where runtime is depicted as a function of the number of nodes in the network, for each of the three limit groups and the normal way, i.e. no limit is used.

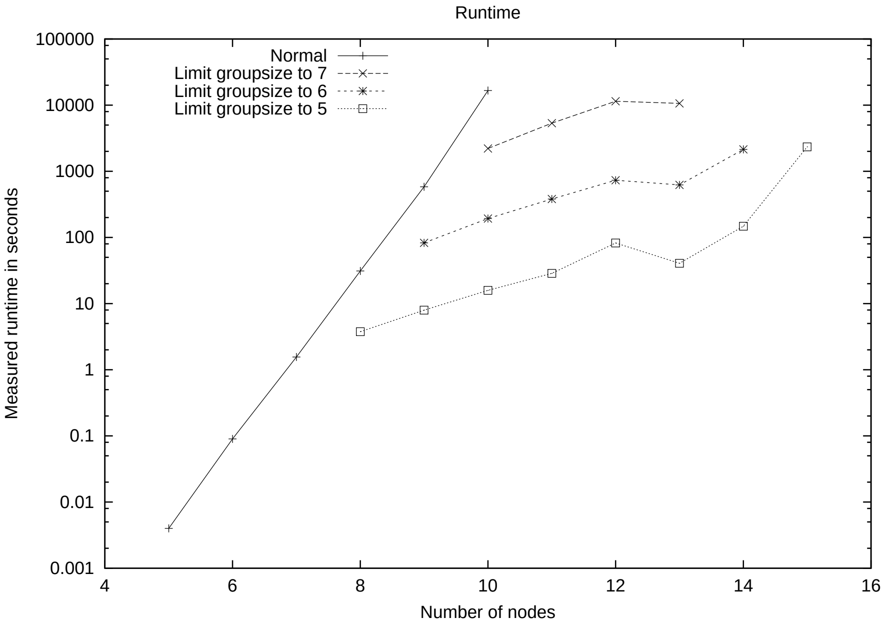


Figure 12.6: *Runtime as function of number of nodes in network*

All networks are divided into 3 groups except for the 15 node networks which are divided into 4 groups. Hence limiting the group size to 5 nodes does not make a difference for networks with 7 nodes or less, since no valid group division can be made containing groups of size larger than 5. The same foes for the other limit groups, thus tests are run only for networks of size larger than or equal to the selected limit plus 3.

The solution value is investigated to check that the optimal solution is found. For networks of size up to 10 nodes, the optimal solution is found in all cases. For 11 nodes an upwards, the optimal solution is unknown, hence instead we compare the solutions found using different limits on the group size.

For all the networks with 11 nodes, the solutions are in all cases the same regardless of the limit used. For two networks with 12 nodes, using limit 7 instead of 6 improve the solution found by less than 1%. The same solutions for the 3 remaining 12 node networks are found using either limit 6 or 7. The difference between using limit 5 and 6 is a bit higher - one solution for the 12 node networks is the same using either limit 5 or 6, and for the remaining four 12 node networks, the solution value found using limit 5 is less than 5% higher than the solution found using limit 6.

No solutions are found for the generated networks of size 13. The runtime drops (at least for limit 5) increasing network size from 12 to 13 nodes. The explanation seems to be that using such a low limit, also limit the minimum size of groups indirectly. For a network with 13 nodes, no groups of size less than 3 can be used, since all nodes are to be put in a group, hence having a group of size e.g. 2 will result in that one of the remaining groups have more than 5 nodes. Hence fewer group divisions are possible and the runtime drops.

For the size 14 networks, 2 feasible (out of 5) solutions are found using limit 5 and 6. The solutions found using limit 5 is approximately 10% worse than the solutions found using limit 6. The reason is that for network with 14 nodes divided into 3 groups with a maximum group size of 5, the only possible group sizes are 4 and 5. Hence a lot of solution possibilities are cut away.

The 15 node network is divided into 4 groups, hence the found solution may be better than for the 14 node network, since there are still room for small groups. The solution cannot be found using higher limits, though, so there is no solutions to compare with.

### 12.3.4 Bound Using Best HNP

The best known solution to the HNP problem can with solution values for some of the calculated groups be used as at least an initial bound (see section 9.5). The bound is only used when calculating the solution value for the primary group, and as discussed, additional use of this bound could be added when calculating the solution value for the secondary groups as well.

To see if it is worth trying this, we will test what effect using the bound has on the runtime at the current form. Figure 12.7 shows the runtime with and without using this way of bounding.

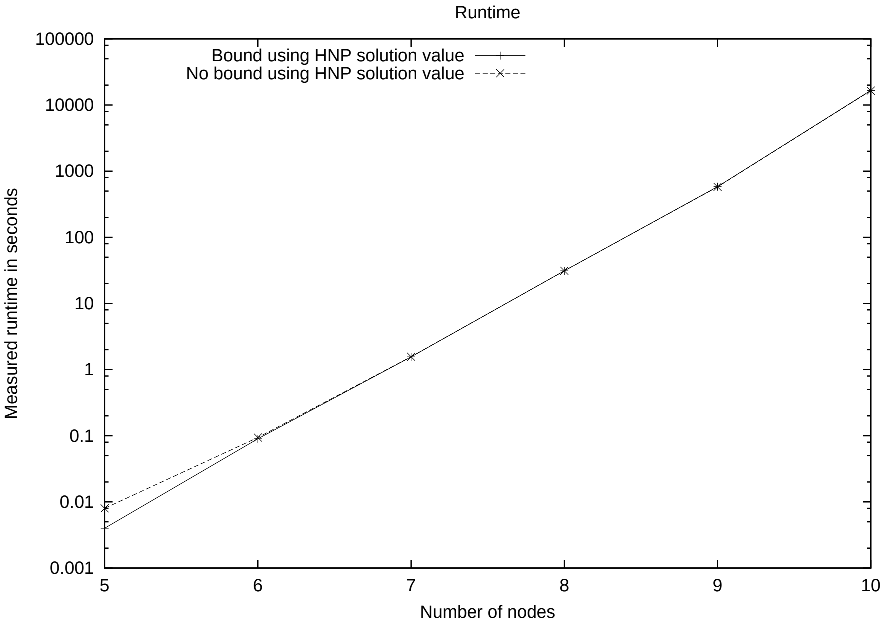


Figure 12.7: *Runtime with and without using best HNP as bound*

The figure shows, that only marginal improvements are achieved, but additional improvement can be expected if used on the secondary groups.

### 12.3.5 Dependency on the Total Amount of Demand

In this section, the runtime is considered in relation to how much the total demand is in networks. The hypothesis is, that with high demand, the bounds do not function well, since it is hard to even find a solution, and even harder to find a good one, thus it does not matter much that a good bound value can be calculated. In particular if a solution does not even exist the bounds calculated are useless.

In the branch and bound process, the amount of demand on each edge is calculated iteratively and it is checked that this amount does not exceed the capacity (see section 5.4). In networks with heavy demand, this enables us to fathom more solutions than in networks with light demand, hence the networks with heavy demand benefits more from this than the networks with light demand.

The runtimes for the different demand groups are shown in figure 12.8.

This shows that for light demand, the solution is found faster than for medium and heavy demand. For the heavy demand group, no solution at all is found, since no tree-solution exist. Here a small improvement is attained compared with the medium demand group. As described above, this is

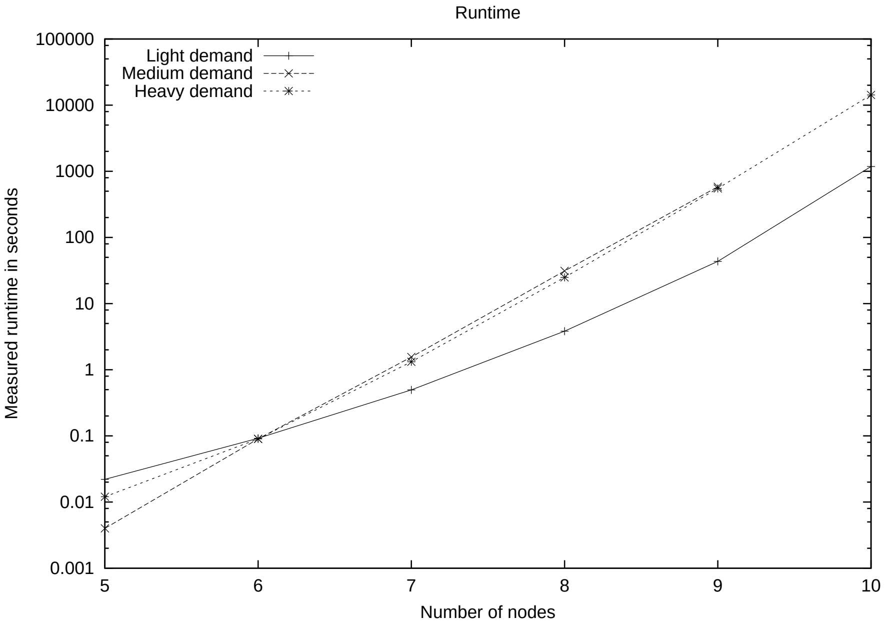


Figure 12.8: *Runtime for instances with different total demand amount*

probably because of the iteratively calculated demand of edges and comparison of capacities which detects fairly quickly that an edge exceeds capacity if demand is heavy in the network.

### 12.3.6 Setup-Cost/Flow-Cost ratio effect on the Flow Bound

The flow-cost bound did not work well, in fact the runtime increased using it (see section 12.3.1). The generated test instances have flow-cost lower than or equal to the setup-cost (see section 12.1). The contribution of cost in the solution value from setting up edges is usually much higher than the contribution from flow though.

