Optimization of Path Protection

Kasper Bonne Rasmussen

Supervisor:
Thomas K. Stidsen

Informatics and Mathematical Modeling
Technical University of Denmark

December 5, 2005

Abstract

As connection oriented network solutions are used more and more for telecommunication, live television broadcast, streaming of sound and video, solving the timing issues in voice over IP (VoIP) and much more, it becomes increasingly important to protect such networks from link failures as they do not have the inherent protection of a packet switched network.

Protection from link failure can be costly since additional capacity is needed for backup paths. In this theses we will present a path protection method called Shared Backup Path Protection which has a capacity requirement very close to the lower bound. We will show how the model is implemented and test the final program on several well known networks.

Kgs. Lyngby, December 5, 2005

Kasper Bonne Rasmussen, s992293

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As digital communication networks becomes ever larger, the likelihood of an equipment or link failure within such a network becomes equally great. It is important that these networks can recover from e.g. a link failure without loosing the connection in order for us to be able to trust the network as a reliable media.

One way of providing this protection from failure is by using a *path protection* scheme. The concept of path protection is to somehow reserve enough bandwidth on an alternate path through the network, to allow for a single link failure to occur on any given link in the network. If the failure occurs on a path that carries data, the data uses the alternate route.

On figure 1.1 the data follows an alternate path when the failure occurs, but in order to use this path it must first be found, and we must ensure that there is enough capacity on the new path to handle the data flow. All this takes time, and if we want to switch to the new path without interrupting the connection, the alternate path must be found in advance.

An advantage of finding the alternate- or backup paths in advance is that

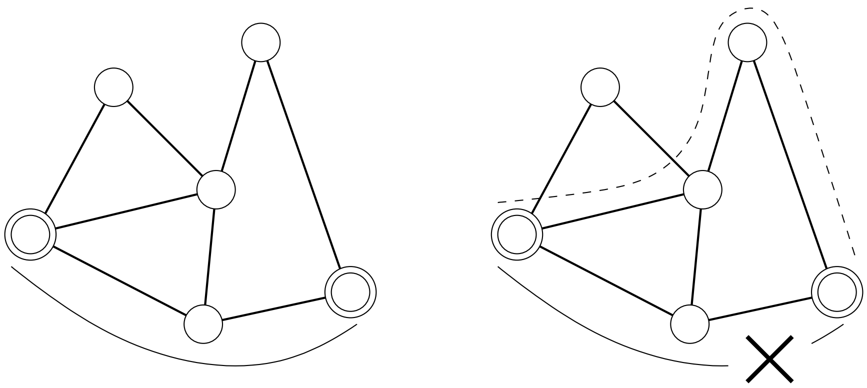

Figure 1.1: The idea behind path protection.

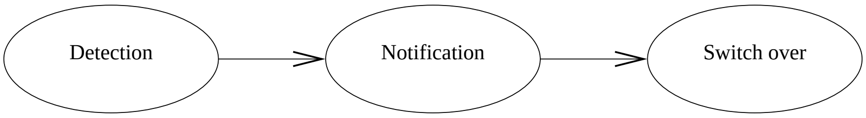

Figure 1.2: Path protection chain of events.

we can switch between them very fast. A disadvantage is that the backup path can not reuse any of the links in the primary path, since we do not know in advance which particular link on the path will break.

The problem is complicated further if there are more than one connection on the network (which there always are in any real world network). Maybe the alternate path can only handle some of the connection load and therefore the backup paths have to be spread out. Maybe the path we had intended to use as backup is now used by another connection.

In order to ensure that we have reliable backup paths we must not only find alternate paths, we must also reserve some bandwidth for them so we are sure they will be available if needed. All these issues must be handled by the path protection scheme we choose.

In this thesis we will explore a path protection scheme called Single Backup Path Protection (sbpp). To better understand the qualities of sbpp we will start by looking at a range of different protection methods. This will also give an idea of what to expect, and what to look for in a path protection scheme.

1.1 Chain of events

Before we begin looking at the individual methods it is important to know how the chain of events are from fault detection to the data flow have been established on the backup path (see figure 1.2).

First of all the fault must be detected. That goes without saying but when we are evaluating the recovery time of the different protection schemes it is important who i.e. what node, is responsible for detecting the fault.

When a fault is detected this information must be conveyed to the network entity responsible for taking the appropriate action. Sometimes that will be the same node that discovers the fault, and sometimes not. The information is transmitted in the form of a Fault Indication Signal (FIS), and the length it has to travel is a vital component of the recovered time [3].

Finally a switch over mechanism is needed to redirect traffic to the backup path. In MPLS[1] networks this is actually two nodes, the first is defined as a path source label switch router (PSL), and the second is a path merge label switch router.

[1]Multi protocol Label Switching

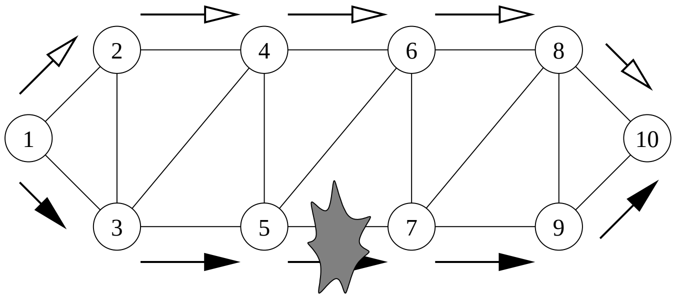

Figure 1.3: 1+1 protection.

The details of how this is accomplished is not the focus of this theses. When we review the different protection methods, it is important to know that these tasks must be handled by nodes that are capable of of doing so, and that all nodes does not necessarily possess these capabilities.

1.2 Protection methods

The different protection methods fall roughly into two categories: Path Protection and Link Protection. Path protection is characterized by the fact that the connection between two points is seen as a continues path, and it is this path we are trying to protect. That means that if the path breaks, i.e. any of the links on the path breaks, we must use an alternate path. In path protection we generally call the path that carries the data "the primary path" and the alternate path is "the backup path".

In Link protection schemes we protect the individual links without regard for how the rest of the path is laid out. That means that if a link breaks, we try to restore the connection between the two points on either side of the link failure.

Although we distinguish between these two groups we will look at both types of methods since they both offer protection of the traffic from source to destination. We will now look at several of the known path/link protection schemes and try to get an idea of the strengths and weaknesses of each one.

1.2.1 1+1 protection

One form of path protection that is widely used today is 1+1-protection, where the data is sent simultaneously via two paths that do not share any links. (see figure 1.3).

The most important advantage of this method is the extremely fast recovery time. If a link failure occurs anywhere on the primary path, the data flow will be interrupted and the destination node will just switch to the backup path.

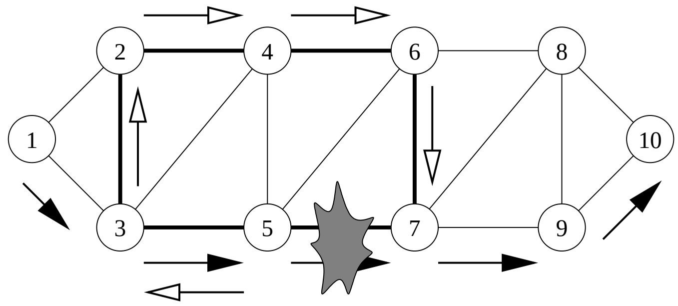

Figure 1.4: Ring (and p-cycle) protection.

Since there is no signaling needed to change paths the switch is almost instantaneous, however this ultra fast recovery time comes at a prize.

When we look at the amount of resources 1+1 protection consumes, a different picture emerges. Let's say we have a network and we need to transport some data from node a to node b using 1+1-protection. First we need to get two paths to the destination node, as the first path (the primary path) p_p we choose the shortest path between a and b. As the second path (the backup path) p_b we choose the next best path, we just used the shortest path so p_b must be at least as long as p_p or longer.

$$p_b \geq p_p$$

That means we use at least 100% extra capacity on the network in order to protect the traffic, probably more.

It seems 1+1 protection is very good in terms of recovery time, but very expensive in terms of network capacity.

1.2.2 Ring Protection

Another protection method used is rings. The ring protection method uses additional capacity allocated in rings to protect the links. If a link on the ring breaks the data is just sent the other way around the ring. (see figure 1.4)

This form of protection is called link protection because it protects individual links and not the entire path. If a link between node i and j fails the data is still routed to node i via the original path, then via the ring, around the failed link to node j and then on to the destination.

This can create some unnecessarily long paths if the rings are big, and is not very efficient in terms of capacity however it is a very simple protection method to implement and is therefore used in many practical situations.

1.2.3 p-cycle Protection

A related link protection method is p-cycles. The p-cycle protection method is very similar to rings, however the p-cycle method distinguishes between

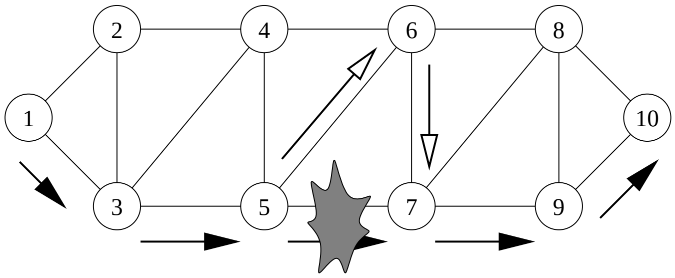

Figure 1.5: Local backup protection.

links that are *on* a cycle and links which connects two nodes on a cycle but are not on the cycle themselves. On figure 1.4 an on-cycle link could be 2-3 and an off cycle link could be 3-4.

If an on-cycle link breaks the data can just travel the other way around the cycle like in ring protection. If a off-cycle link (straddling link) breaks the data can be sent either way around the cycle that each of the ends are connected to [14].

The problems with p-cycles are the same as with rings, the paths can get very long, however since p-cycle protection also protects the straddling links it is more efficient than ring protection in terms of capacity. The recovery time is comparable to that of link protection.

1.2.4 Local Backup Protection

This is another link protection scheme. It tries to find the optimal "replacement path" for all the links in the network, so that no matter which link fails is has an optimal replacement ready for that link (see figure 1.5).

It is more efficient in terms of capacity than ring- or p-cycle protection, since the backup paths will use the shortest (or cheapest) route to restore the link and not a possibly much longer ring. however this comes at a price.

In order for each node in the network to be able to redirect the traffic to a backup path, each node must have PLS capabilities, and in order for any node to be an "endpoint" for other backup paths they must also have PML capabilities.

The fact that all the nodes in the network needs advanced capabilities makes this method very expensive and complex to implement. The recovery time of local backup protection is reasonably good, but the complexity can lead to low resource utilization [3].

1.2.5 Local Dynamic Backup Protection

Local dynamic backup protection is very similar to local backup protection. In this scheme we try to reduce the demands on each individual node by

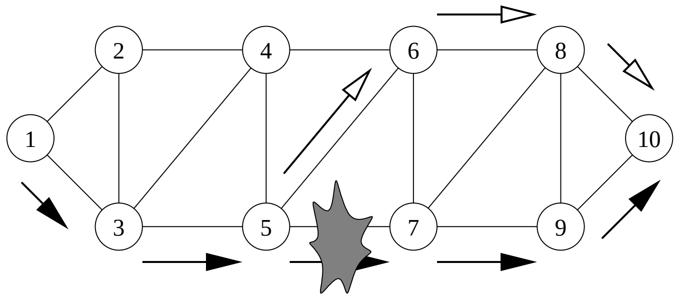

Figure 1.6: Local Dynamic Backup Protection.

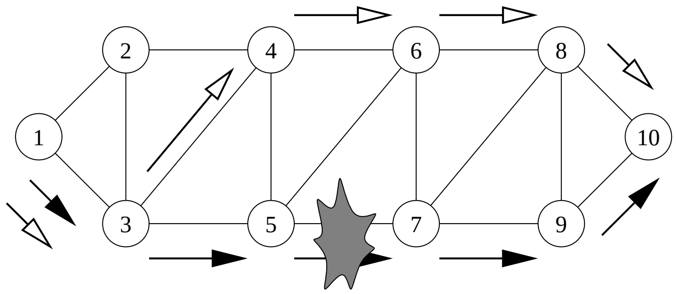

Figure 1.7: Global Rerouting.

moving the PML functionality to the destination node (see figure 1.6).

This means that we must find an alternate path from each node on the primary path to the destination node. This can however introduce a problem on the nodes close to the destination. Not a capacity problem since only one of the backup paths can ever be used at a time, but they must still be stored on the nodes as possible backup paths, so a node can end up having to manage a long list of paths. (one for each link on the primary path).

The recovery time is about the same as for local backup, but the capacity utilization is probably a bit better.

1.2.6 Global Rerouting

In global rerouting protection we have a completely new backup path for *all* primary paths in the network for *every* link. In other words for each and every possible link failure, we have a completely new set of paths supplying the demands on the network (see figure 1.7).

This is a protection method that differs from the others in that it is totally impractical to implement in a network, since a link failure anywhere in the network would tear down *all* the connections in the network and reroute them in an optimal way.

The reason this is interesting is that in terms of the capacity needed to protect the traffic, no protection method can ever hope to do better

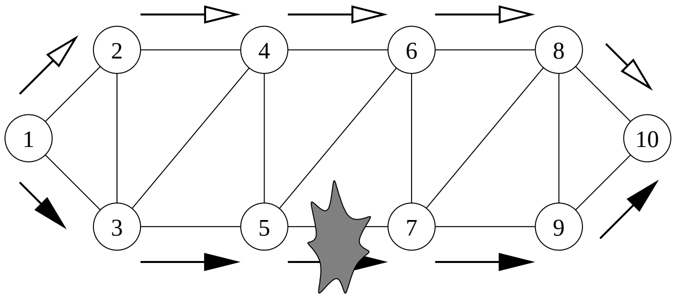

Figure 1.8: Single Backup Path Protection.

than global rerouting. This makes it useful as a lower bound for protection methods and provides the means of comparing all the methods to see how well they perform

The comparison is *only* on the amount of capacity needed to protect the network traffic against link failure. It does not take into account restoration time, node complexity, price etc.