In this section we will measure the effect of the flow bound for different setup-cost/flow-cost ratios. The setup-cost/flow-cost ratio is defined as the setup-cost for an edge divided by the flow-cost for an edge times the capacity. Hence this is two different measures but they are of course related.

In the test data used so far this ratio is the same for both the primary and the secondary edges (namely 1). The same ratio for the primary and secondary edges is used in this test as well. The ratio is controlled by modifying the setup-cost, and thus if primary setup-cost is changed by a factor, the secondary setup-cost is changed with the same factor, such that the setup-cost/flow-cost ratio is the same for both primary and secondary edges.

The runtime is measured for 5 instances of a HNP with 9 nodes, demand is medium and the runtimes reported are averages. The result is shown in figure 12.9.

The figure shows the runtime as a function of the flow-cost/setup-cost ratio, and two versions of the algorithm are used, one which use the flow bound and another one, which does not. It does not seem that the ratio has any influence on whether the flow bound should be used.

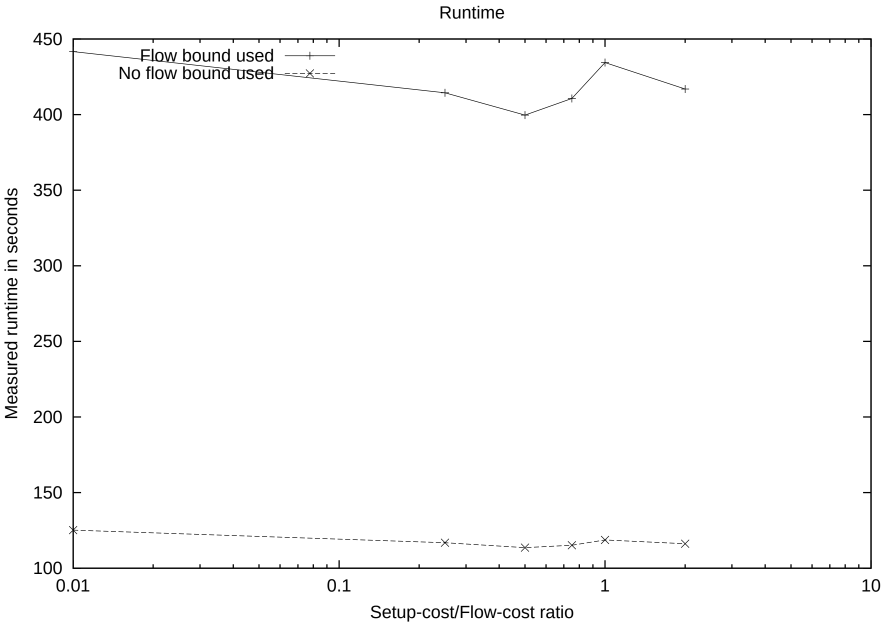


Figure 12.9: *Flow bound dependency on the setup-cost/flow-cost ratio*

### 12.3.7 Capacity Influence on Flow Bound

The flow bound performs differently depending on the capacity. The capacity has been varied for a network with 8 nodes, the result is shown in figure 12.10.

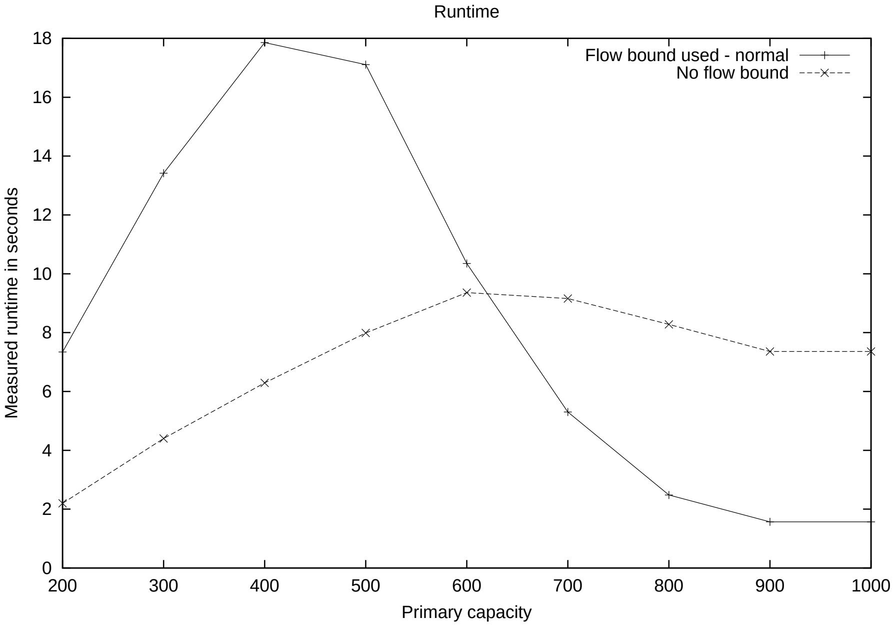


Figure 12.10: *Flow bound dependency on the capacity*

The capacity of the secondary edges is one fourth of the primary capacity. The figure shows, that if the capacity is above 600 and fixing the demand and costs at their current values, the flow bound should be used, whereas if it lower than 600, the flow bound should not be used to obtain the best performance. Since the standard test sets use a capacity of 400, the flow bound should not have been used, since a better performance would have been obtained without it.

## 12.4 Performance Tests of the Heuristic Algorithm for NHNP

In this section the NHNP algorithm is tested separated from the HNP algorithm.

### 12.4.1 Local Search

The effect of running local search with the two neighbourhoods are measured and compared with not running local search at all. The simple version of local search has the neighbourhood consisting of solutions where an edge is either added or removed, and the neighbourhood of the extended version allows in addition for swapping edges, i.e. an edge is added and one is removed (see section 6.5).

The tests are run on the medium demand group, and results are depicted in figure 12.11 showing deviation from the best solution, and figure 12.12 shows the runtime.

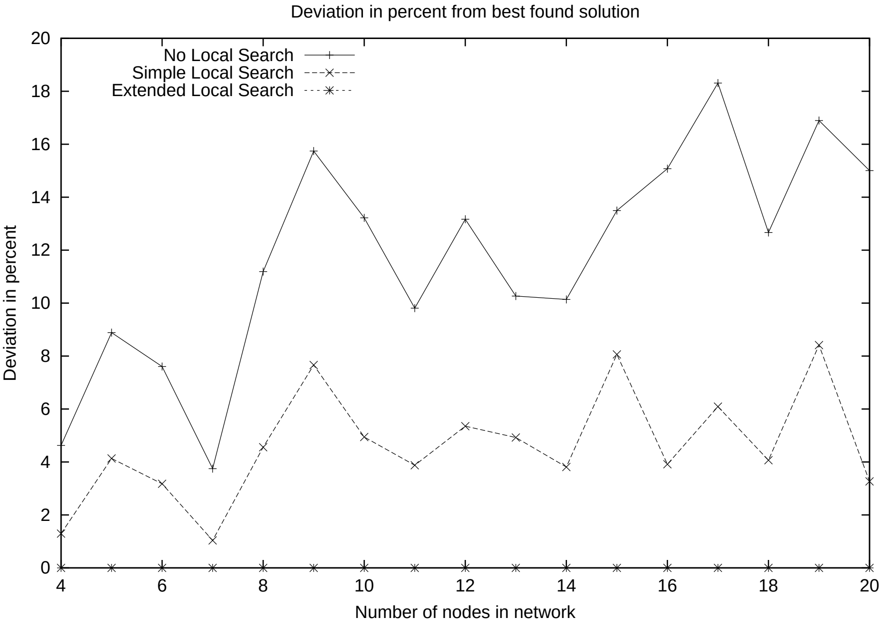


Figure 12.11: *Deviation from the best solution with and without local search with varying neighbourhoods.*

As expected the extended algorithm obtains the best solutions, whereas its runtime increases substantially with the number of nodes. When used as a subroutine in solving HNP's, the runtime for both the fast and the extended version exceed what we consider reasonable for networks with more than 15 nodes. Also solving networks with more than 10 nodes takes too much time using the extended version.

Thus when used as a subroutine, the neighbourhood used depend on the number of nodes in the network. Up to 10 nodes, the extended neighbourhood is used, between 11 and 15 nodes, the simple neighbourhood is used, and if networks with more than 15 nodes are to be solved, no local search is used at all.

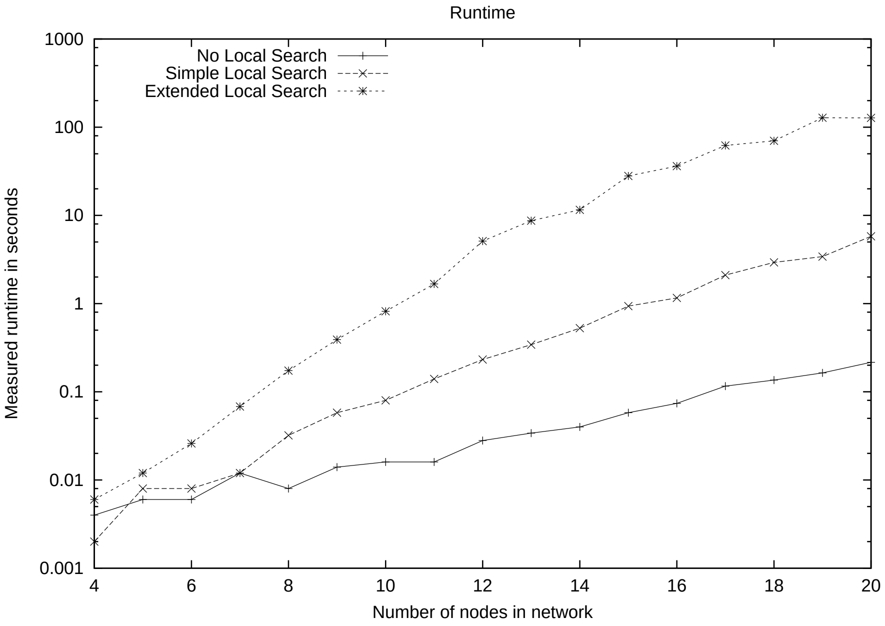


Figure 12.12: *Runtime with and without local search with varying neighbourhoods*

If the extended neighbourhood is used regardless of group sizes, the HNP algorithm will in some cases not finish in reasonable time, since much time is spent solving a single group. Runs have been seen where less than 500 iterations of the simulated annealing were run, but the runtime exceeded 10 hours.