1.2.7 Single Backup Path Protection

Single Backup Path Protection (sbpp) is the focus of this theses and works by reserving capacity for a backup path just like 1+1-protection, but unlike 1+1-protection it does not transmit data on the reserved backup path until it is needed i.e. a link on the primary path fails. (see figure 1.8).

This small change enables us to route several backup paths over the same links and only pay for the amount of traffic the worst case link failure would cause.

On figure 1.9 the same network using the two different protection methods, single backup path protection and 1+1 protection, are shown next to each other. As we can see from the figure the middle link does not need to receive capacity for both backup paths since they can never be used at the same time[2].

In this small example we save a single unit of capacity, which might not seem that impressive, however here we only have two demands. As the number of demands becomes higher so does the number of backup paths that can use the same links and the benefit of using sbpp will be more clear. Also as we shall see in chapter 6 one demand can be split up, thereby taking advantage of the shared backup paths.

The only drawback with single backup path protection is that the FIS needs to be sent back to the source node before the switch over can happen. This does have one advantage though: All the complexity is on the rim of the network so the nodes in the middle does not have to be very expensive.

[2]We work under the assumption that only one link failure can occur at a time.

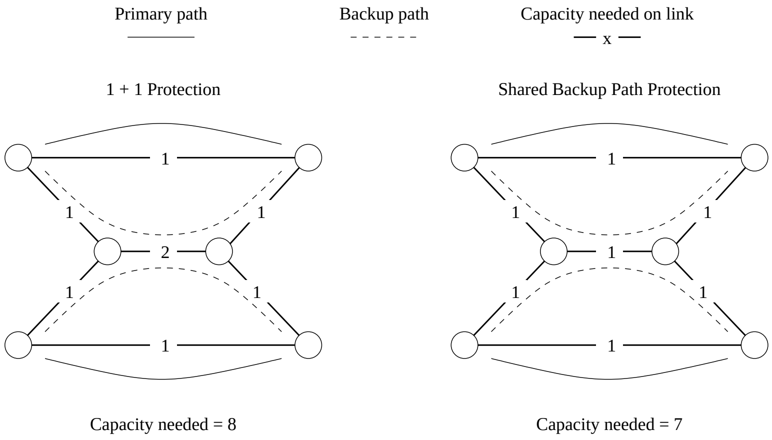

Figure 1.9: Sbpp vs. 1+1 protection

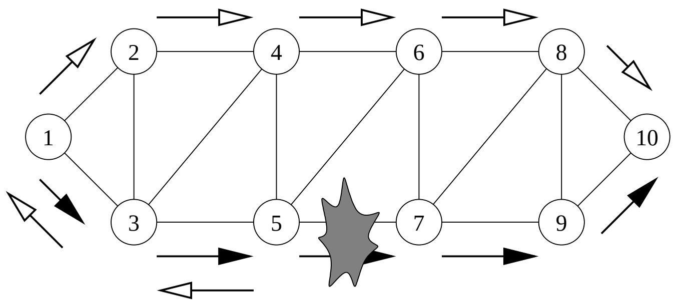

Figure 1.10: Reverse Single Backup Path Protection.

1.2.8 Reverse Single Backup Path Protection

Reverse single backup path protection (rsbpp) is an attempt to minimize the data loss caused by the restoration time in sbpp. When a link failure is detected the data is sent back to the source node along with the FIS, so that when the source node makes the switch over it can send all the data along the backup path. There will still be a "hole" in the data flow, but the data that was transmitted in the time it took to send the FIS is not lost.

It is clear that there must be a poorer capacity utilization than with sbpp. Exactly how poor is investigated in chapter 6.

1.3 Methods compared

In order to get some idea of the resource efficiency of the different protection methods they have all been compared to global rerouting, which is the lower bound for protection methods on circuit switched networks [13]. The result of that comparison can be seen on figure 1.11.

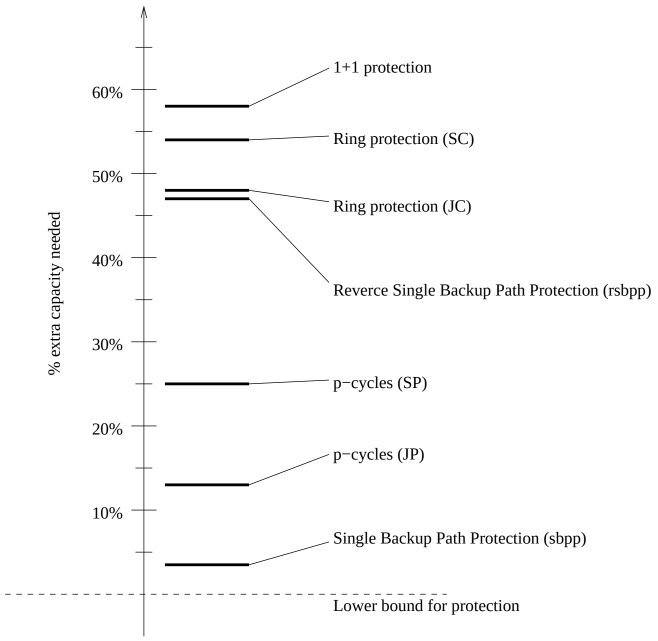

Figure 1.11: Various known protection methods, and their average capacity requirements compared to the lower bound.

The most capacity efficient path protection method on figure 1.11 is Single Backup Path Protection. The rest of this theses will explore various properties of sbpp such as the capacity requirements, the running time and the implementability.

Chapter 2

Single Backup Path Protection

Single Backup Path Protection is a protection method in which you allocate two paths from the source node to the destination node, in which none of the links from one path is used in the other, a so called *disjoint path pair*.

Another feature of Single Backup Path Protection (sbpp) is that since the backup paths are not used until a link failure actually occurs, multiple backup paths can use the same links at no additional cost as long as enough capacity is reserved on all the links to handle the worst case failure situation.

In this thesis a failure situation is when a link in the network breaks, and the traffic on all the paths using that link has to use there backup paths. The model can handle one failure situation at a time, i.e. one link failure at a time.

2.1 The model

The model consists of a master problem and a subproblem linked together by a column generation algorithm. The objective of the master problem is to find the cheapest set of disjoint path pairs, that will satisfy the current demand.

The disjoint path pairs are generated by the sub problem based on the prices of the spans in the network. Initially the prices are the *weight*[1] of the spans, but when the master problem is solved the prices are updated by the column generation algorithm, to reflect which spans are "most busy" thus enabling the sub problem to find the most optimal disjoint path pairs.

As new disjoint path pairs are found the master problem has more to choose from and the prices of the spans are updated to reflect the new

[1] The initial weight can be the same for all the spans in the network *or* based on the length of the span *or* something else. The overall solution is evaluated based on the initial price of the spans.

"situation". This loop continues until no disjoint path pair that can improve the overall solution can be found.

In order to make the model manageable, a few assumptions have been made that are not necessarily true in real life but necessary for the model to work.

- We assume that we have infinite bandwidth on all links. We will not use huge amounts of bandwidth, in fact the hole point of the protection scheme is to use as little bandwidth as possible, but it ensures that we can always place a path on a set of links.
- We also assume that the links are infinitely dividable, i.e. we can use *any* percentage of any link.
- Lastly we assume that only one link can fail at a time. This assumption might seem restrictive, but it is unlikely that two links would fail at exactly the same time, and it is necessary for the model to work.

2.2 Notation

Before we get too far into the description of the master- and sub problem, we will briefly describe how indices are used in this formulation.

The indices $i, j, k, l, q, r \in \{1, .., N\}$ are node indices. E.g. ij is the bi-directional span between node i and node j. Even though they all represent node indices it is very important to notice witch set of indices are used, as each set have a different meaning.

ij	is the span from node i to j when the span is working.
qr	is used when the two indices represent a span that is broken
kl	is only used when specifying a demand.
p	represents a disjoint path pair

Apart from that, an index in parentheses (ij) means the *oriented* link from node i to node j. ij with no parentheses means, as mentioned, the bi-directional span from i to j.

The words "link" and "span" both refer to the connections between nodes. The word link is used when ever we talk about the oriented connection between two nodes, and span is used for the bi-directional connection.

2.3 The master problem

In the master problem $x_p^{kl} \in R_+$ is the flow on the disjoint path pair p that satisfy a demand from node k to node l and $y_{ij} \in R_+$ is the required capacity on span ij.

The objective of the master problem is to minimize the total required capacity of the protection for all the demands (2.1). In other words minimize the amount we have to pay for all the spans we use. The spans have a fixed price so you could say that we are minimizing the usage of spans.

This objective is of course subject to some constraints. First of all, all demands must be met (2.2). The second constraint (2.3) will be described at length below, but basically it states that all primary paths must pay for the spans on that path, and the backup paths must pay if the primary path fails. Finally no paths can have a negative data flow. The same goes for individual links (2.4).

One formulation of the master problem looks like this:

Minimize:

$$\sum_{ij} C_{ij} \cdot y_{ij} \tag{2.1}$$

Subject to:

$$\sum_{p} x_p^{kl} \geq D^{kl} \quad \forall kl \tag{2.2}$$

$$\sum_{kl} \sum_{p} a_{p,ij}^{kl} \cdot x_p^{kl} + \sum_{kl} \sum_{p} a_{p,qr}^{kl} \cdot b_{p,ij}^{kl} \cdot x_p^{kl} \leq y_{ij} \quad \forall (il, qr) | ij \neq qr \tag{2.3}$$

$$x_p^{kl}, y_{ij} \in \mathbf{R}_+ \tag{2.4}$$

D^{kl} is the demand between node k and l. C_{ij} is the price of span ij.

2.3.1 The second constraint

The second constraint can be confusing, so we will go through that step by step. The first important thing to notice is the meaning of the indicator variables a and b:

- $a_{p,ij}^{kl} = 1$ if link ij is part of the primary path in x_p^{kl} and 0 otherwise.
- $a_{p,qr}^{kl} = 1$ if link qr is part of the primary path in x_p^{kl} and 0 otherwise.
- $b_{p,ij}^{kl} = 1$ if link ij is part of the backup path in x_p^{kl} and 0 otherwise.

Now if we look at the first part of the constraint $\sum_{kl} \sum_p a_{p,ij}^{kl} \cdot x_p^{kl}$, it is clear that if the span ij is part of the primary path it will force the variable y_{ij} to be raised until the constraint is satisfied. Keep in mind that we only pay for the y_{ij}'s so this way we always pay for a span if it is part of the primary path.

The second part of the constraint $\sum_{kl} \sum_p a_{p,qr}^{kl} \cdot b_{p,ij}^{kl} \cdot x_p^{kl}$ has to do with the backup paths. When we know the meaning of a and b it becomes clear what is going on. If the span qr is part of the primary path and the span ij

is part of the backup path it will force the variable y_{ij} to be raised until the constraint is satisfied.

In other words we only pay for a span on the backup path in those situations where that span can be used *if* a span on the primary path fails.

The first and second part of the constraint can never be activated at the same time, since the same link can not exist in the primary path *and* the backup path at the same time $a^{kl}_{p,ij} + b^{kl}_{p,ij} \leq 1$. This is what we mean when we say that two paths are disjoint. The disjointness is ensured by the sub problem.

Figure 3.3 on page 22 is a visualization of the master problem, and it might help to look at it now. The figure will be described in more detail in chapter 3

Almost all the data needed in the master problem is already available: All the spans are given by the network data, as are the span prices. The demands are fixed and must be provided separately. The only thing missing is the actual disjoint path pairs. To find them we must define a new problem.

2.4 The sub problem

In the subproblem $x_{(ij)} \in \{1, 0\}$ is the *oriented* flow on link (ij) where (ij) is a link on the primary path. $y_{(ij)} \in \{1, 0\}$ is the *oriented* flow on link (ij) where (ij) is a link on the backup path. $z_{ij,qr} \in \{1, 0\}$ is an indicator variable described in detail below.

Notice that when ever a pair of indices is in a parentheses it means they represent an *oriented* link. If there are no parentheses it means they represent a bi-directional span.

The objective of the sub problem is to find new disjoint path pairs through the network. The price of these paths is what we want to minimize (2.5).

The first two constraints are flow constraints, one for the primary path (2.6) and one for the backup path (2.7). They ensure that all data that "flows" into a node will also flow out again, except in the source node and the destination node.

The next constraint (2.8) ensures that a link can at most be used by *one* of the paths, and if it is used in one direction, no path, not even the same one can use it again in the other direction. This is okay since there is always a positive cost on the links, and therefore it is never useful to go back and forward along the same link.

The final constraint (2.9) ensures that if a link on the primary path breaks, an indicator variable $z_{ij,qr}$ is set for each link on the backup path. One formulation of the sub problem looks like this:

Minimize:

$$\sum_{(ij)} c_{(ij)} \cdot x_{(ij)} + \sum_{ij} \sum_{qr} \beta_{ji,qr} \cdot z_{ji,qr} \tag{2.5}$$

Subject to:

$$\sum_{(ij)} x_{(ij)} - \sum_{(ji)} x_{(ji)} = \begin{cases} 1 & \text{for } i = k \\ -1 & \text{for } i = l \\ 0 & \end{cases} \quad \forall i \tag{2.6}$$

$$\sum_{(ij)} y_{(ij)} - \sum_{(ji)} y_{(ji)} = \begin{cases} 1 & \text{for } i = k \\ -1 & \text{for } i = l \\ 0 & \end{cases} \quad \forall i \tag{2.7}$$

$$x_{(ij)} + x_{(ji)} + y_{(ij)} + y_{(ji)} \leq 1 \quad \forall \{ij\} \tag{2.8}$$

$$x_{(qr)} + x_{(rq)} + y_{(ij)} + y_{(ji)} - 1 \leq z_{ij,qr} \quad \forall (ij, qr) | ij \neq qr \tag{2.9}$$

where $c_{(ij)}$ is the price of link ij in the network. These prices are updated every time the master problem is solved. β is the cost vector taken from the dual row prices of the master problem.

The link prices of the network $c_{(ij)}$ are defined as:

$$c_{(ij)} = \sum_{qr} \beta_{ij,qr}$$

where β is the cost vector described above. In other words the price of a link in the network is determined by the sum of all the row prices of the rows in the matrix representing that link. More informally we can say that the link price is a measure of how much we are willing to pay for extra capacity on that link.