The exact choice of where to use which neighbourhood can be used to control the amount of time spent on a NHNP, and then indirectly the total runtime of the HNP algorithm. But as mentioned in section 6.5, the idea of switching neighbourhood at a fixed number of nodes is rather inflexible.

An alternative algorithm which allow more control is a simulated annealing algorithm. It would be even better to use an algorithm, which could find a lower bound on a NHNP fast. This algorithm could be used by the HNP algorithm as described in the following.

Assume the HNP algorithm at some point has a current best solution and has selected a neighbour solution. The neighbour solution modifies either one or two secondary groups and the primary group compared with the current best solution. In order to find out whether the neighbour solution is better than the current best solution, it may not be necessary to calculate the value of the neighbour solution, a lower bound may suffice, if it is higher than the value of the current best solution. In this case it is not better than the current solution, and it may be discarded (possibly accepted anyway because of inherent randomness in the HNP algorithm).

The bound value of the neighbour solution can be calculated by summing the value of the groups which are unmodified from the current best solution, and calculating lower bounds on the remaining groups.

### 12.4.2 Path Assignment to Demand

In section 6.4 we argued that the paths should be assigned to demands in order of demand, which is denoted the normal algorithm. Possibly paths should be assigned to demands $ij$ where edge $ij$ was included in the solution before assigning paths to remaining demands in order of demand, which is denoted the simple algorithm. This was done to make sure a relieve edge exists (see section 6.4).

In most cases such an edge exists anyway but this is not good enough, we require that if a solution exists, then a solution has to be found and hence a relieve edge should exist. Thus two options exists. Use the simple algorithm or try assigning paths to demands in order of demand (i.e. the normal algorithm) and if at some point no relieve edge exists, restart using the simple algorithm (see section 6.4).

Both algorithms have been run on the standard test set of NHNP's. The solution values and the runtimes are compared, since both may be influenced. The test is run with medium demand and 5 instances are generated for each network size. The results are shown in figure 12.13 and figure 12.14.

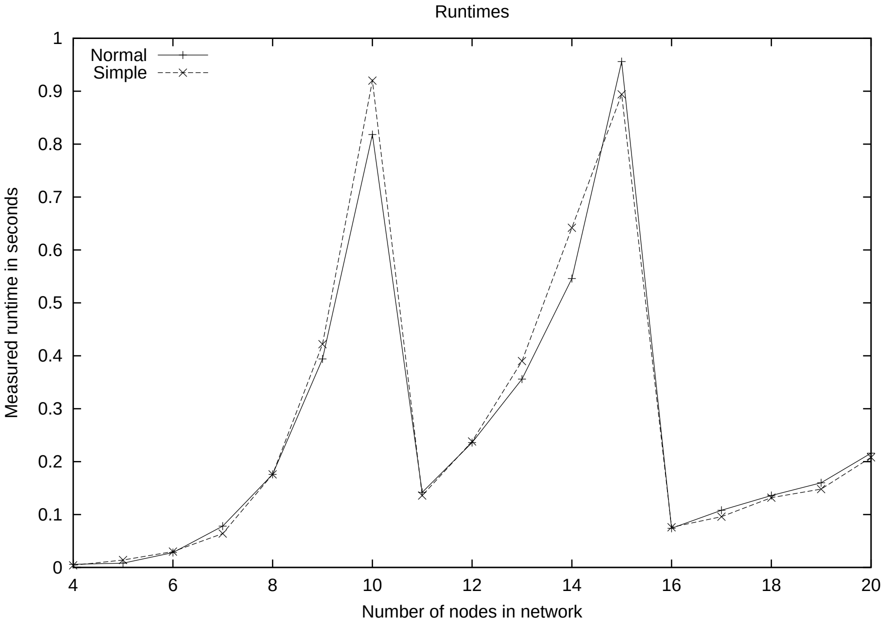


Figure 12.13: *Runtime for the two versions of the path assignment*

We have tested using the two different neighbourhoods in the local search depending on the number of nodes in the network as described in section 12.4.1. This is the reason the runtime increase with the number of nodes until from 10 to 11 nodes where the runtime drops, increase until 15 nodes and drops again and finally increase from 16 nodes and upwards.

In general it does not seem to be the case that one of the algorithms are faster than the other, though for networks with less than 11 nodes, the normal version of the algorithm seems to be fastest, and for networks with more than 15 nodes the simple version is fastest. A conservative conclusion is that the runtime of the algorithm depends entirely on the neighbourhood chosen.

The deviation is shown for single instances, and for each number of nodes 5 points exist for each algorithm (i.e. normal and simple). Many of the points has 0 deviation and hence falls at the same point. In most cases the normal version of the path assignment algorithm performs better than the simple version, except for the single network with 17 nodes, where the deviation is 19% from the solution found with the simple version of the algorithm.

It also seems, that when using some sort of local search (for up to 15 nodes), the normal version is the best, hence a conclusion is that which algorithm to use (the normal or the simple one) depends on whether local search is used. The choice may also depend directly on the size of the networks considered, but since the choice of algorithm does not seem to matter much, this idea is not pursued.

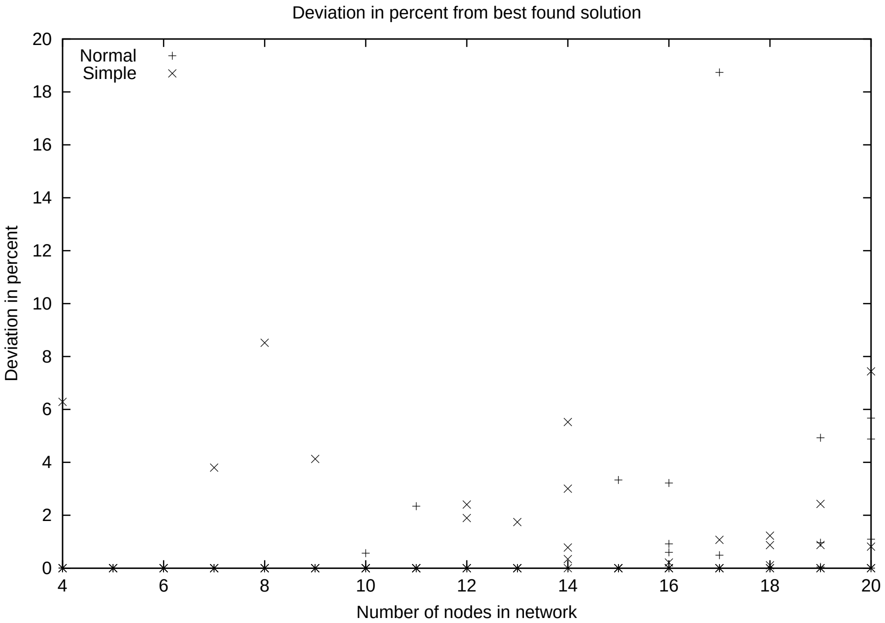


Figure 12.14: *Deviation for the two versions of the path assignment algorithm*

## 12.5 Performance Tests of the Heuristic Algorithm for HNP

In this section we test the simulated annealing algorithm which solves the HNP. The performance of the simulated annealing depends critically on the choice of parameters for the simulated annealing. Usually they are modified for one particular problem instance, but this is not possible in practice for all the generated test instances, since there are too many. Instead we will use the parameters specified in section 10.8 and 10.9 for all tests.

### 12.5.1 The Value of the Initial Solution Compared with the Final Solution

One possibility to solve a HNP is to run a greedy heuristic as the ones which are used to find the initial solution. The heuristics used for finding the initial solution are not optimized to find the best valued solution, but rather to find a feasible solution which is a good foundation for the simulated annealing algorithm. Nevertheless comparing the initial solution with the final solution can give an idea on how well the simulated annealing algorithm performs.

The initial solution is found in three different ways using the three algorithms random, simple and assignment (see section 10.5). The tests are run on the medium demand test set, 1 instance each. Using the three algorithms we find an initial solution and record the value of the initial solution. For all three runs we also record the final solution, and select the best of the tree. For theses four groups, the solution values are depicted in figure 12.15.

As expected the solution values are best for the assignment algorithm and worst for the random algorithm. For networks with more than 15 nodes, the minimum improvement from using the assignment algorithm to the final solution is 7%, and the best improvement is for the 100 node network which is 18%.

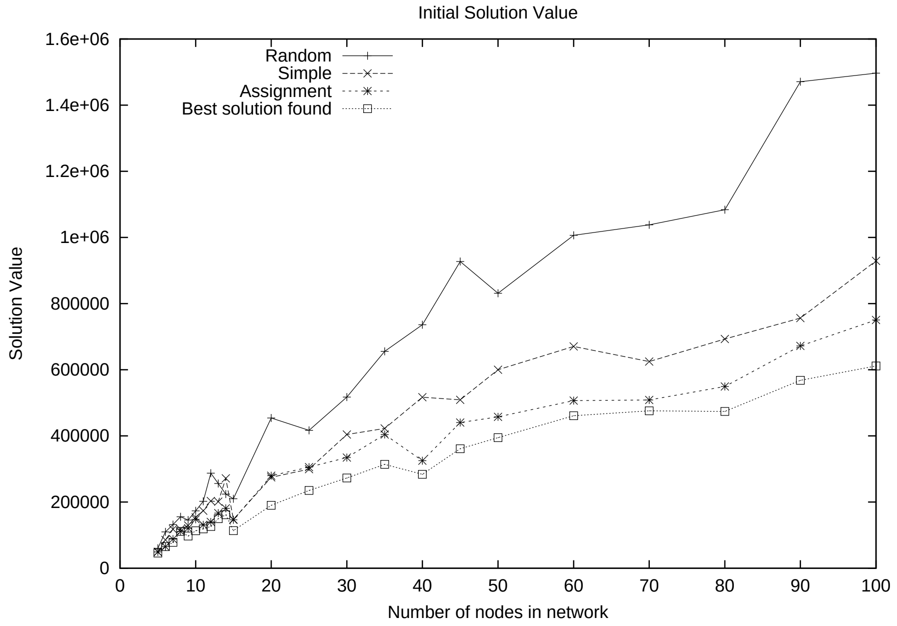


Figure 12.15: *Solution value as function of number of nodes for the three initial solution algorithms compared with the best solution*

### 12.5.2 Finding Initial Solution for Heuristic Solution of NHNP

Finding an initial solution has an effect on how good a solution we end up with. But it is not necessarily the case that the least cost solution is the best to start out with. In this section we test what effect the three algorithms for finding the initial solution has on the final solution. The three schemes are a random initial solution a simple low valued solution and a low valued solution found using the assignment algorithm (section 10.5).

The three algorithms are run on the standard test-set, one instance and medium demand. The runtime of the three algorithms is in the order of seconds, hence since the runtime of the simulated annealing algorithm is much higher, the runtime of the algorithms for finding the initial solution does not matter much.

The solution value is depicted in figure 12.16 showing the deviation from the best solution as usual.

There is no general conclusion from this - the random way of finding the initial solution may be better than the found low valued solution. The simple way of finding the initial solution seem marginally better than the others, but the difference is small, and since only one instance is run for each depicted point it may not be the case in general.

### 12.5.3 Effect of Reusing Calculated Group Solutions

Reusing calculated solution values for groups by saving them in a hash table gives an enormous speed up as we shall see shortly. Also the speedup is higher for networks with many groups and hence many nodes, at least for the test-set. The saved calculated solution values for groups are usually reused in neighbour-solutions, and in fact modifying a solution change either one or two secondary groups plus the primary group. Hence for the largest networks which are divided into 9 groups, a reduction of the runtime can be expected to be on at least 70% for the largest networks.

The runtime with and without reuse of group values are measured by running tests on the standard

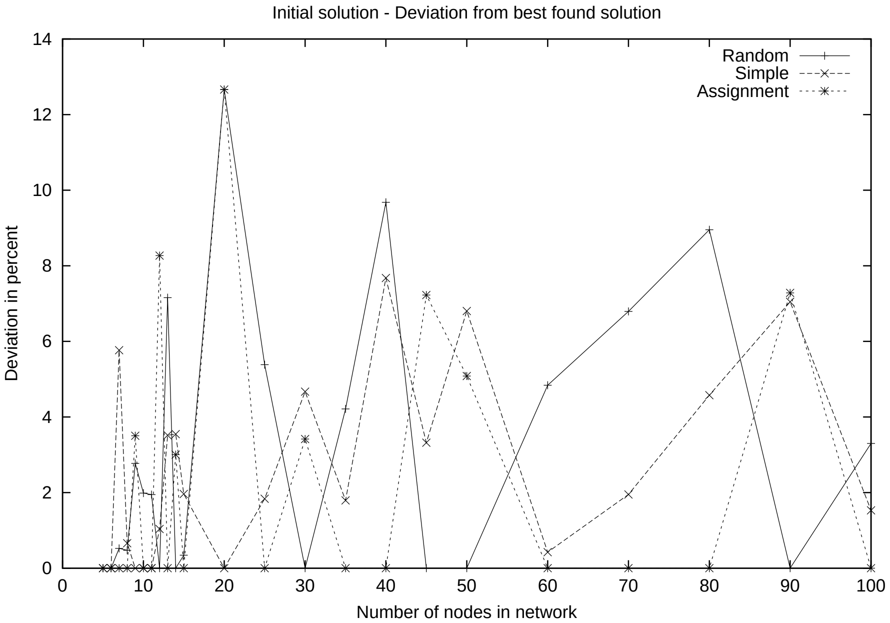


Figure 12.16: *Solution value deviation for the three versions of the initial solution finder algorithm*

test-set, one instance for each and medium demand. The solution value is the same, since no other modifications are done to the algorithm. The runtime is shown in figure 12.17.

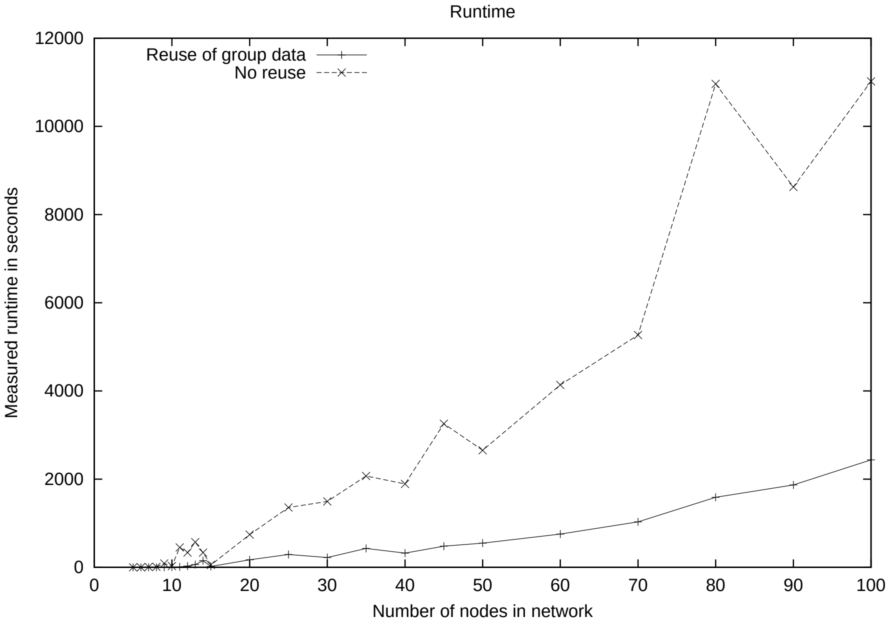


Figure 12.17: *Runtime with and without reuse of solution values for secondary groups*

As expected the runtime decrease is high, more than a 70% runtime improvement is achieved in the large networks. As mentioned the improvement is better the more groups the network is divided into, which is a good thing, since these are the networks which take substantial time to solve.

### 12.5.4 Limit Neighbourhood

In section 10.3 we described the neighbourhood. The neighbourhood was limited by, for a given group, considering only the $|V|/G$ closest nodes which are not in the group. In this section we investigate how removing this limitation affects runtime and solution quality.

The performance tests have been run on the data-set with medium demand and one instance only for each. Two versions have been run, one where the neighbourhood is limited and one where it is not. The runtimes are shown in figure 12.18 and the deviations are shown in figure 12.19.

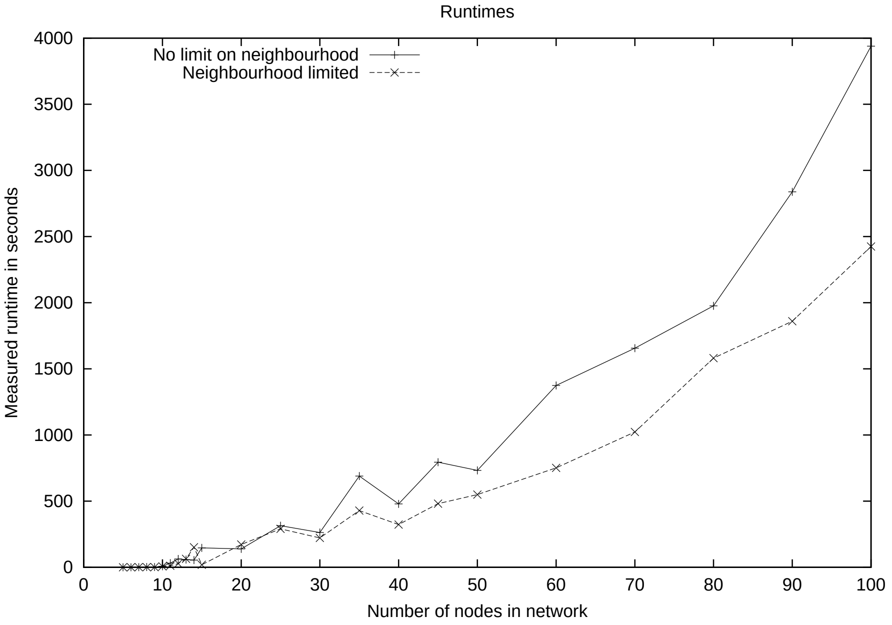


Figure 12.18: *Runtime for finding HNP with and without limitation on neighbourhood*

The runtime is higher when there is no limit on the neighbourhood - up to 50% higher for the large networks. Some of the increased runtime stems from an increased number of iterations, but this cannot explain all of the increase in runtime. When no limit on the neighbourhood is used, the number of group-neighbours considered is higher than the number of concentrator neighbours (see section 10.3). For the concentrator-neighbours, more groups can be reused than for the group-neighbours, hence since we consider more group-neighbours the runtime is higher.