2.4.1 The z values

The final constraint (2.9) ensures that the value of $z_{ij,qr}$ will be raised if ij is used by the backup path *and* qr is used by the primary path.

This enables us to use the $z_{ij,qr}$ values in the objective function instead of the y_{ij}'s and in doing that, we only pay for the links on the backup path, if using that link will cause us to need to reserve more capacity.

The capacity needed on each link is given by the following formula:

$$C_i = \sum_p P_{pi} + \max\left(\sum_\rho B_{\rho i}\right)$$

in other words: The capacity C_i of a link i is the sum of the capacity of all the *primary* paths using that link $\sum_p P_{pi}$ since there is data running on those, we have to reserve capacity for that no matter what. Plus the maximum capacity needed by *backup* paths using that link.

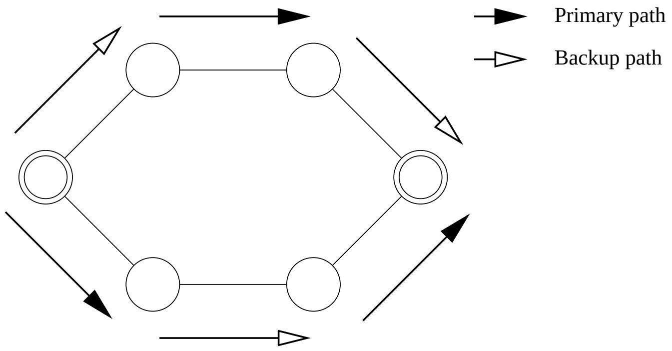

Figure 2.1: A possible primary and backup path for the data if the constraints where combined in a single constraint.

Since a backup path is only used if the primary path p fails, a situation could occur where we already need to reserve capacity for a backup path, and a second can then use the link for free because it will never use the capacity at the same time as the other one.

The way this is accomplished in the sub problem is to use the master problems row prices as a cost vector for the $z_{ij,qr}$ values. That way only the ones with a row cost greater than 0 will contribute to the prize of the sub problem.

2.4.2 Combine the flow constraints?

At first glance it might seem that the two flow constraints (2.6) and (2.7) can be combined in a single constraint. That would simplify the problem and save some space in the implementation. The new constraint would then look like this:

$$\sum_{(ij)} x_{(ij)} - \sum_{(ji)} x_{(ji)} + \sum_{(ij)} y_{(ij)} - \sum_{(ji)} y_{(ji)} = \begin{cases} 2 & \text{for } i = k \\ -2 & \text{for } i = l \\ 0 & \end{cases} \qquad \forall i$$

However this is not possible, since the primary path and the backup path could then be mixed. Traffic entering a node via the primary path could exit the node via the backup path. Figure 2.1 illustrates this mix.

2.5 Complexity of the master- and sub problem

In order to describe the complexity of the two problems we will first briefly introduce some complexity classes [6], [9].

P is the class of problems which can be solved in polynomial time (quickly).

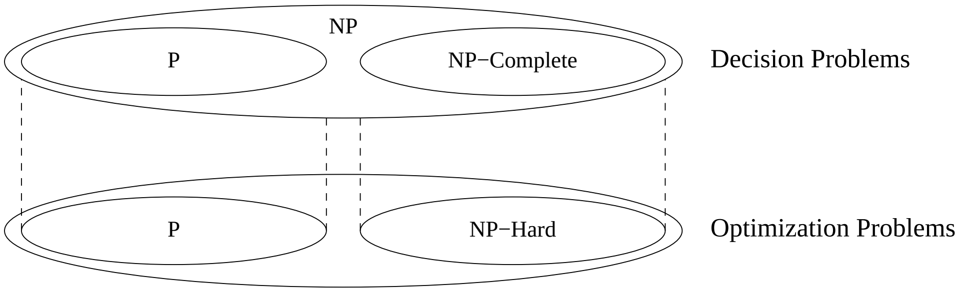

Figure 2.2: The relationship between the different complexity classes (Assuming $P \neq NP$).

NP is the class of problems for which answers can be checked by an algorithm whose run time is polynomial in the size of the input. Note that this does not require or imply that an answer can be found quickly, only that any claimed solution can be verified quickly.

$NP - Complete$ is the subset of NP for which no other NP problem is more than a polynomial factor harder. That means that any NP-complete problem can be "reduced" to any other NP-complete problem in polynomial time. This class contains the hardest NP problems.

These complexity classes are defined for so called decision problems. When a decision problem is proved to belong to the class of NP-complete problems, then the corresponding optimization problem is said to be NP-hard, see figure 2.2.

The master problem is an lp problem so it can be solved in polynomial time, and is as such trivial in this context. The sub problem however is more interesting.

The sub problem is an optimization problem which tries to find the cheapest disjoint path pair connecting k and l. The corresponding decision problem is:

"Is there a disjoint path pair with a price $\leq k$?"

First of all it is easy to determine if a proposed solution has a price less than k, just verify that the disjoint path pair does in fact connect k and l and that no links are used in both paths, then just sum up the prices of the links, so it is clearly in NP.

Secondly it can be written as a binary integer program (BIP), and then we can ask: does this BIP have a feasible solution?

Since the decision problem is in NP and can be "reduced" to a BIP (which is NP-complete) the decision problem must be NP-complete. Given that the decision problem associated with the sub problem is NP-complete, the original sub problem "what is the cheapest path?" must be NP-hard.

Chapter 3

Converting to COIN

In this chapter we will look at how the model is converted so it can be implemented and solved by a computer. The algorithms and implementation details are described in chapter 4 and chapter 5 respectively, this chapter will focus on the conversion.

In order to better explain various details in this chapter an example network is used, this network can be seen on figure 3.1. Although some figures and explanations are based on this network, it is a trivial matter to extend the arguments to any other larger network. It is used for it's simplicity.

3.1 Converting the problem to COIN

Before we can start writing algorithms to solve the master- and sub problems we need to convert the model into something we can do calculations on with a computer.

We are using the COIN[1] [1] api so that will dictate how the conversion must take place. The coin model can be seen on figure 3.2. It consists of 7 parts: The main matrix which corresponds to the constrains of the problem, an upper and lower row bound which can be manipulated to create *greater-than-or-equal*, *less-than-or-equal* or *equal* constraints, upper and lower column bounds used to bound the decision variables, and finally the objective values which act as a cost vector for the decision variables in the objective function.

The three types of constraints *greater-than-or-equal*, *less-than-or-equal* and *equal* can be created using the row bounds in the following way:

[1]COmputational INfrastructure

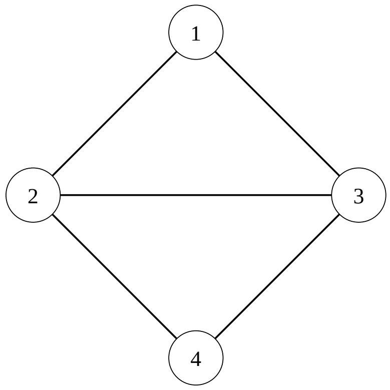

Figure 3.1: The `testnet2` network. A small network used for illustrative purposes.

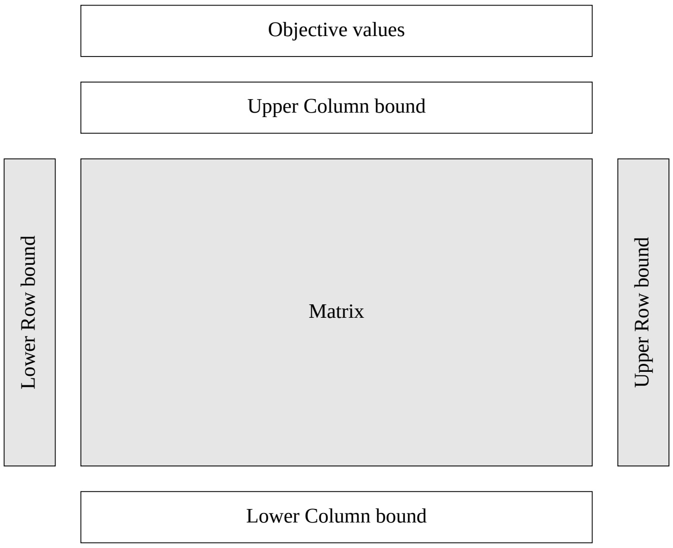

Figure 3.2: Coin model. The shaded part corresponds to the data shown in figure 3.3 and figure 3.5

Constraint	Lower Row bound	Upper Row bound
$a \leq x \leq b$	a	b
$a \geq x \geq b$	b	a
$x = a$	a	a

Despite that, we are going to convert all the constraints to *less-than-or-equal* or *equal* constraints. It will simplify the creation of the row bound arrays and ensure that all the dual prices have the same sign.

3.2 Master problem

First we will convert the problem to a "standard" form. That means all the decision variables must be present in the objective function and appear only on the left side of the constraints, and all constraints must be *less-than-or-equal* or *equal* constraints.

The master problem on standard form looks like this:

Minimize:

$$\sum_{ij} 0 \cdot x_{ij} + \sum_{ij} C_{ij} \cdot y_{ij}$$

Subject to:

$$-\sum_{p} x_p^{kl} \leq -D^{kl} \quad \forall kl \tag{3.1}$$

$$\sum_{kl}\sum_{p} a_{p,ij}^{kl} \cdot x_p^{kl} + \sum_{kl}\sum_{p} a_{p,qr}^{kl} \cdot b_{p,ij}^{kl} \cdot x_p^{kl} - y_{ij} \leq 0 \quad \forall (il, qr) | ij \neq qr \tag{3.2}$$

$$x_p^{kl}, y_{ij} \in \mathbf{R}_+ \tag{3.3}$$

The y_{ij} has moved to the left side of the (3.2) constraint, and the demand constraint (3.1) has been multiplied with -1 to get a *less-than-or-equal* constraint. In the objective function the sum $\sum_{ij} 0 \cdot x_{ij}$ has been added.

Now that the master problem is on standard form we can replace all the constraints with a matrix A and all the decision variables with a vector x. The problem then looks like this:

Minimize:

$$c^T x \tag{3.4}$$

Subject to:

$$lb_{row} \leq \mathbf{A}x \leq ub_{row} \tag{3.5}$$

Where c^T is a vector containing the costs of the decision variables, i.e. 0 for the old x'es and the price of the links C_{ij} for the old y's.

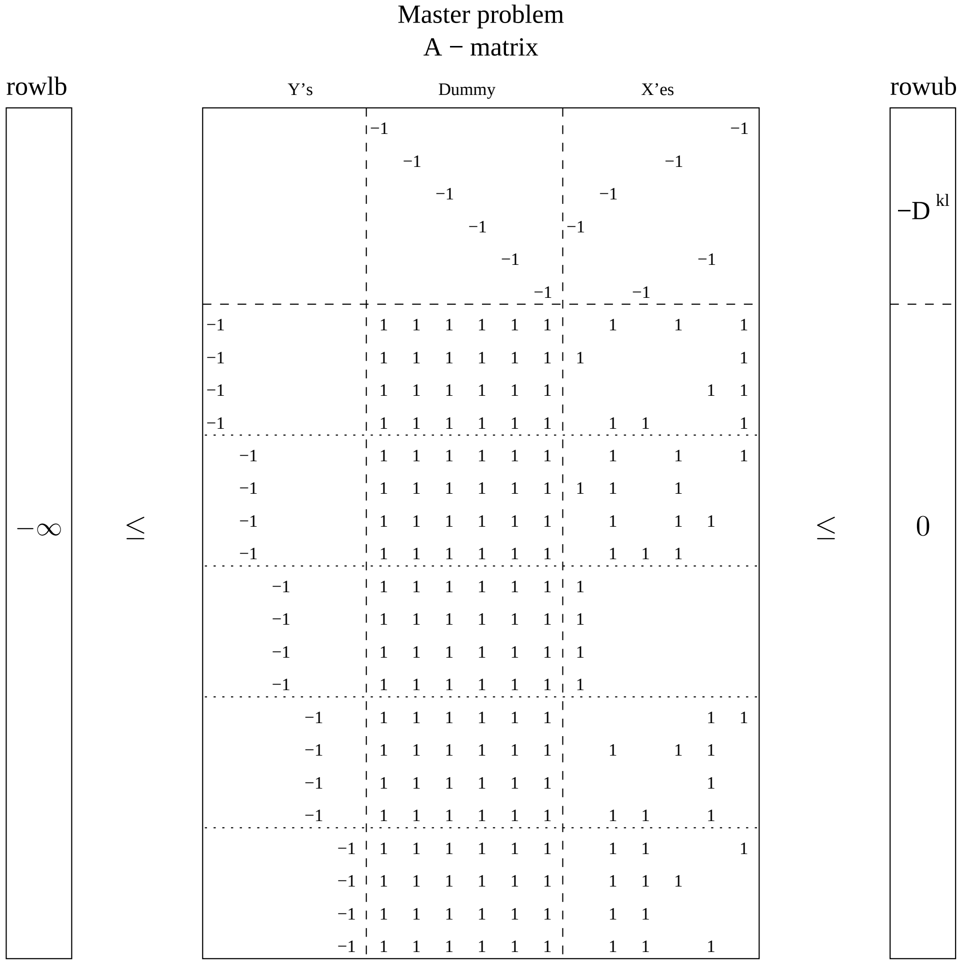

Figure 3.3: The A matrix and the upper and lower row bounds of the master problem. To save space, only the x-columns that make up the optimal solution are included. This example represents the `testnet2` network.

lb_{row} and ub_{row} are the lower and upper row bounds. Figure 3.3 shows the A matrix and the lower and upper row bounds for a the test network in figure 3.1

3.2.1 Master problem A matrix

In figure 3.3 each column of the A matrix corresponds to one decision variable in the master problem. Each row of the A matrix corresponds to one constraint in the master problem.

In the following description I will refer to the specific situation described by figure 3.3, but the explanation applies equally well to any problem.

The first six rows represent the demand constraint (3.1). The number of demands is $n(n-1)/2$ in this case 6. All demands must be met, that means that all rows in that first block must be covered by at least one '-1', in order to balance the corresponding number in the D^{kl} vector.

The rest of the matrix is divided into L equal parts of $L-1$ rows, where L is the number of spans in the network, in this case 5. The L parts each represent one span, and the $L-1$ rows in each block represent the number of other spans that can be broken while you use this span, or the number of *failure situations*. The reason there are not L rows pr. span is that you can not use "this span" if it is broken.

The price in the objective function is 0 for all the x-columns, so what determines if one column (disjoint path pair) is better than another? The answer is of course how many, and which, spans the the disjoint path pair use.

The dummy columns right next to the y-columns are added from the beginning to make sure we have a feasible solution, but they use all the spans in the network[2] and are thus *very* expensive. Every time we use one unit of flow on one dummy path, all the y's have to be increased by one in order to stay within the constraints.

As we solve the subproblems and thereby find better disjoint path pairs through the network, we add the actual x-columns. The spans used by a particular disjoint path pair determines which rows in the rest of the column must be one.

A disjoint path pair consists of two paths, a primary and a backup. If a span s is part of the primary path it must be paid for, no matter which other spans in the network break down. In other words all the failure situations under s must be paid for. In the first real (not dummy) x-column on figure 3.3, span number 2 is part of the primary path[3].

If a span is part of the backup path it must be paid for only in those failure situations where a span on the primary path fails. In the first real (not dummy) x-column on figure 3.3, span number 0 and 1 are part of the backup path, so they need only be paid for in the failure situation where span 2 breaks.

The columns shown on figure 3.3 are only the y-columns, the dummy columns and the x-columns that make up the final optimal solution to the network on figure 3.1.

3.3 A graphic representation

A more graphic representation of the same solution can be seen on figure 3.4.

For each span on figure 3.4 we must reserve capacity for all the primary

[2]These paths are usually not even possible but that does not matter here. It is a question of making the path so expensive that they will never be a part of the final solution.

[3]Note: The first span is number 0 so span number 2 is the third block.

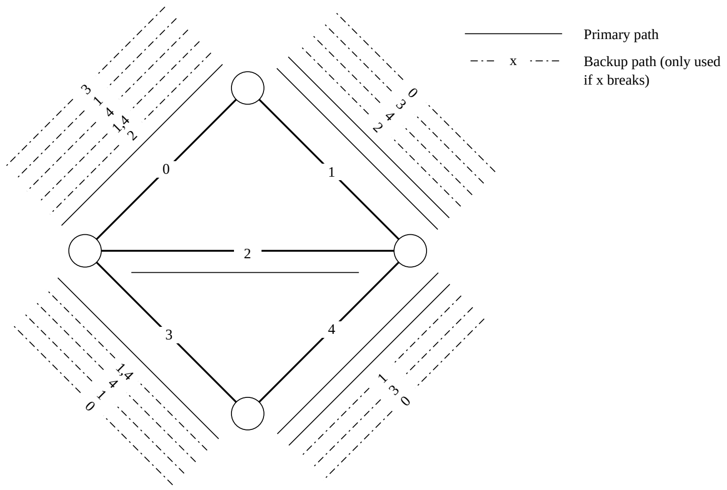

Figure 3.4: This is the optimal solution for the `testnet2` network.

paths and enough extra for the backup paths to cover the worst case scenario. The worst case capacity for all the spans are:

Span 0 One for the primary path and two extra in case span 1 or span 4 breaks. Three in total

Span 1 Two for the primary paths and one extra in case any of the spans 0, 2, 3 or 4 breaks. Three in total

Span 2 One for the primary path

Span 3 One for the primary path and two extra in case span 1 or span 4 breaks. Three in total

Span 4 Two for the primary paths and one extra in case any of the spans 0, 1 or 3 breaks. Three in total

That means that the total number of spans we have to pay for in order to protect the traffic on this network is:

$$3 + 3 + 1 + 3 + 3 = 13$$

3.4 Sub problem

Just like the master problem the sub problem needs to be "standardized". The standardized sub problem looks like this

Minimize:

$$\sum_{(ij)} c_{(ij)} \cdot x_{(ij)} + \sum_{(ij)} 0 \cdot y_{(ij)} + \sum_{ij} \sum_{qr} \beta_{ji,qr} \cdot z_{ji,qr} \tag{3.6}$$

Subject to:

$$\sum_{(ij)} x_{(ij)} - \sum_{(ji)} x_{(ji)} = \begin{cases} 1 & \text{for } i = k \\ -1 & \text{for } i = l \\ 0 & \end{cases} \qquad \forall i \tag{3.7}$$

$$\sum_{(ij)} y_{(ij)} - \sum_{(ji)} y_{(ji)} = \begin{cases} 1 & \text{for } i = k \\ -1 & \text{for } i = l \\ 0 & \end{cases} \qquad \forall i \tag{3.8}$$

$$x_{(ij)} + x_{(ji)} + y_{(ij)} + y_{(ji)} \leq 1 \quad \forall \{ij\} \tag{3.9}$$

$$x_{(qr)} + x_{(rq)} + y_{(ij)} + y_{(ji)} - z_{ij,qr} \leq 1 \quad \forall (ij,qr) | ij \neq qr \tag{3.10}$$

The objective function (3.6) now contains all the decision variables, and in the last constraint (3.10) the $z_{ij,qr}$ is now on the left side of the *less-than-or-equal* sign. The flow constraints remain unchanged even though they have *equal* signs instead of *less-than-or-equal* signs, but that just means that the upper and lower bounds are equal.

Now we need to convert the problem to matrix form just like the master problem.

Minimize:

$$c^T x \tag{3.11}$$

Subject to:

$$lb_{row} \leq \mathbf{A}x \leq ub_{row} \tag{3.12}$$

Where c^T is a vector containing the costs of the decision variables, i.e. the price of the links for the old x'es, 0 for the old y's and the dual row price from the master problem for the old z's.

lb_{row} and ub_{row} are the lower and upper row bounds. Figure 3.5 shows the A matrix and the lower and upper row bounds for a the network in figure 3.1

3.4.1 Sub problem A matrix

Figure 3.5 is an example of the sub problem A matrix as it would look for the `testnet2` network on figure 3.1 with node 1 as source and node 4 as destination.

On figure 3.5 each column of the matrix corresponds to one decision variable in the sub problem. Each row of the matrix corresponds to one constraint in the sub problem.

The x-columns represent *oriented* links in the primary path and the y columns represent *oriented* links in the backup path. The z-columns are the z-values described in chapter 2

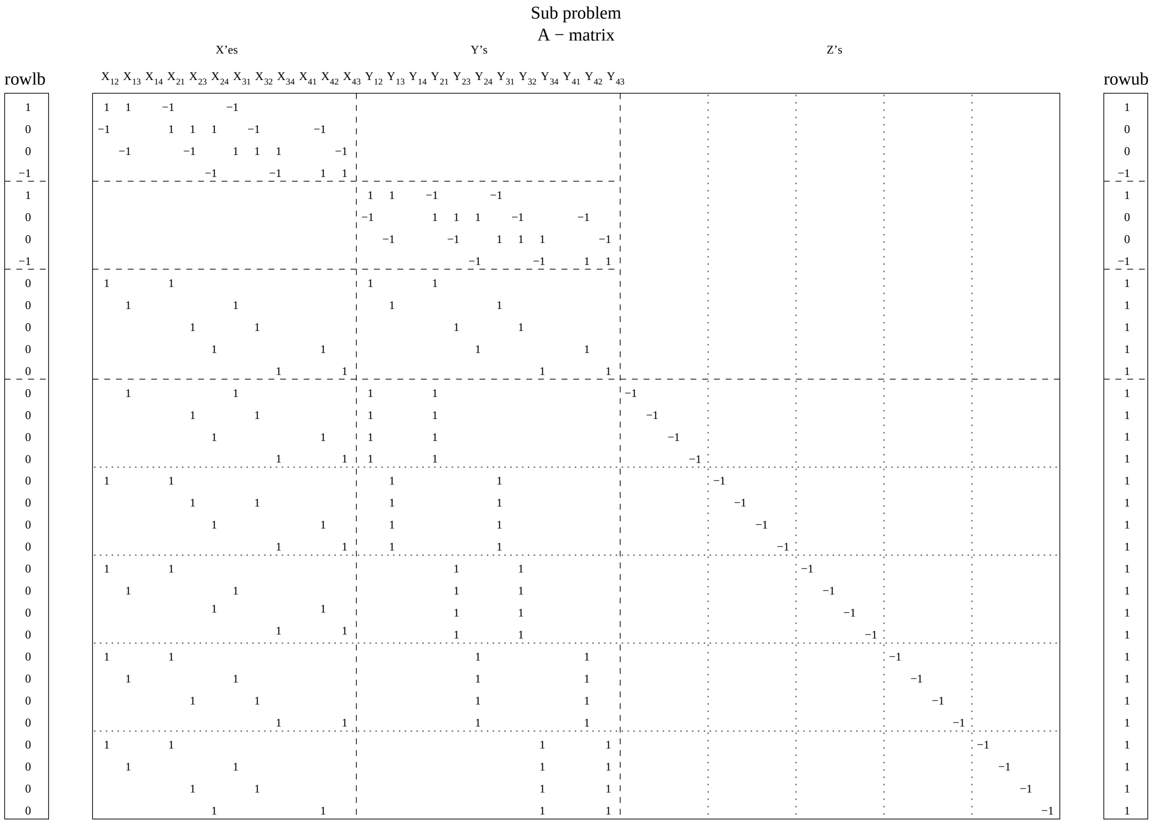

Figure 3.5: The A matrix and the upper and lower row bounds of the sub problem. This example represents the `testnet2` network.

The first two blocks of the matrix are the flow constraints. The lower and upper bound for them are 0 except for the start node and the destination node. For each link (column) there is a '1' and a '-1' representing at which node the link starts and stops.

The third block is the "no share" constraints (3.9). E.g. if the *oriented* link x_{12} is used by the primary path then none of the links x_{21}, y_{12} or y_{21} can be used again. This ensures that the two paths found are disjoint.

The rest of the matrix is divided into L parts of $L-1$ rows, where L is the number of spans in the network, in this case 5. The L parts each represent a span, and the $L-1$ rows represent the possible *failure situations*.

If for example the primary path use the link x_{23} and the backup path use link y_{21} and y_{13}, then $z_{21,23}$ and $z_{13,23}$ must be set to one to stay within the constraints. Since the z-values rather than the y-values are included in the objective function, this forces the backup path to be paid for only if the price of the z-value > 0, i.e. if it increases the worst case situation described in section 3.2.1.

This is because the row prices of the master problem is used as a cost vector for the $z_{ij,qr}$ values as described in section 2.4.1.

Chapter 4

Applied algorithms

In this chapter we will cover the different algorithms used in the program. This is a description of how the algorithms work, any relevant implementation details are covered in chapter 5.

The four main algorithms used in this program are:

- Column generation
- MIP algorithm
- Bhandari algorithm
- LP–solver

Other algorithms are used within the program to accomplish various tasks, but they are either so small that they are not worth mentioning, or so well known that they can be referenced by name, and need no further introduction.

4.1 Column generation

The column generation algorithm is the master control algorithm for the program. It is responsible for the coordination of the various subproblems and the master problem. Algorithm 4.1 shows the basic column generation algorithm in pseudo code.

First we have to read the network and get the demands. This is the only input the algorithm needs, the rest is just calculation. Next we create the master- and subproblem. The master problem implementation is done using the COIN [1] interface to CPLEX. The subproblem can be solved using either the MIP algorithm alone *or* the MIP algorithm sped up by the Bhandari algorithm described in this chapter.

After the initialization we enter the main column generation loop, in which the master problem is first solved in order to get the dual prices for

Algorithm 4.1 Column generation

```
network = ReadNetwork()
demand = InitializeNetworkDemand()
master = InitializeMasterProblem(network, demand)
sub = InitializeSubProblem(network)
while TotalGap > 0 do
   TotalGap = 0
   prices = solve(master)
   update(network, prices)
   for all subproblems do
      dpath = solve(sub)
      if price(dpath) < MasterRowPrice then
         add(dpath, master)
         TotalGap += (MasterRowPrice − price(dpath))
      end if
   end for
end while
```

the spans. Those prices are then used as the prices of the spans in the network.

Next all the subproblems are solved in order to get a number of disjoint path pairs though the network, based on the prices of the spans. If a disjoint path pair has a price that is less than the reduced cost of the demand it satisfy, that path can improve the solution and is therefore added to the master problem.

When all the subproblems have been solved, the master problem is solved again in order to update the network prices and then the subproblems are solved again, and so on. The loop continues until no more disjoint path pairs can be added to the master problem, this will cause the TotalGap to be 0 and the algorithm to stop.

4.1.1 Improvements

A couple of things was tried to improve the performance of the column generation algorithm:

- Pruning columns
- Limit the number of columns added pr. iteration
- Starting with Bhandari's algorithm

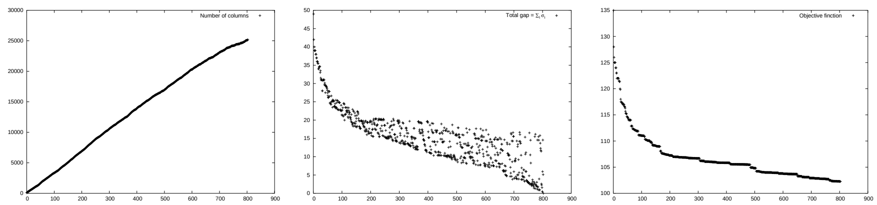

Figure 4.1: This is data from the `11n26s` network with no improvements. Left to right the graphs show: *Number of Columns*, *Total gap* and the *Objective value*.