In general the limited neighbourhood finds better solutions than the case where the neighbourhood is not limited. The zig-zag structure of the graph stems from the number of groups which are in the network. In the data-set the number of groups is 4 for 20 and 25 nodes and 5 for 30 and 35 nodes and so forth up to 9 groups in the network with 100 nodes. Variations on this is investigated in section 12.5.7. From the current data an immediate conclusion is, that the limited neighbourhood finds better solutions than the unlimited neighbourhood for higher values of the relationship between the number of nodes in the network and the number of groups.

Though using the unlimited neighbourhood performs better than the limited one in some cases, depending on the number of groups, it seems safe to conclude that the limited one should be used, since it is faster and the quality of the solutions on average are much better than the corresponding solutions found with the unlimited neighbourhood.

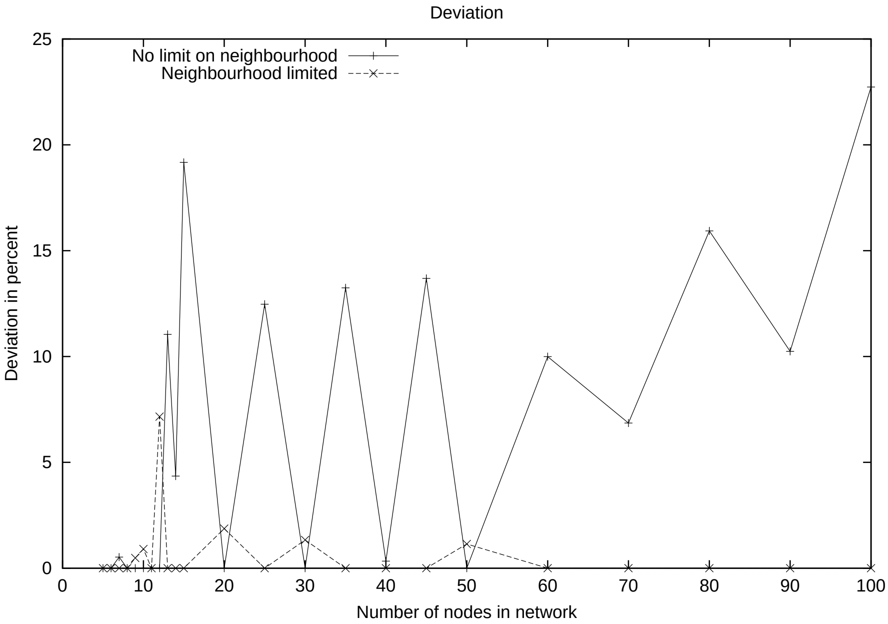


Figure 12.19: *Deviation in solution value for finding HNP with and without limitation on neighbourhood*

### 12.5.5 Cycled and Random Neighbourhood

In this section we compare cycling the neighbourhood with selecting neighbour-solutions at random (see section 10.3.2). The solution value has been measured, and the deviation is calculated as usual. The test has been run using the medium demand test set, and three instances for each. The result is shown in figure 12.20.

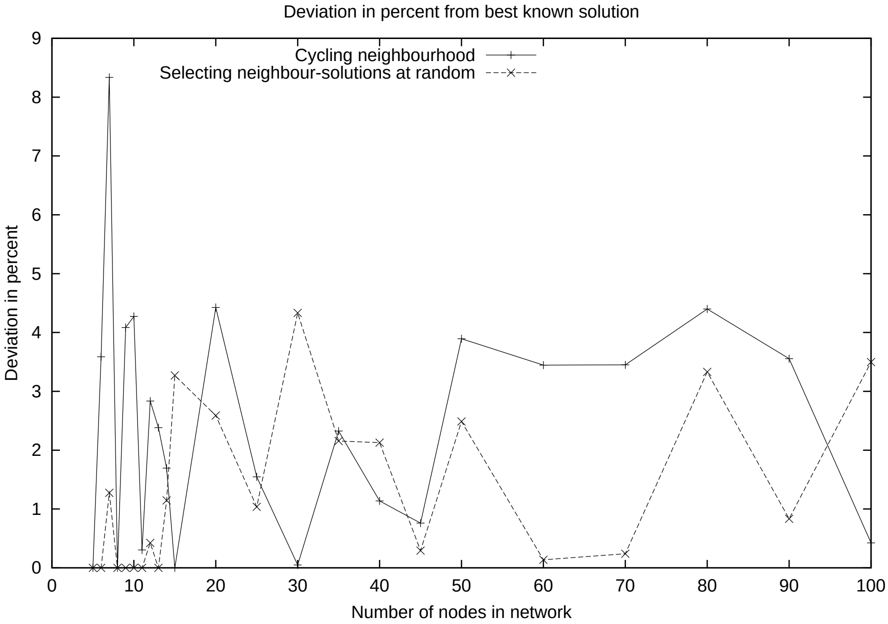


Figure 12.20: *Deviation between cycling neighbourhood and selecting neighbour-solutions at random*

In all cases except for the networks with 7 nodes, the average deviation is less than 5%. For the large networks with between 50 an 90 nodes selecting neighbour-solutions at random performs the best. But it can also be seen (since none has deviation zero) that at least in one of the three tested instances, cycling the neighbourhood found the best solution. I.e. for e.g. the 3 instances with 50 nodes, neither cycling the neighbourhood nor selecting neighbour-solutions at random has an average deviation equal to zero. Hence each of the two must have found the best solution at least once each.

Hence it is hard to say which is best, but in these particular test instances the random neighbourhood performed marginally better.

### 12.5.6 Measure Function Value at each Iteration

The simulated annealing algorithm should initially start out by searching the search space at random - it should accept solutions with higher objective function values than the current. In the end of the run, no solutions with higher solution values should be accepted.

To check that this corresponds to the way the algorithm works, the solution value at each iteration is recorded, and the best solution is recorded. The objective function value is plotted for each iteration for a network with 10 nodes and a network with 100 nodes. The demand is medium. The tests were run when finding the simulated annealing parameters, and in particular the figures 12.21 and 12.22 shows some old results, found with other simulated annealing parameters.

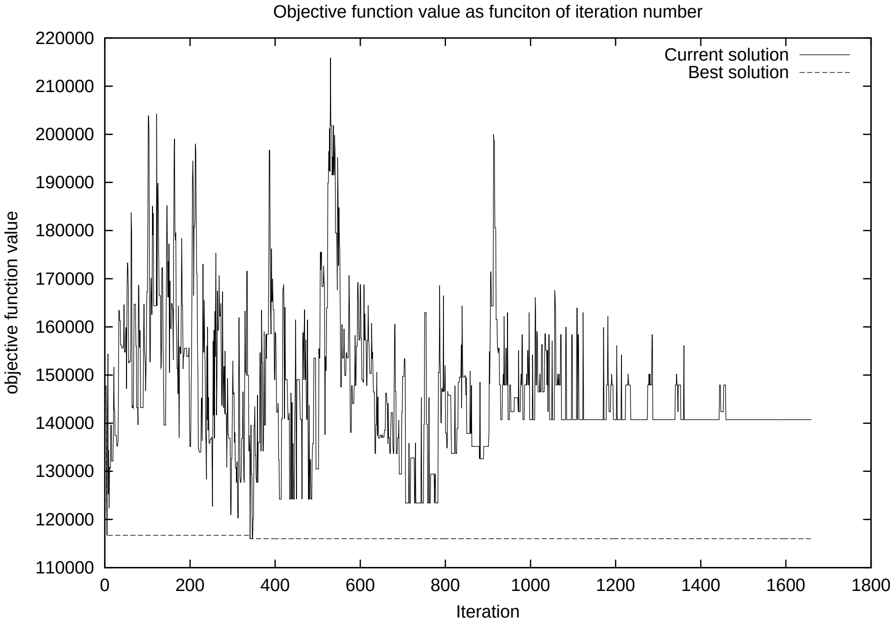


Figure 12.21: *Objective function value for network with 10 nodes - old version*

In both cases, as expected the objective function value varies most in the beginning and vary less as the temperature decrease. Hence in this sense it seems that the algorithm performs as desired. But for the network with 10 nodes, it seems that the temperature is too high initially, since a number of worse solutions are accepted initially, such that the algorithm escaped the seemingly good local minimum. This was generally the case for small networks, hence the initial temperature was lowered for small networks.