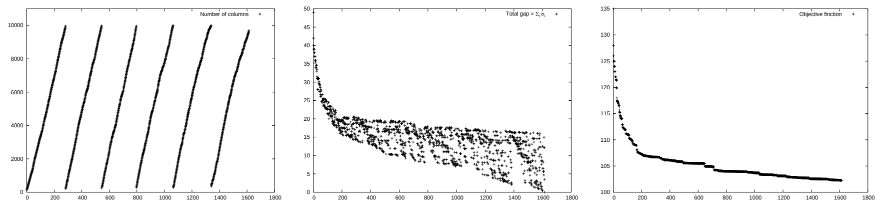

Figure 4.2: This is data from the `11n26s` network when using the first pruning strategy (Prune every column not in basis). Left to right the graphs show: *Number of Columns*, *Total gap* and the *Objective value*.

Pruning columns

The Master problem is taking a lot of time when the problems are getting big. A profiling of the algorithm reveled that, on the large networks, it takes more time to solve the master problem, than it does to solve all the subproblems in a single iteration.

In order to try and fix this problem, a column pruning strategy is implemented to reduce the size of the master problem when it reaches a certain limit. Two different forms of pruning are tested, both activated when the size of the master problem exceed 10,000 columns.

Figure 4.1 shows the data without doing any pruning. What is interesting is that the algorithm finds an optimal solution in about 800 iterations.

Figure 4.2 shows the data generated when the first pruning strategy is implemented. The pruning strategy is to delete *all* unused columns in the master problem whenever the size exceed 10,000 columns. An unused column is one that is not part of the current solution.

This strategy will take the number of columns down to the initial number every time pruning is done. It does not improve the running time. In fact the running time is about 5% longer than the original, but it completes twice the number of iterations so it is significantly faster pr. iteration.

It looks like too many columns are deleted. Obviously there is some time to be gained if only the columns that truly are not needed anymore can be deleted, and the rest left alone.

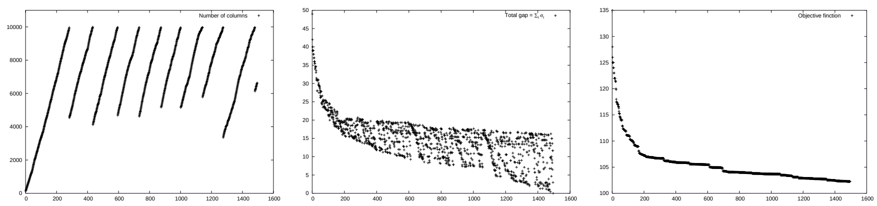

Figure 4.3: This is data from the `11n26s` network when using the second pruning strategy (Prune every column with a positive reduced cost). Left to right the graphs show: *Number of Columns*, *Total gap* and the *Objective value.*

This observation leads me to try a second pruning strategy. This strategy is to delete all the columns with a *positive* reduced cost whenever the size exceed 10,000 columns. Figure 4.3 shows the data from that experiment.

The number of iterations is about 1,000 lower than with the first pruning strategy, however still a lot higher than the original algorithm, and the actual running time is over 20% higher than the original.

With no way of knowing which columns may be part of the solution in the future, it looks like pruning might not be such a good idea.

Limit the number of columns added pr. iteration

Another way to limit the size of the master problem and at the same time speed up the solution of the subproblems is to solve a limited number of subproblems in each iteration. That way the master problem grows less, and only some of the sub problems are solved in each iteration, so both the master problem and the sub problem takes less time pr. iteration. Of course you have to be careful to still solve *all* the subproblems and not just solve the first ones over and over.

Figure 4.4 shows the data from that experiment. The number of columns added every time is 10, so the "Number of Columns" graph is just a straight line and have not been included. Note that this does not mean that only 10 subproblems are solved in each iteration, it means that enough subproblems are solved to find 10 with a negative reduced cost.

It is interesting to see that the algorithm converged in about half the number of iterations compared to the version where you just add all the subproblem columns with a negative reduced cost in every iteration. That together with the other benefits of a smaller master problem and faster subproblems, resulted in a running time of about 9% of the original, that is an improvement by a factor of more then 11.

That looks pretty good, but what if we go further down and add five or two or just one column at a time? It turns out that the best running time is achieved if you only add one column at a time, and doing that will reduce the running time to about 2% of the original. That is a factor of about 50.

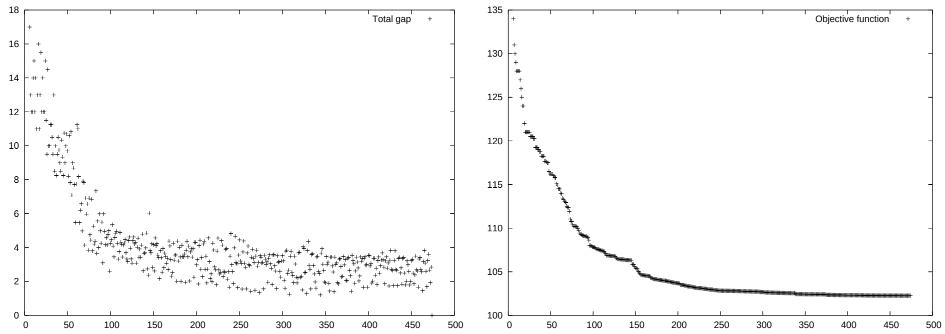

Figure 4.4: This is data from the `11n26s` network when using the limit columns strategy. Left to right the graphs show: *Total gap* and the *Objective value*.

The first two graphs on figure 4.5 shows an example of such a run on the `28n45s` network.

Starting with Bhandari's algorithm

Finally we can try to get a further speed increase by first solving the subproblems with the fast heuristic called Bhandari's algorithm [10] described in section 4.3, and then when the algorithm can no longer improve, we finish the job with the MIP–solver.

As we can see on figure 4.5 the performance gained is not as dramatic as with the "limit columns" strategy. The 28n45s network takes about 3877 seconds to solve with the limit columns strategy and about 3819 seconds to solve when you also start solving with Bhandari's algorithm. That is an improvement of just 1,5%

There is a slightly bigger effect when the problem size increases. On the 43n71s network it takes about 33,9 hours to solve it with just the column limiting strategy and about 31,7 hours when starting with Bhandari. That is an improvement of about 6,4%. While this improvement has less effect on the running time than the limit columns strategy, it is still an improvement and will be used to generate the final results.

The reason for this more moderate effect is that this is an effort to speed up the sub problem, but the most time consuming part of the algorithm, in the large networks, seems to be solving the master problem. Further effort should probably go into that.

4.2 MIP algorithm

The Mixed integer programming, or MIP algorithm, is the main way of solving the sub problem. It is a direct implementation of the subproblem model from chapter 2 using COIN [1].

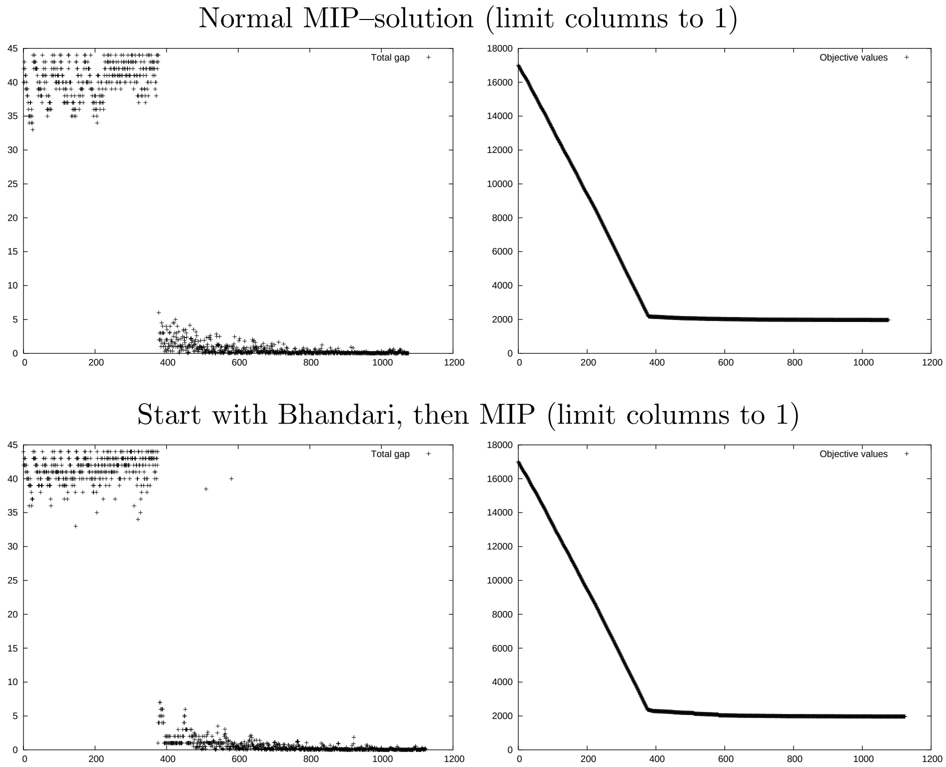

Figure 4.5: This is the `28n45s` network solved using only the MIP–solver, and using first the Bhandari heuristic and then the MIP–solver. The left ones are the *Total gaps* and the right ones are the *Objective values*

Once the main data structures have been created they can be re-used over and over, so to solve a sub problem using this technique, we need only:

- Change the values representing the source and destination nodes in the row bounds.
- Copy the new link prices and dual values to the objective function
- Let CPLEX solve it
- Restore the row-bounds
- Create a disjoint path pair

The values we need to change in the row bounds are based on which demand we are solving the sub problem for. All flow-constraints must be equal to 0 except the source and the destination node, so we change the row bounds to reflect which source and destination node we are working on.

The parameters of the objective function consists of the link prices as the x-columns, and the dual values of the link-rows in the master problem

Algorithm 4.2 Bhandari algorithm

```
path1 = BreathFirstSearch()
ChangeWeights(path1)
path2 = BreathFirstSearch()
RestoreWeights(path1)
RemoveOverlap(path1, path2)
```

as the z-columns. That means they must be changed every time the master problem has been run.

When the bounds are set we can let CPLEX solve it. This is by far the most time consuming part of the algorithm.

When the solver is done, we must restore the row bounds so they are ready for the next sub problem, and then we can extract the disjoint path pair and return it.

4.3 Bhandari algorithm

The Bhandari algorithm [10] is a heuristic that can find good solutions to the sub problems. Algorithm 4.2 shows the pseudo code for the Bhandari algorithm.

First we need to find the shortest path through the network using any shortest-path-algorithm. A Breath-first-search (BFS) is particularly well suited for the job since the algorithm needs to handle negative weights, so Dijkstra can not be used without significant modifications. Another reason to use BFS is that i tends to converge very fast as soon as the destination node is reached.

> For non negative graphs[1], improvement in efficiency [compared to Dijkstra] by a factor of as much as five has been observed for sparse graphs with 100 vertices or so [10, p.33].

When we have have found the first path we change the weights w of all the links on that path. All the links will be assigned a weight of infinite in the direction used by the first path, and the same links in the opposite direction will be assigned the weight $-w$ (see figure 4.6).

When the weights are transformed the shortest path must be found again. This time it is very important to use a shortest path algorithm that can handle negative weights, since we just added negative weights on all the links on the primary path.

There is no way the second path can share links in the same direction as the first path, since those weights was set to infinite. However there is

[1] The graphs are initially non negative, since no links in a real network can have a negative length (or whatever determines the weight). After the weight change some weights become negative so that further complicates the use of Dijkstra.

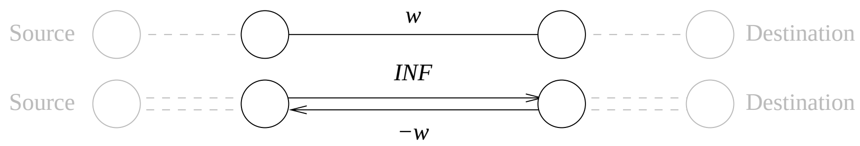

Figure 4.6: The weights of the links before and after the weight change.

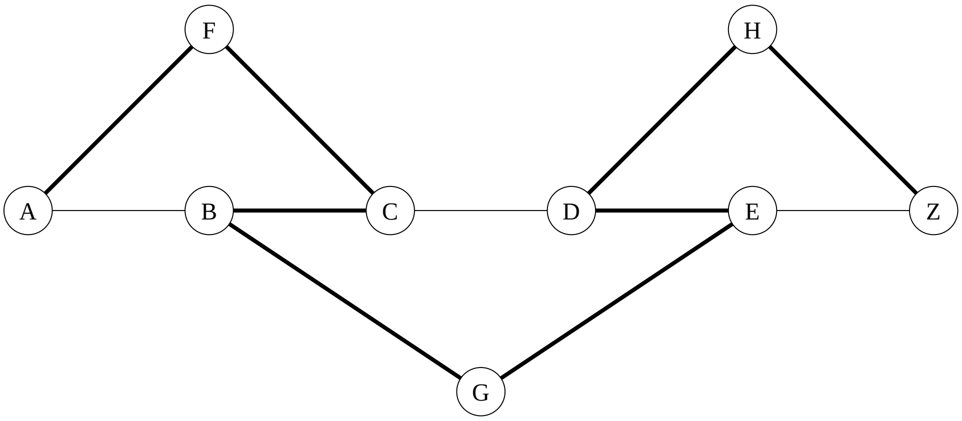

Figure 4.7: Two paths through a network. The first shortest path $path1 = ABCDEZ$, and the second shortest path (with new weights) $path2 = AFCBGEDHZ$.

a good chance that the second path will contain some of the same links in the opposite direction. They where after all made extra attractive by giving them a negative weight.

If the two paths do *not* share any links we are done, and the two paths can be returned as a disjoint path pair. If however there are overlaps, as in figure 4.7 the overlaps must be removed. The remaining links on the two paths can then be combined into two independent paths.

On figure 4.7 the overlaps are link BC and link DE. Once they are removed the disjoint path pair consists of the two paths ABGEZ and AFCDHZ. Unless we are working on a "throw away" copy of the network data, we should restore the weights to there original values at some point after we obtain the second shortest path.

4.4 LP–solver

The master problem is solved using the COIN interface to CPLEX. The algorithm is somewhat like the one used for the MIP-algorithm:

- Add new columns to the master problem.
- Let cplex solve it

- Save the objective value

The first time the master problem is solved we need to create the data structure for COIN to work on, just like in the sub problem. After that COIN maintains its own internal data structure.