For the network with 100 nodes, it seems that a low temperature was reached too early, since after 2500 iterations (out of 4000), the algorithm did not accept many higher cost solutions. Since this

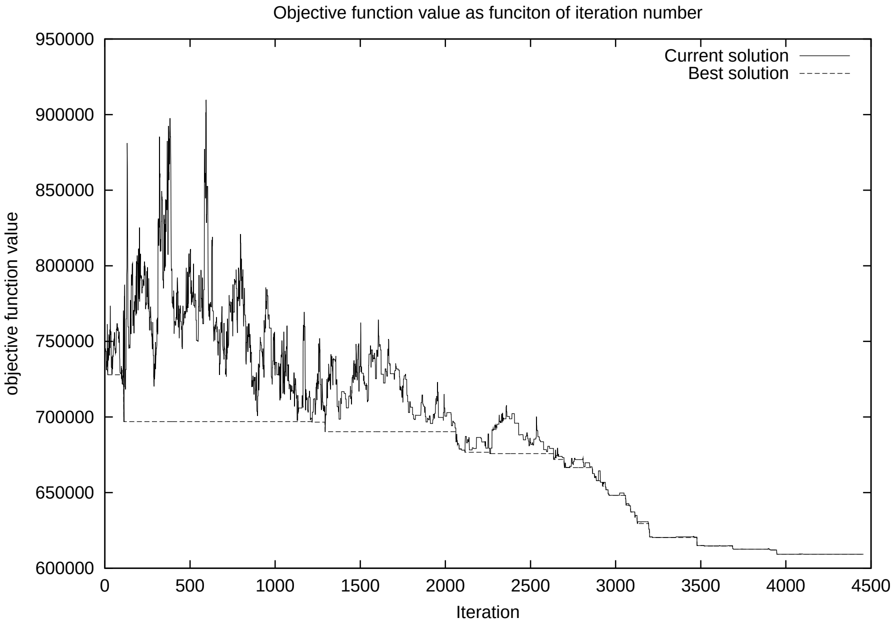


Figure 12.22: *Objective function value for network with 100 nodes - old version*

was the case for most large networks, the update factor was increased, such that the temperature did not decrease as fast as before. This was done for large networks only.

The networks solved with the modified algorithm are shown in figure 12.23 and 12.24.

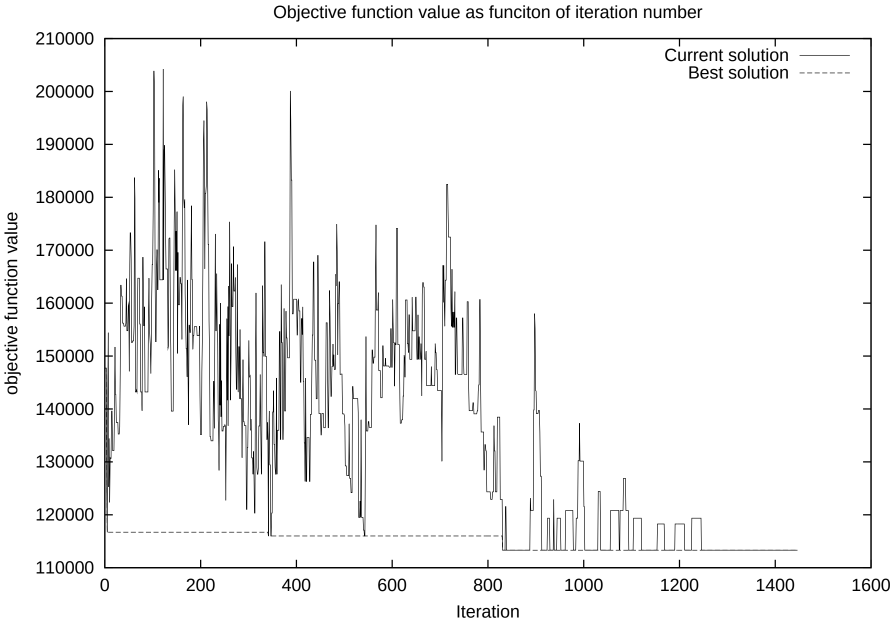


Figure 12.23: *Objective function value for network with 10 nodes - old version*

When solving the network with 10 nodes, the method now returns to the best solution and improve this. When solving the 100 nodes network, the same problem to some extent exists though now fewer iterations is used to attain roughly the same result. Hence the update factor could in this case probably be increased even more, at least it could be tried.

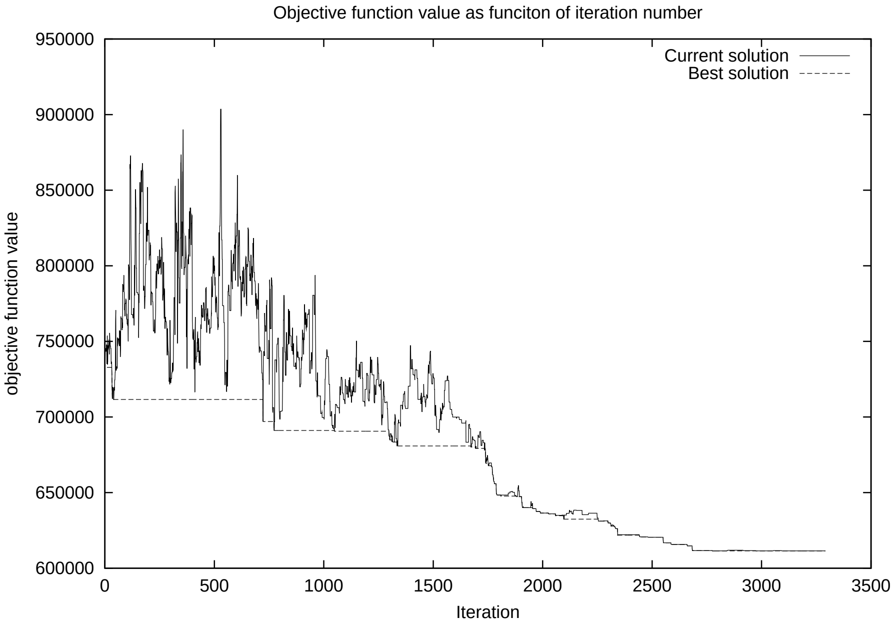


Figure 12.24: *Objective function value for network with 100 nodes - old version*

The parameters for the simulated annealing algorithm were tested by modifying single parameters and recording the final solution value, but no pattern was seen in the results, hence in general it is hard to find the best parameters. Thus if a single problem instance is solved, and finding a good solution is important, the solution quality can probably be improved by trying different parameters.

### 12.5.7 Number of Groups

In the entire definition of hierarchical networks and the implementation of the algorithms, we have assumed that the number of groups was selected and specified by the user. Another possibility is that the number of groups is found by the algorithm. It is unknown whether the limitation on the number of groups to consider is a severe limitation, i.e. how much the solution value differs for solutions with different number of groups.

Given a problem instance, the solution value can be found for a division into a different number of groups by simply running the algorithm several times, specifying different number of groups. This is done for 3 problem instances generated from the medium demand set with 10, 50 and 100 nodes. The runtime for the different instances is shown as a function of the number of groups in figure 12.25. The objective function value is shown as a function of the number of groups in the networks in figure 12.26.

The runtimes for the networks with 10 nodes does not immediately seem to follow a pattern. One explanation could be that large groups are the most time consuming to solve. For the network divided into 2 groups all divisions have a secondary group of size 5 or more, whereas when divided into 4 groups, no size 5 groups arise. When dividing into 5 groups, the primary group is a 5 node group, hence the time grows as well. Some of the results could also be due to pure coincidence - only one instance is solve for each point.

For the networks with 50 nodes, the runtime is lowest for the network divided into 7 groups (ignoring the division into 11 & 12 groups), which again stems from that if the network is divide

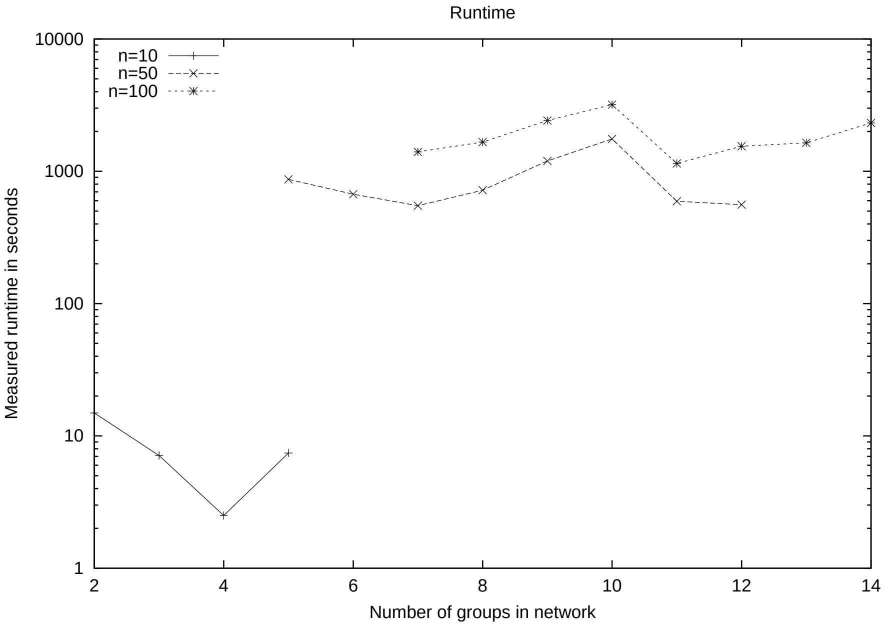


Figure 12.25: *Runtime as a function of number of groups in network*

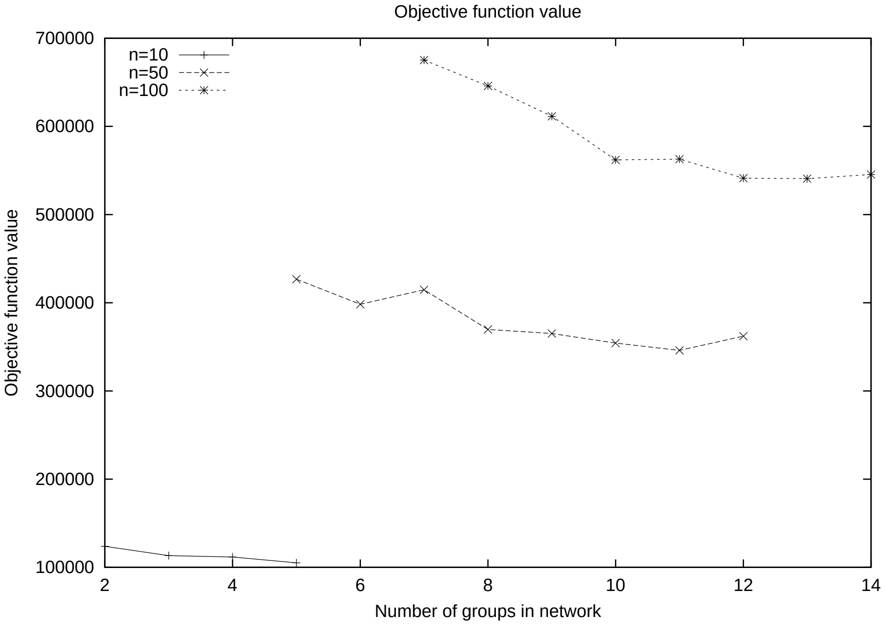


Figure 12.26: *Objective function value as a function of number of groups in network*