Before we solve the master problem we must add one or more new columns. The first time the master problem is solved, all the new columns are dummy columns, but after that, the column generation algorithm will add the columns fond by the sub problem before trying to resolve.

When CPLEX is done the objective value and some other values are saved for fast and easy access.

4.4.1 Improvements

This algorithm, specifically the part where CPLEX is solving is the most time consuming part of the entire program for large networks. For that reason it is very tempting to look into ways of improving the running time of this part.

Since almost all the computation is in COIN we must change the model to improve the running-time of this part. One way to do this is by stabilization.

Stabilization

One way to improve the running time of the solver is to apply stabilization to the problem. Several people suggest stabilization as a possibility for reducing the running time and number of iterations of lp problems [8, 7].

The stabilized version of the master problem (P_{mp}) looks like this:

Minimize:

$$c^T x + \delta_-^T y_- + \delta_+^T y_+$$

Subject to:

$$\begin{aligned} \mathbf{A}x - y_- + y_+ &\leq b \\ y_- &\leq \epsilon_- \\ y_+ &\leq \epsilon_+ \end{aligned}$$

The idea behind stabilization is that the dual variables can fluctuate and assume extreme values in the early iterations. This can lead to a false representation of with rows are worth updating, and you thus spend a lot of time generating columns that are never used in the final optimal solution.

If we look at the dual problem (D_{mp}):

Maximize:

$$b^T \pi$$

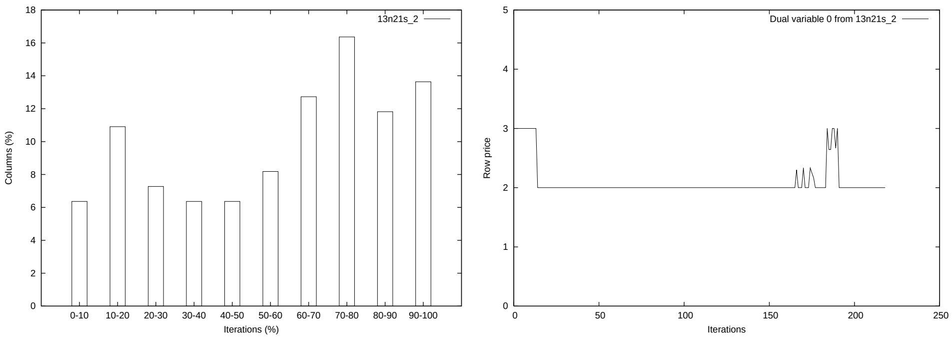

Figure 4.8: This is the distribution of the solution columns and a plot of a dual variable, both *without* stabilization.

Subject to:

$$\mathbf{A}^T \pi \leq c$$
$$-\delta_- \leq \pi \leq \delta_+$$

we see that the choice of the penalty vector δ sets bounds for the dual values π, and so this method provides us with a way of controlling the fluctuations.

However there *are* no, or very little, fluctuation of the dual variables in this program.

The first part of figure 4.8 shows where in the process the final solution columns where found. It turns out that even in the early iterations there are a relatively big percentage of columns found, and so the stabilization prerequisite of extreme values, and thus poor early solutions, is not met here. The lack of fluctuation and extreme values is shown clearly by the second part of figure 4.8 which shows the value of a dual variable in all the iterations.

If we view the final values of the dual variables as a point in n-dimensional space, we can find the distance from the current point in n-dimensional space to that point.

$$d = \sqrt{\sum (\pi - \pi^*)^2} \tag{4.1}$$

This is the Euklidian distance. The graphs on figure 4.9 shows the euklidian distance for the `13n21s_2`, and the `28n45s` network

The reason they do not start at iteration 0 is that only one column is added to the master problem at a time, so the first $n(n-1)/2$ iterations will have demands that are still covered by dummy columns and will thus give a false impression of the distance to the solution. After all the columns have been updated once (after $n(n-1)/2$ iterations) we see that the distance falls rapidly towards 0, meaning that the problem converges at once and not in

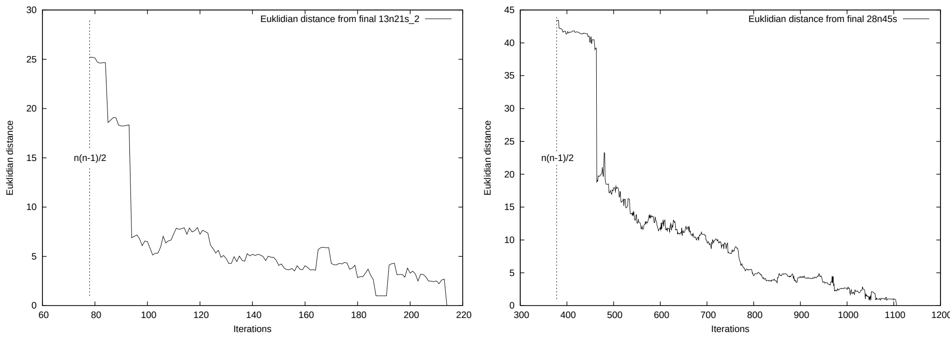

Figure 4.9: This is the euklidian distance from the current iteration to the final iteration of the `13n21s_2` network and the `28n45s` network.

the final few iterations. That confirms the conclusion from figure 4.8 that stabilization will not be of any great help in this problem.

It is not impossible that a small performance increase could be gained by implementing stabilization, but the bounds on the dual values δ would have to be very precise, and the overhead of calculating them would probably cancel out any performance increase.

Chapter 5

Implementation

This chapter contains a brief description of all the classes in the program and some implementation specific running time considerations.

For a thorough description of specific methods see the source code and comments in appendix A or the source code documentation.

5.1 Environment

The program is made in the following environment:

- CPU: UltraSPARC-III+ 1200 MHz
- OS: SunOS sunfire 5.9
- Language: c++
- Compiler: g++ (GCC) 3.3.3 (OpenPKG-2.0)
- COIN (CVS checkout: June 24th 2005)
- CPLEX Interactive Optimizer 9.0.0

5.2 Classes

On figure 5.1 you can see how the different classes that make up the program interconnect.

The following list briefly describes what each of the classes do. The classes are listed alphabetically.

basefilereader is the base class for all the different filereader objects. This enables the `network` object to have file format independent access to the data. It is also easy to add support for a new file format, just create a new class to parse the format and make sure it derives from `basefilereader`.

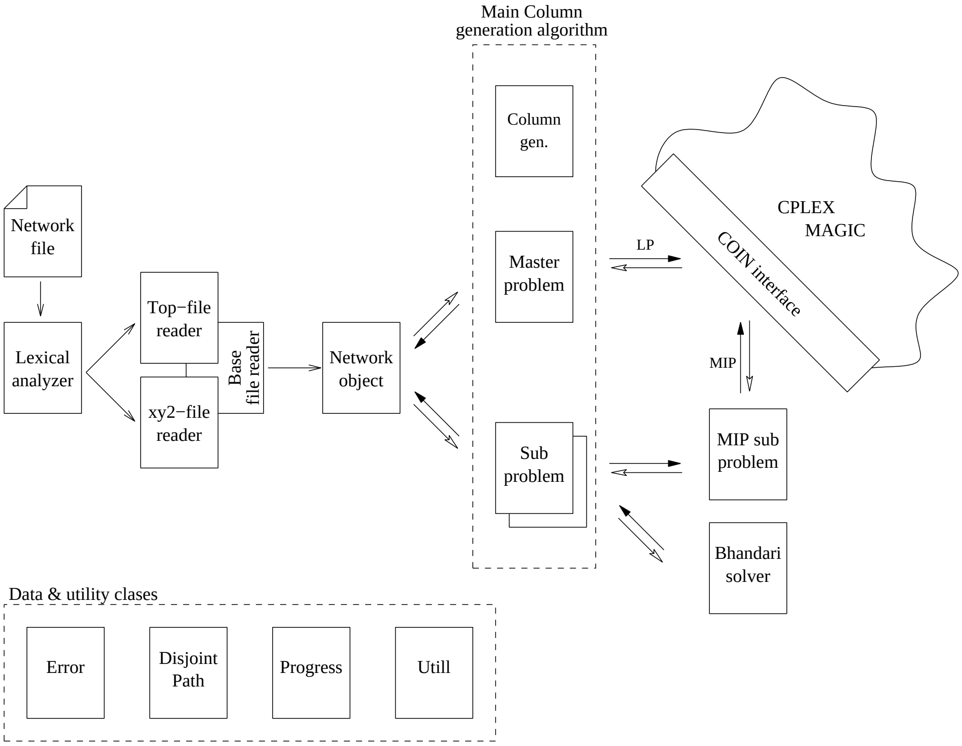

Figure 5.1: Relationship between the different classes.

columngen is the master algorithm. It solves the master problem and adds columns to the sub problem, using the other classes.

disjoint is a data type used for transferring solutions between the sub problems and master problem.

error contains several global error handling routines.

lex is the lexical analyzer. It reads a file and converts the characters to tokens which makes higher level parsing easier.

network uses a file format independent filereader to read the network data from a file. It then provides an api for querying and manipulating the data.

progress provides a progress bar and some other "user information" functions.

sbpp_sub_bhandari is the Bhandari solver. It is a heuristic that finds disjoint path pairs in a network quite fast.

sbpp_sub_mip is the MIP–solver implementation of the sub problem. It uses the COIN osi-api to access the underlying numerical solver (In this case CPLEX) to find disjoint path pairs.

sbppmasterproblem is the master problem. It also uses COIN osi-api to access the underlying numerical solver, but it is used to find the right combination of the available paths.

topfilereader is inherited from `basefilereader` and is used to parse top-network-files.

utill is a collection of various miscellaneous functions and data containers, such as a `set` used in the Bhandari solver.

5.3 Running time analysis

In this section we will look at how the running time is distributed among the various functions.

The distribution of running time changes with the size of the network. In the main column generation algorithm for the `43n71s` network, 99,98% of the running time is divided by these 3 functions:

MasterProblem –> solve	**71,90%**
OsiCpxSolverInterface –> resolve	71,79%
OsiCpxSolverInterface –> loadProblem	0,09%
OsiCpxSolverInterface –> initialSolve	0,01%
SubProblem –> solve	**27,96%**
OsiCpxSolverInterface –> branchAndBound	25,93%
OsiCpxSolverInterface –> setInteger	1,92%
OsiCpxSolverInterface –> loadProblem	0,09%
CreateMasterProblem	**0,12%**
MasterProblem –> CreateInitialCoinPackedMatrix	0,12%

As one can see on the table it is almost only OsiCpxSolverInterface[1] functions that takes up time.

This is desirable in terms of optimization. If it is only OsiCpxSolver-Interface functions that takes up time then there is no point in trying to optimize any of the other classes to increase performance

The exact distribution of time changes with the network size and density. Smaller networks have a relative higher percentage of running time in the sub problem and less in the master problem. It is unclear exactly what causes more time to be spent in the master problem in some networks and less in others. On figure 5.2 is a plot of the relative running time of the

[1] COIN's Open Solver Interface (osi) for CPLEX (cpx)

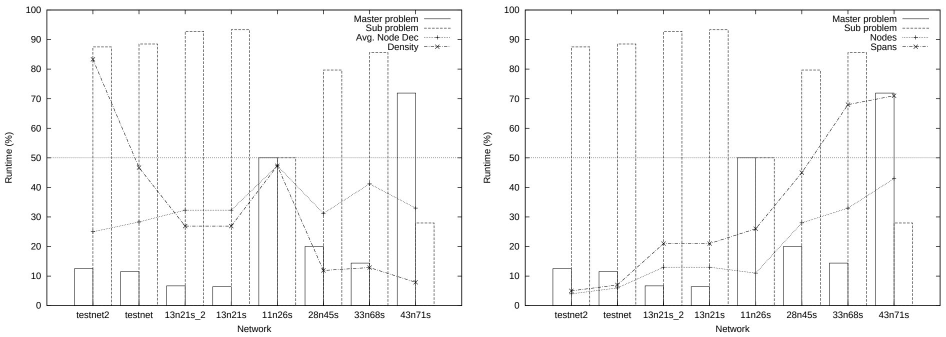

Figure 5.2: The bars show the relative runtime of the master- and sub problem for all the networks. The graphs on the left show the average node degree ($\times 10$) and density of the networks and the graphs on the right show the number of nodes and spans of the networks.

master- and sub problem for different networks shown as bars, and on top of those are plots of either the avg. node degree and density or the number of nodes and spans in the networks. It is not possible, based on this figure, to say if the master problem will be more and more dominant as the network size increases, but the running time of the master problem do play a more significant role in the large networks compared to the small.

The network `11n26s` seem to differ from the others in how the running time is distributed. One explanation of this could be that the `11n26s` network is more dense than the other networks, that leads to more combinations of paths and therefore more load on the master problem. Density alone is not the answer though because the two largest networks `33n68s` and `43n71s` both have very low density, but very different distribution of the master- and sub problem running time.

One thing that *is* certain is that even though the running time is distributed differently in all the networks, the OsiCpxSolverInterface takes up more than 95% of the total running time.

5.4 Data structures

The data structures in the program play a vital part in the performance and extendibility of the program. Here is a short description of each of the important data structures:

Disjoint path is a data type used to transfer data from the subproblem to the master problem. The operations that are really important are to be able to add new spans to a disjoint path pair very fast when a sub problem finishes, and to run through them again when they are added to the master problem.

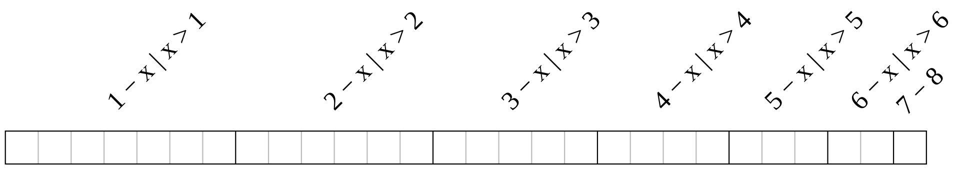

Figure 5.3: The structure of the demands.

Both an array implementation and a linked list implementation can accomplish this but in the linked list implementation we must create the "links" in the list on the fly, and that means a lot of memory operations witch are notoriously slow, so the array implementation was chosen.