into fewer groups, secondary groups are in general large, and on the other hand if the network is divided into more groups, then the primary group is large, and the runtime increase.

The division of the solution into 11 & 12 groups have a lower runtime, since the neighbourhood used in the local search is reduced for groups of size larger than 10 nodes (see section 12.4.1) and the primary group is calculated for each iteration and thus have major influence on the runtime.

For the network with 100 nodes, the same explanations are valid as for the network with 50 nodes.

The objective value seems to drop as the number of groups increase. For the problem instances with 50 and 100 nodes, investigation of the solutions shows, that if the network is divided into few groups, the demand cannot be satisfied unless many edges are added, i.e. many more than simply connecting the nodes is required.

Since the setup-cost for primary edges is only double of the setup-cost of secondary edges and the capacity of primary edges is 4 times the capacity of secondary edges, we get "more capacity for the same money" if using the primary edges. This becomes apparent for the solutions to the problems divided into many groups. In this case more primary edges are selected, but the total number of edges is much lower than when the same network is divided into few groups.

For the networks with 100 nodes, when going from 10 to 11 groups, the objective function value does not drop correspondingly. The same explanation as for the runtime is valid - the neighbourhood is reduced for the local search of the primary group, hence the solution does not have such a good value as it probably could have had if the extended neighbourhood was used.

### 12.5.8 Dependency on Demand

If the demand is low in the network, then so is the flow-cost and the chance of exceeding capacities on edges. Finding the initial solution to a NHNP is done by finding a MST minimizing the setup-cost, hence the initial solution is suspected to be better and found faster for networks with light demand than for networks with heavy demand. This is what is verified in this section.

The tests are carried out using the standard test-set for the three demand groups light, medium and heavy. The runtimes are depicted in figure 12.27.

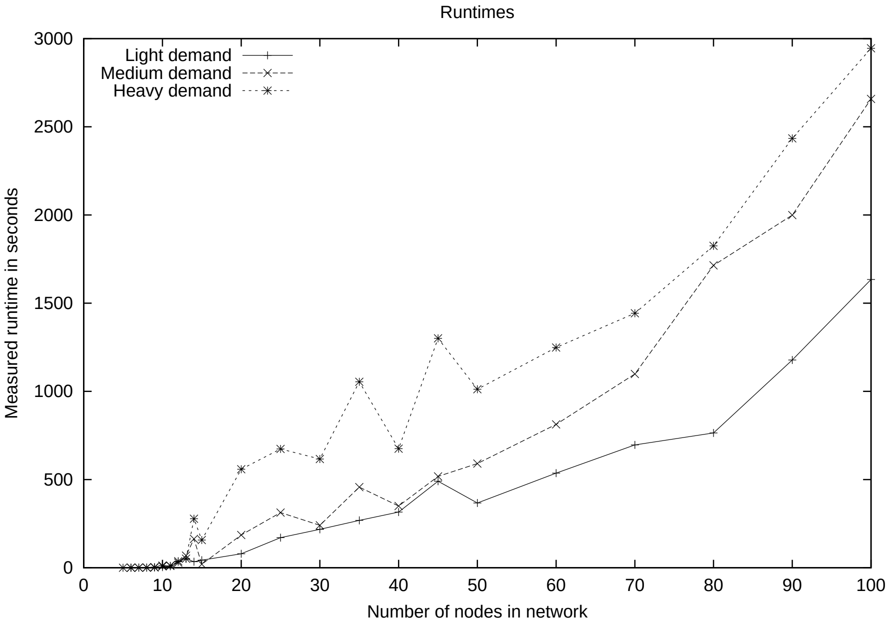


Figure 12.27: *Runtimes for finding solution to HNP for networks with light, medium and heavy demand*

The figures show, that the runtime is lower for the lightly loaded networks compared to the medium and high loaded networks. This is expected, since assigning paths to demands depend heavily on the amount of demand which are to be assigned paths. If many edges are to be added, this cannot be easily handled by the first phase of the path assigner, hence the second part will have to look at many neighbour solutions, regardless of whether the simple or the extended neighbourhood is used.

## 12.6 Quality of Heuristic Solutions

For HNP networks with up to 10 nodes, we can find the optimal tree-solution. For networks with up to 15 nodes we can find some solutions by limiting the maximum size of groups. Hence these solutions are used as reference for comparison with the heuristic solutions. The heuristic solutions are found using the standard set up of the heuristic algorithm.

The measured quality of the solutions are shown in figure 12.28 for light demand test set and figure 12.29 for medium demand test set.

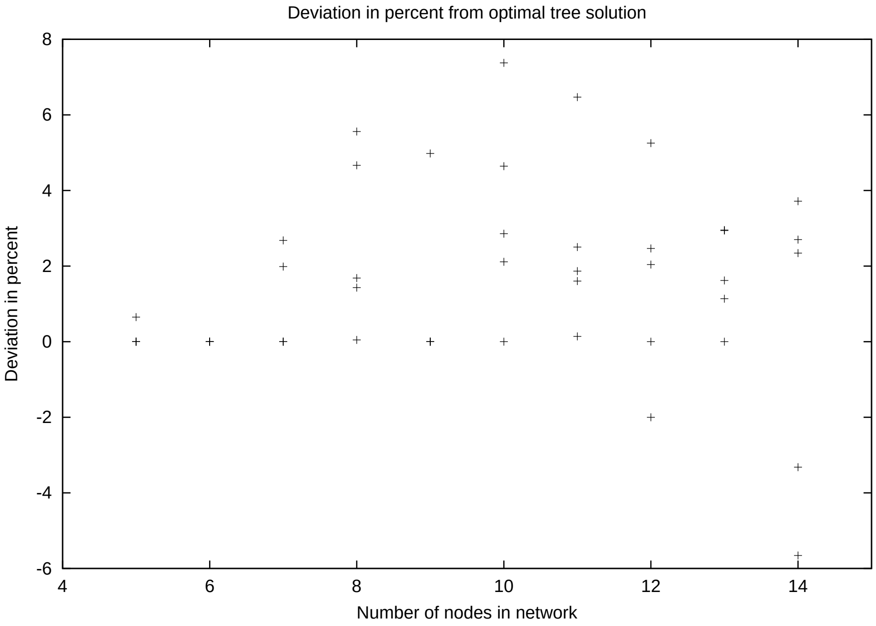


Figure 12.28: *Deviation from the optimal tree solution - light demand*

Note that the heuristic solution can be better than the“optimal” solution, since the optimal solution is restricted to tree solutions. This is indicated as negative deviation.

Five instances for each network size is generated. For the light demand networks tree solutions can be found for all networks, which is not the case for medium demand networks. For some networks tree solutions cannot be found either because they do not exist, or because group sizes are limited. These networks are not included in the figure. For the medium demand networks, a tree solution could not be found in 7 out of the 40 networks with 12 nodes or less, and only 4 solutions are found for the networks with 13 to 15 nodes.

As a passing remark this does not correspond with how we defined the demand sets since the medium demand networks were defined as networks with a demand such that a tree solution existed. The reason that this happens is partly a coincidence - the exact demand between node pairs are generated but also the total demand is probably estimated too high. The demand was estimated by considering example networks solved heuristically. If the heuristic solution had only few more extra edges than the number of nodes, it was assumed that a tree solution existed. Obviously this is not true in all cases.