The only problem with an array implementation is that enough memory must be reserved to handle the largest possible disjoint path pair or we will still have to do expensive memory operations. That is not a problem in this case, since we know that no path can be longer than the total number of links. This of course wastes some memory, but given the fact that there are less than 100 links and that we reuse the same disjoint path object over and over so we only need to create one, this is a good, clean, fast solution.

Network structures can be realized in several ways. In this case we use a double representation to be able to quickly satisfy all requests.

There is an array of node structures each with a linked list of spans used to describe how the network is interconnected and to be able to return all the links connected to a particular node in O(1) time. Then there is an array of span structures so we, given a span, can determine what two nodes are connected to it in O(1) time. Finally there is a matrix that acts as a node-to-span-map in which you, given two nodes, can find the span that connects them in O(1) time.

Since all requests are handled in constant time the network object provides a good and fast data structure for all the algorithms that use it.

Demands are necessary so we know how much data that needs to be transported on the network.

There are one demand for each node pair so we could just use the top half of a matrix, but that would waste the bottom half and since this is not a performance critical structure there is no point in accepting a waste. Instead the data is arranged in a single array, see figure 5.3, where the first part is all the demands from node 1 to all the other nodes, the next part is all the demands from node 2 to all the other nodes (except 1) etc.

In addition to those there are several data structures used in the COIN library such as CoinPackedMatrix (CMP) and CoinPackedVector (CPV), information on them can be found on the COIN–OR website [1].

Chapter 6

Results and Discussion

In this chapter the different networks will be described, and we will present all the results and discuss their meaning and significance.

We will also look at the alternate path protection method called Reverse Single Backup Path Protection (rsbpp) described in chapter 1 and look at the pros and cons of that method versus sbpp.

6.1 The networks

The seven[1] different networks presented in table 6.1 are what all the graphs and other results are based on. The two smallest networks `testnet` and `testnet2` are to small to be realistic, but make good testing and debugging networks. The rest are realistic networks from different sources [13], [14].

Some of the networks might have alternate names in other articles. E.g. the `28n45s` network is called `france` in [13], but since this paper is an exploration of the sbpp protection model and not an actual solution to say a specific transport problem, I prefer the more descriptive numerical name over the geographical one.

The seven networks in table 6.1 make up a diverse test set since they vary in size, density and span costs.

The density of the networks describe how many spans the network has, compared to the maximum number of possible spans a network with that amount of nodes, could have. The density of a graph $G = (N, S)$ with N nodes and S spans is

$$density = \frac{\text{number of spans}}{\text{max. number of posible spans}} = \frac{S}{N(N-1)/2} = \frac{2S}{N(N-1)}$$

[1]There are actually eight networks listed in table 6.1, but `13n21s` and `13n21s_2` are the same network with different span costs.

Name	Number of Nodes	Number of Spans	Density	Average Node Degree	Span costs
43n71s	43	71	7,9%	3,30	1
33n68s	33	68	12,9%	4,12	1
28n45s	28	45	11,9%	3,12	1
13n21s	13	21	26,9%	3,23	$[1; 8]$
13n21s_2	13	21	26,9%	3,23	1
11n26s	11	26	47,3%	4,73	1
testnet	6	7	46,7%	2,83	$[0.9; 1]$
testnet2	4	5	83,3%	2,50	1

Table 6.1: The networks.

The average node degree of the network is the average number of spans that are connected to each node. It is calculated with the following simple formula:

$$AvgNodeDeg = \frac{2 \cdot \text{Number of Spans}}{\text{Numer of Nodes}} = \frac{2S}{N}$$

because each span connects to two nodes the number of spans must be multiplied by two.

The span cost is the price of each of the individual spans. A single '1' means that all the prices are one on all the spans, and an interval $[1; 8]$ means that the prices are between one and eight (both inclusive) on the spans.

6.2 Solutions

The solutions in table 6.2 are generated with the *limit columns to 1* strategy and the *start with bhandari* strategy since this is the fastest of the algorithms described in chapter 4.

The demands are all one. That means that between every node pair there is a demand and all the demands are equal, in this case one. The number is scalable so if all the demands where 2, the objective value would just be twice as big.

Iterations and Number of columns are the number of iterations it took to solve the problem and the total number of columns in the final iteration, including the y columns and the initial dummy columns.

The solution time is the number of seconds it took to reach the solution, and the Objective value is the final result.

6.3 Quality

On table 6.3 the networks and there objective value are listed with the lower bound and a gap. The lower bound is the absolute lowest cost of path

Name	Demands	Iterations	Number of columns	Solution time[1]	Objective-value
43n71s	all one	3699	4673	96535	5220,33
33n68s	all one	2560	3156	193486	2352,66
28n45s	all one	1124	1547	3819	1967,8
13n21s	all one	329	428	81	1195,67
13n21s_2	all one	218	317	43	260
11n26s	all one	540	621	129	102,267
testnet	all one	32	54	<1	45,1
testnet2	all one	14	25	<1	13

[1] This is only useful in comparison with the other networks, and can not be compared to results calculated on other machines.

Table 6.2: The solutions.

Name	Lower bound	Objective-value	Gap
43n71s	5077	5220,33	2,8%
33n68s	2299,4	2352,66	2,3%
28n45s	1914,2	1967,8	2,8%
13n21s	1169	1195,67	2,3%
13n21s_2	248	260	4,8%
11n26s	97,6	102,267	4,8%
testnet	44	45,1	2,5%
testnet2	13	13	0%

Table 6.3: The quality of the solutions.

protection, if we are allowed to reroute all the connections in the network when a link failure occurs, not just the paths directly affected, but all paths in the network. This is referred to as global rerouting in chapter 1. The values are taken from [13].

This is of course not practical in a real world situation, but it provides a useful lower bound to test how well this algorithm works. As is evident from the table the algorithm is *very* close to the lower bound, between 2,3% and 4,8% if we discard the two small test networks.

If we look at the quality of the solutions in terms of capacity requirements sbpp performs exceedingly well. With an average gap of 2,8% from the absolute lower bound the results are very close to perfect.

6.3.1 Solution time

The quality in terms of solution time of the sbpp algorithm is more debatable. It range from under one second on the very small networks to about 50 hours on the largest. Obviously 50 hours is to much time to be practical if the environment is very dynamic, but in large backbone networks which seldom change topography it might not be a problem.

6.4 Implementability

Sbpp can be a bit hard to implement in a live network in its current form. First of all it requires complete knowledge of the demands on the network and the link prices on all links.

On the other hand if the routing problem was solved on a computer and then the nodes in the network where just told what to do, so the nodes themselves did not have to be intelligent, it could be realized with the technology we have today.

It would be a problem to add or remove nodes from the network as the entire problem would have to be solved again, at least to get an optimal solution, but replacement of nodes would be easy since the new node would just need the path configuration of the old one to fit into the plan.

6.5 Objective values

On first glance it might seem odd that the objective values can assume non integer values when the demands and prices are all integer. It happens because the flow that supply a particular demand is split into several paths.

To understand why it might be preferable to split a demand across two or more paths it is important to remember that the master problem is an lp problem and can use any percentage of any span.

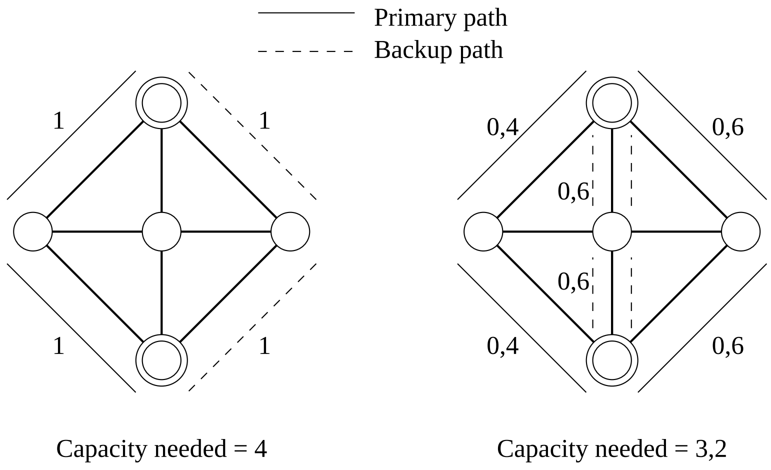

Figure 6.1: Split path.

An example of a split path can be seen on figure 6.1. On the left part of figure 6.1 is a normal disjoint path pair. All the data travels along the primary path and therefore we need to reserve capacity for it. If one of the spans on the primary path fails all the data must use the backup path, and therefore we must reserve the same amount of capacity there.

On the right part of the figure is the same network and the demand between the source and destination nodes is still one. This time the flow is split into two disjoint path pairs. The one running along the left side of the network carries 40% of the data and the one running along the right side carries 60% of the data. Both backup paths share the spans from the source to the destination.

The combined capacity of the spans on the primary path is still two, but it is sufficient to reserve 0,6 on each of the backup spans as that is the maximum amount of capacity we will need no matter which of the primary paths fail.

It would have been even better to let each of the primary paths carry 50% of the traffic instead of 40-60. That would reduce the needed capacity to 3.

6.6 Integer solution

The master problem of the sbpp model is an lp problem and that means that you can use half the capacity on a link and that way only pay half the price. In some real world situations that is probably not possible, as you might be in a situation where you pay for a line whether you use the maximum capacity or only a small fraction of the capacity.

One way to model this limitation is to add the constraint that either

Name	Sbpp	Integer	Gap
43n71s	5220.33	5221	0,00%
33n68s	2352,66	2354	0,05%
28n45s	1967.8	1968	0,01%
13n21s	1195.67	1196	0,02%
13n21s_2	260	261	0,38%
11n26s	102.267	104	1,69%
testnet	45.1	45,1	0,00%
testnet2	13	13	0,00%
Average gap			0,27%

Table 6.4: The original sbpp solution compared to the solution of the final master problem solved as a mixed integer problem.

all of the decision variables in the master problem, or just the y's, *must* be integers. If we say that only the y's have to be integers that means that we can still split up the traffic supplying a single demand across multiple paths, but once you use a link you pay the full price for it.

If we chose to make *all* the decision variables of the master problem integer, it means that it would be impossible to only use, say half the capacity on a disjoint path pair, but that would never be useful since we pay full price for all the links on that path we might as well use all the capacity on them, so the two methods well give the same result. Since they both give the same result, we use the one where only the y's are integers. It gives the simplest problem, and it is the most intuitively correct way to model the situation.

In order to get an optimal integer solution we would have to convert the master problem to a MIP problem and then our column generation algorithm can no longer be used.

To investigate whether or not a complete integer "branch and bound" or "branch and price" solution is necessary, we try to solve the final master problem again, but this time we add the constraint that all the y-variables must be integers. If this integer value is very close to the optimal lp value we can argue that it is not necessary to recalculate the entire problem using branch and bound/price. The results of this test can be seen in table 6.4.

As we can see in the table, the average gap is 0,27%. In fact, in many of the networks the integer solution is just the lp solution rounded up and is thus proven to be the optimal integer solution to the problem.

This confirms that the results we get are optimal, or very close to optimal even in situations where the lp-simplification is not applicable.

6.6.1 The integer results

There are a couple of things worth noting in table 6.4. The first is the fact that the integer solution of testnet is not integer. This is because the link

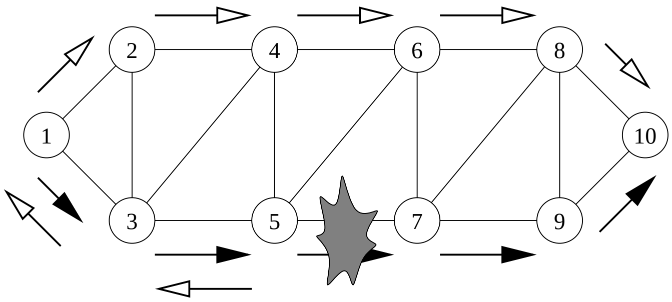

Figure 6.2: Reverse Single Backup Path Protection (rsbpp)

prices of testnet are not all integer. The spans them selves (the y-columns) are integer values, but when multiplied by the non integer costs according to (2.1) we get a non integer sum.

Also worth noting is the fact that the integer solution of the `11n26s` network is 1,69% higher than the lp relaxation. That is pretty high compared to the other networks. One explanation of this is that the network is much more dense compared to the other networks and therefore there are more opportunities to split the data flow between paths that can not be utilized in an integer solution. See figure 6.1.

Another explanation is that this is a relatively small network so even a relatively small difference yields a high percentage. The `33n68s` network is also more dense than most of the others, but because it is so large, a difference of about 2,33 only yields a gap of 0,05%.

6.7 Reverse Single Backup Path Protection

In a situation where the recovery time is more important than the actual transmission time, e.g. live television broadcasts we might be concerned with the time it takes to discover that a link failure has occurred and then retransmit all the data since the failure.

In such a situation it might be more desirable to have the node just before the link failure transmit the data back to the source node and then via the backup path to the destination, since that node is the first to discover that a link failure has occurred [3], [2].

This is called Reverse Single Backup Path Protection (rsbpp) and is illustrated on figure 6.2.

6.7.1 New formulation

The formulation of the Reverse Single Backup Path Protection (rsbpp) problem is almost identical to the sbpp problem described throughout this paper.

We will take a look at the master and sub problems to see exactly where they differ.

6.7.2 Rsbpp master problem

We need to reserve enough capacity for the data to be sent out and back again on all the spans in the primary path except the last one. That means we need to pay twice for all the spans in the primary path, except the last one before the destination. The last one can never transport data forward and back again since no span ahead of it can break.

The backup path is unchanged see figure 6.2. The following is a formulation of the rsbpp problem:

Minimize:

$$\sum_{ij} C_{ij} \cdot y_{ij} \tag{6.1}$$

Subject to:

$$\sum_{p} x_p^{kl} \geq D^{kl} \quad \forall kl \tag{6.2}$$

$$\sum_{kl} \sum_{p} a_{p,ij}^{kl} \cdot (d_{ij}^{kl} + 1) \cdot x_p^{kl} + \sum_{kl} \sum_{p} a_{p,qr}^{kl} \cdot b_{p,ij}^{kl} \cdot x_p^{kl} \leq y_{ij} \quad \forall (il, qr) | ij \neq qr \tag{6.3}$$

$$x_p^{kl}, y_{ij} \in \mathbf{R}_+ \tag{6.4}$$

The objective function and the demand are exactly the same as in the sbpp formulation. We still only pay for the spans we use and all demands must still be met.