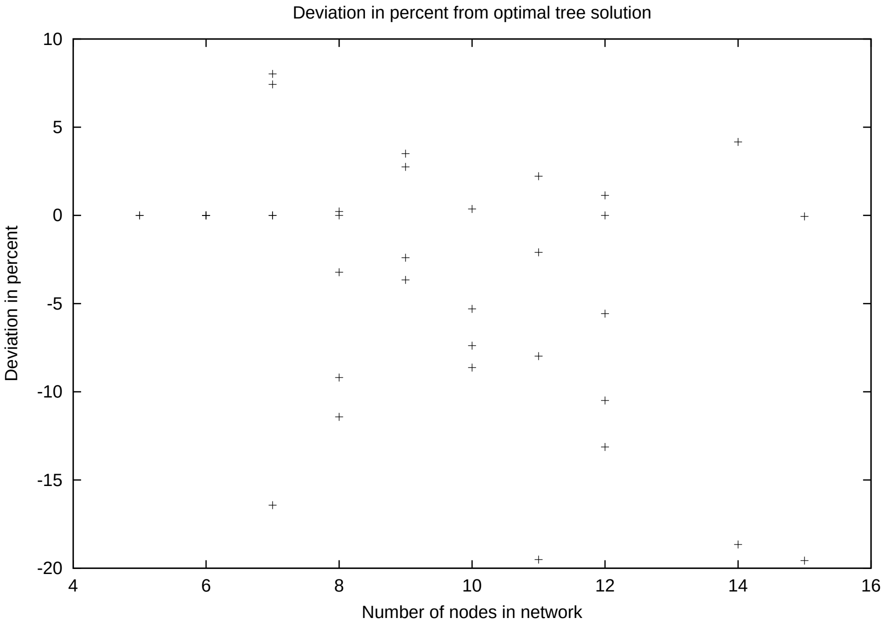


Figure 12.29: *Deviation from the optimal tree solution - medium demand*

Some heuristic solutions have the same solution value as the optimal tree solution, hence it is not possible to distinguish between points representing one and points representing more instances. Another way to get an idea of how well the heuristic solution algorithm performs is to state the number of times the heuristic solution algorithm find the same solution as the optimal solution algorithm. For the light demand networks, this is the case for 19 out of 50 networks, most of them for the networks with few nodes. For the medium demand networks, the same solution is found in 10 out of 37 networks.

For the light demand networks, the heuristic solutions are in most cases worse than the optimal solutions. This agrees with that the demand is light, hence the setup-cost dominates the solution value, and since the minimum setup-cost attainable is a tree, a tree solution has a low total cost value. All heuristic solutions have costs within 8% of the optimal tree solution.

For the medium demand networks many heuristic networks have a lower value than the optimal tree solution. This is of course so since the demand is higher than for the low cost, hence situations arise in which it is cheaper to have a non-tree than a tree solution. Hence the deviation is not necessarily a good estimate on the quality of the solution given the demand. The maximum deviation is 8%, but usually much better.

# Chapter 13

# Conclusion

Motivated by hierarchies in telecommunication networks, hierarchical networks have been defined and described. The underlying networks chosen are capacitated networks with flow- and setup-costs, which are in general hard to optimize. Other underlying networks could have been chosen, e.g. uncapacitated networks or networks with only setup-costs, and hierarchies could be defined using these networks as well.

Solution algorithms are developed solving the hierarchical network problem to optimality for few nodes, and heuristically for up to 100 nodes well within time limits, i.e. faster than a couple of hours. The solution algorithms are based on a generally applicable division into mainly two phases, one which deal with the hierarchies, and one which deals with the network. Given this division, it should be possible to reuse the first part dealing with hierarchies for other types of networks.

The optimal solution algorithm can solve hierarchical networks with nodes of up to 10 nodes and if group sizes are limited up to 15 nodes. The algorithm is based on a branch-and-bound scheme. Depending on the network data, using the bounds has a positive influence on the runtime, though the optimal solution algorithm shows to have two problems: The number of different groups grows exponentially with the number of nodes, and the runtime for solving each group grows exponentially with the number of nodes in the group.

Some group calculations can be reused, since changes in one group does not necessarily influence other parts of the hierarchical network. This reduce the effect of the first point above, but the primary group nevertheless has to be recalculated, hence the number of different group divisions affects the algorithm severely. Usually, a division of the network into groups where one or few groups contain many more nodes than other groups is not desirable. Thus, the second point is dealt with by accepting only groups of a maximum size. This way we avoid the optimization of the time consuming large groups.

The Heuristic applied is based on simulated annealing of phase 1, and a greedy algorithm in phase 2 eventually followed by a local search scheme. The simulated annealing approach requires identification of temperatures, update scheme and accept criteria. These depend highly on the number of nodes in the network solved, hence parameters are found as functions of the number of nodes. Reuse of group-data are also used, and in particular for large networks, the gain is high.

We test the improvements of the optimal algorithm and the heuristic algorithm, both for runtime and for the quality of the solutions found, where relevant. This shows that the algorithmic constructs have effect both with respect to runtime and in the heuristic case on the quality of solutions.

The performance of the algorithms are also measured when the input data are varied. For the optimal solution algorithm, the performance depend on the amount of demand and the capacity of the edges. In general the choice of which version of the algorithms to use depend on the data.

In both the optimal and the heuristic case it is very important that solved groups are saved and reused, since this gives a high speed up.

Measuring quality of heuristic solutions is in general hard. We compare heuristic solutions to the optimal solutions when these can be found, and compare solutions “internally” between different runs of the heuristic. Also we compare with solutions which are found using greedy algorithms - the solutions are used initially in the simulated annealing algorithm.

The structure of the solutions is in some cases investigated, and the solutions seem reasonable. I.e. if total demand is low, solutions are usually trees, and groups consist of nodes, which are close, whereas if the total demand is increased, the groups are usually separated, i.e. the demand control the group division and many edges may be selected in groups.

## 13.1 Outlook

Many questions are left for further investigation, some important ones are considered here. The heuristic algorithm should be applied to some real world data in order to compare solutions found with existing solutions. This would give a better indication on whether the found solutions are beneficial. Applying the algorithms to real world data would probably require extension of the heuristic solution algorithm to handle more than two levels, and also it would be beneficial to extend the algorithm such that it could find the best number of groups to divide the network into.

Given the results in this thesis there does not seem to be any doubt, that these extension could be carried out, though some important questions are left unanswered. The most important is probably how data can be reused between neighbour-solutions. In the implemented algorithm the key to the performance was the reuse of group data, but primary group data were never reused. If more hierarchies were present, reuse should be possible in more levels, and the performance of such a scheme is unknown.

Other open questions are whether better solution algorithms can be applied to the NHNP’s in order to improve solution quality and lower runtime. Since capacitated networks are well studied, it is likely that such algorithms exist and can be adapted in the solution process. The solution algorithms should probably be modified to fit our needs, which is that of solving many moderately sized networks fast as opposed to solving a large network.

Another possibility is to look into what other underlying models could be used with hierarchies. As mentioned the flow-cost is not used in telecommunication networks - costs can instead be modeled using the setup-cost. Hence what happens if the underlying model used is a model without flow-cost? Is this easier to solve or is the runtime the same?

An unpleasant limitation is that we do not allow flows to split. In telecommunication networks flows can split, at least in some fixed sizes, but how can this be handled?

It does not seem likely that an optimal solution strategy can be built for solving the hierarchical networks. This is no doubt so, since the problem of solving the underlying networks is NP-hard, but even if the underlying model were replaced by e.g. an uncapacitated network type, it seems unlikely that such algorithms can be found. But of course these can be interesting, if used as models for building better heuristics.

# Bibliography

[1] A. K. Aggarwal, M. Oblak, and R. R. Vemuganti. A heuristic solution procedure for multicommodity integer flows. *Computers and Operations Research*, 22(10):1075–1087, 1995.

[2] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 459 –468, 1993.

[3] Anataram Balakrishnan, Thomas L. Magnanti, and Prakash Mirchandani. Designing hierarchical survivable networks. *Operations Research vol. 46, Issue 1*, 1998.

[4] Dhritiman Banerjee and Biswanath Mukherjee. Wavelength-routed optical networks: Linear formulation, resource budgeting tradeoffs, and a reconfiguration study. *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications*, 1997.

[5] L. Brunetta, M. Conforti, and M. Fischetti. A polyhedral approach to an integer multicommodity flow problem. *Discrete Applied Mathematics*, 101(1-3):13–36, 2000.

[6] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization.* Wiley-Interscience, 1998.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1996.

[8] J. Current and H. Pirkul. The hierarchical network design problem with transshipment facilities. *European Journal of Operational Research*, 51(3):338–47, 1991.

[9] J.R. Current. The design of a hierarchical transportation network with transshipment facilities. *Transportation Science*, 22(4):270–7, 1988.

[10] J.R. Current, C.S. ReVelle, and J.L. Cohon. The hierarchical network design problem. *European Journal of Operational Research*, 27(1):57–66, 1986.

[11] Rudra Dutta and George N. Rouskas. A survey of virtual topology design algorithms for wavelength routed optical networks. 1999.

[12] Bernard Gendron, Teodor Gabriel Crainic, and Antonio Franginoi. Multicommodity capacitated network design. *http://www.di.unipi.it/ frangio/curvitae.html*, 1997.

[13] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research.* McGraw-Hill, 1995.

[14] K. Holmberg and D. Yuan. A lagrangian heuristic based branch-and-bound approach for the capacitated network design problem. *Operations Research*, 48(3):461–81, 2000.

[15] A. Kamath, O. Palmon, and S. Plotkin. Fast approximation algorithm for minimum cost multicommodity flow. *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 493–501, 1995.

[16] V. Kumar. An approximation algorithm for circular arc colouring. *Algorithmica*, 30(3):406–17, 2001.

[17] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics.* Springer, 2000.

[18] M. Pioro and P. Gajowniczek. Simulated allocation: a suboptimal solution to the multicommodity flow problem. *Teletraffic Symposium, 11th. Performance Engineering in Telecommunications Networks. IEE Eleventh UK*, page 31/1, 1994.

[19] H. Pirkul, J. Current, and V. Nagarajan. The hierarchical network design problem: a new formulation and solution procedures. *Transportation Science*, 25(3):175–82, 1991.

[20] N.G.F. Sancho. The hierarchical network design problem with multiple primary paths. *European Journal of Operational Research 96*, 1996.