The only difference is in constraint (6.3) where there is now an additional $(d_{ij}^{kl} + 1)$. The indicator variable $d_{ij}^{kl} = 1$ if the span ij is *not* connected to the destination node l and 0 if it is.

6.7.3 Rsbpp sub problem

None of the constraints of the sub problem needs to change from the sbpp- to the rsbpp formulation. We still need two disjoint paths through the network. The only necessary change here is in the objective function to get the correct price of the paths.

Minimize:

$$(d_{ij}^{kl}+1)\cdot\sum_{(ij)} c_{(ij)}\cdot x_{(ij)} + \sum_{ij}\sum_{qr}\beta_{ji,qr}\cdot z_{ji,qr} \tag{6.5}$$

Subject to:

$$\sum_{(ij)} x_{(ij)} - \sum_{(ji)} x_{(ji)} = \begin{cases} 1 & \text{for } i=k \\ -1 & \text{for } i=l \\ 0 & \end{cases} \quad \forall i \tag{6.6}$$

$$\sum_{(ij)} y_{(ij)} - \sum_{(ji)} y_{(ji)} = \begin{cases} 1 & \text{for } i=k \\ -1 & \text{for } i=l \\ 0 & \end{cases} \quad \forall i \tag{6.7}$$

$$x_{(ij)} + x_{(ji)} + y_{(ij)} + y_{(ji)} \leq 1 \quad \forall\{ij\} \tag{6.8}$$

$$x_{(qr)} + x_{(rq)} + y_{(ij)} + y_{(ji)} - 1 \leq z_{ij,qr} \quad \forall(ij,qr)|ij\neq qr \tag{6.9}$$

We added $(d_{ij}^{kl}+1)$ in front of the primary path sum where the indicator variable $d_{ij}^{kl}=1$ if the span ij is *not* connected to the destination node l and 0 if it is (the same as in the master problem). That way we pay twice for all the links on the primary path except the last link, and the backup path is untouched.

In the formulations of the rsbpp master and sub problem we use d_{ij}^{kl} as an indicator for whether the span ij ís connected to the destination node l or not. One implementational advantage of this approach is that d_{ij}^{kl} only need to be created once and then it can be used for lookups when needed. This is more efficient than adding a new sum for the part of the primary path that needs to reserve extra capacity.

6.7.4 Sbpp vs. rsbpp

Table 6.5 lists the rsbpp results along side the sbpp results and the lower bound. The “sbpp gap” is the relative distance from the sbpp solution to the rsbpp solution and the “total gap” is the relative distance from the lower bound to the rsbpp solution.

The results on table 6.5 speak for them self. The method is far more expensive than sbpp in terms of capacity, however the rsbpp method *is* more robust.

In sbpp the problem is that in the time it takes the node right in front of the link failure to signal the source node to use the backup path, data will already have been sent via the now broken path. The source node must now re-send the data via the backup path and that causes a “hole” in the data flow because re-sending usually means going back to the source of the data and recreating it. Sometimes, like with live television broadcasts, it is not possible to just recreate lost data. In that case you just have to accept that a few seconds of transmission is lost.

Name	Lower bound	Sbpp	Rsbpp	Sbpp gap	Total gap
43n71s	5077	5220,33	7842,83	50%	54%
33n68s	2299,4	2352,66	3570,12	52%	55%
28n45s	1914,2	1967,8	2877	46%	50%
13n21s	1169	1195,67	1524	27%	30%
13n21s_2	248	260	340	31%	37%
11n26s	96,7	102,267	133,267	30%	38%
testnet	44	45.1	53.3	18%	21%
testnet2	13	13	14	8%	8%

Table 6.5: The original sbpp solution and the lower bound compared to the rsbpp solution

In rsbpp when the node right in front of the link failure discovers the problem it just sends the data back to the source node and then via the backup path. There will still be a small time hole in the data flow but no data needs to be re-send.

Chapter 7

Future research

In this chapter we will look at some possibilities for future research related to single backup path protection.

7.1 Stop–release in sbpp

The question of stop–release is the question of whether or not it is useful to release the capacity of the primary path when a span on the primary path breaks. At first glance it seams likely that another path can use some of the released capacity thereby making an overall cheaper solution.

Formally the sbpp master problem using stop–release can look like this:

Minimize:

$$\sum_{ij} C_{ij} \cdot y_{ij} \tag{7.1}$$

Subject to:

$$\sum_{p} x_p^{kl} \geq D^{kl} \quad \forall kl \tag{7.2}$$

$$\sum_{kl}\sum_{p}(1 - a_{p,qr}^{kl}) \cdot a_{p,ij}^{kl} \cdot x_p^{kl} + \sum_{kl}\sum_{p} a_{p,qr}^{kl} \cdot b_{p,ij}^{kl} \cdot x_p^{kl} \leq y_{ij} \quad \forall (il, qr)|ij \neq qr \tag{7.3}$$

$$x_p^{kl}, y_{ij} \in \mathbf{R}_+ \tag{7.4}$$

The only difference from the original master problem in chapter 2 is the added $(1 - a_{p,qr}^{kl})$ in constraint (7.3). The meaning of $a_{p,qr}^{kl}$ is explained in chapter 2, but informally this change means that if a span on the primary path breaks, the primary path no longer use the capacity on any of it's spans.

In order to see whether this change will have any useful effect, we can go though the possible benefits one by one. Note that the model stops as soon

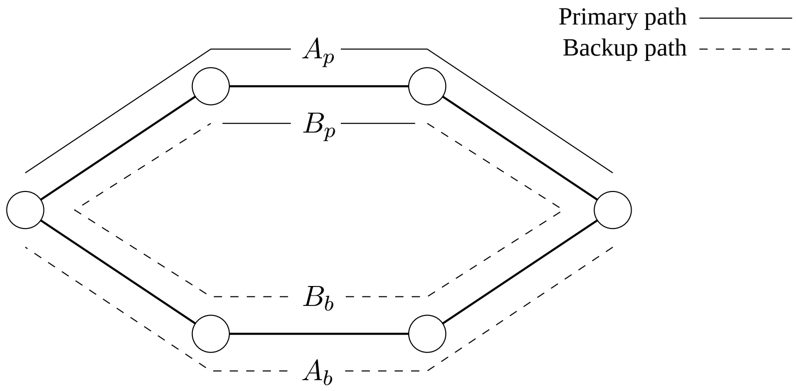

Figure 7.1: A scenario in which the use of stop–release would improve the solution. The full lines are primary paths and the dotted lines are backup paths.

as a span breaks, so any benefit will have to be in the amount of capacity we need to reserve for a particular link.

Let us assume that the disjoint path pair on which the link failure occurs is called A where A_p is the primary path and A_b is the backup path. There are four types of paths who can possibly take advantage of the released capacity: The primary path itself A_p, the backup path A_b, another primary path supplying another demand B_p or another backup path supplying another demand B_b.

The first candidate A_p can clearly not benefit from any released capacity since the capacity is only released when A_p breaks. The backup path A_b can not benefit either since it can not use any spans used by the primary path (they are disjoint).

That leaves other primary and backup paths. Another primary path B_p that share some unbroken links with A_p can not benefit from any released capacity since enough capacity for both paths have to be reserved on all shared links. After all, both paths should be running simultaneously most of the time. What about another backup path B_b that share some unbroken links with A_p? We also in this case need to reserve enough capacity on all the links to accommodate the worst case scenario. A link failure could occur anywhere, so B_b can not rely on capacity from a broken A_p becoming available, because it won't if the failure occurs on a link used by B_p but not A_p. The only possible way this released capacity could be used is if there *are* no such links, in other words B_p must be "covered " by A_p (see figure 7.1).

If the situation is similar to the one on figure 7.1 then the backup path B_b need not reserve capacity on the links covered by A_p because the only way B_b will be used is if B_p is broken, but since B_p is covered by A_p that implies that A_p is also broken and thus no longer need the capacity on the links it covers.

In the situation on figure 7.1 the price of protection would be 2 less using stop–release then without, and it could be an object of further research to test how often this occurs in practice, and if the improved solutions are worth the added complexity.

7.2 Bhandari prices

One of the things that help speed up the column generation algorithm is the fact that we initially use a heuristic called Bhandari to generate some disjoint path pairs.

We are not able to rely completely on this heuristic though, because the disjoint path pairs are generated based on some inaccurate prices. The Bhandari algorithm finds the disjoint path pair with the least combined price, and then returns them as a disjoint path pair. When the pair is created the Bhandari algorithm just chooses one of the paths to be the primary and the other to be the backup path, however the price of the links in the network depends on that choice.

When one of the paths are chosen to be the primary path, the link prices for the backup path changes due to the fact that we only have to pay for the links on backup paths if they might be used together with some other paths on that link, as explained in chapter 2 and 3.

When the prices change we are no longer sure that we have the cheapest disjoint path pair. The (now chosen) backup path might be cheaper or indeed free if we chose another path. Due to this problem the Bhandari algorithm can only be used as a starting point for the column generation algorithm. After a number of iterations, Bhandari can no longer improve the solution, and we have to use the MIP based method of solving the sub problem.

It is not likely that we will ever find an improvement to Bhandari that would make it capable of solving the sub problem completely[1], but with a few modifications it is likely that we can get much closer to the optimal solution before we have to use the MIP-solver.

Further research could uncover whether recalculating the backup path with the new prices after choosing the primary path would bring us closer to the optimal solution.

7.3 Optimize master problem

As is suggested from the running time analysis in chapter 5 the time spent solving the master problem plays a more and more significant role as the

[1]Not in polynomial time anyway. In chapter 2 we show that the sub problem is NP-Hard, so such a solution would imply P=NP

network size increases.

Exactly what parameters are responsible for more or less time spent in the master problem as oppose to the sub problem are unclear, and anyone wanting to optimize the master problem further would want to clarify that first, in order to see whether or not the master problem will indeed become more and more dominant as the network size increase or some factor other than size are responsible.

In chapter 4 we conclude that stabilization of the master problem will not have any great effect on the running time of the column generation algorithm, but there may be other useful methods.

Another topic that might deserve further research is the pruning strategy's discussed also in chapter 4. If a way of reducing the size of the master problem, without removing columns that are needed to verify the solution could be found, it would surely mean a faster algorithm.

Chapter 8

Conclusion

In this theses a model for single backup path protection was presented. The model was converted to a form that made it strait forward to implement, and the implementation and result where presented along with some key algorithms used in the implementation.

The sbpp model consisting of a master problem and a sub problem are closely modeling a real world network and the assumptions of infinite bandwidth and infinitely dividable links did not turn out to be a problem in the solution.

The COIN library is a well suited environment for implementing linear programming- and integer programming models. After the problem was converted to matrix form it was relatively strait forward to implement it using the features of COIN, especially when combined with the column generation algorithm.

Many improvements where made to the original algorithms and the combined result of all the optimizations was a running time 60 times faster than that of the original program. We saw that the running time now depends almost exclusively on the COIN library and the underlying solver, in this case CPLEX, so any future improvement would be on the model itself.

Stabilization was evaluated as a way of improving the model, but was found not to be helpful. It is possible that other lp-optimization methods would yield a faster result, but one vital part in improving the running time would be to determine exactly where the majority of the time is spent. It is still unclear exactly what network properties are responsible for whether the master problem or the sub problem are the most time consuming.

The most impressing part of this model is the quality of the results it produces. The capacity requirements are only a few percent from the absolute lower bound of path protection. Even if we impose the restriction that we must pay for the entire link no matter how small a part we use the results are in many cases just rounded up from the lp-solution to the nearest integer solution, thereby proving that the results are close to what

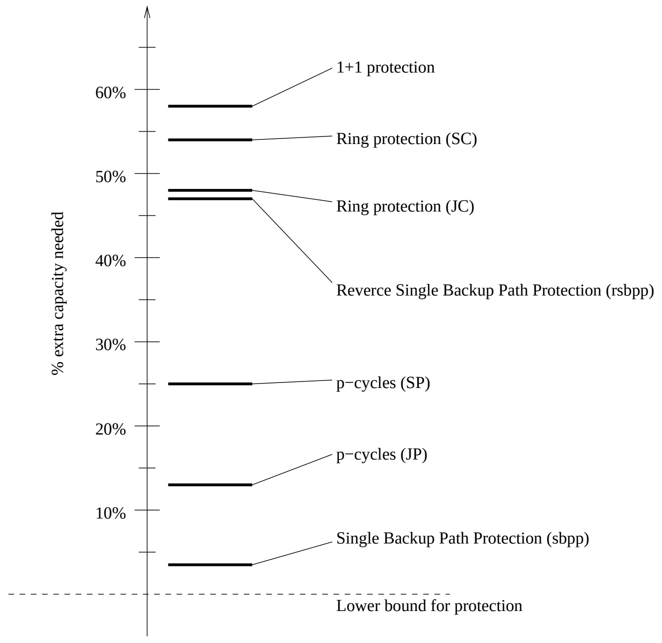

Figure 8.1: Various known protection methods, and their average capacity requirements compared to the lower bound.

we would expect to get if this protection scheme was implemented in a real world network.

The closely related reverse single backup path protection model was presented and the capacity requirements where about comparable to ring protection. In this authors opinion the dramatic increase in capacity requirement compared to sbpp is not a price worth paying for the added robustness.

Overall single backup path protection is superior in terms of capacity requirements but not as fast as 1+1 protection when it comes to recovery from link failure.

If you want to protect the integrity of your network, but doesn't need instantaneous recovery or don't want to pay the capacity-price that 1+1 protection demands, Single backup path protection is a very attractive alternative as can be seen on figure 8.1.

Appendix A

Selected source code

This appendix contains part of the source code. All the source code can be found on the CD.

A.1 columngen.cc

```
#include <stdio.h>
#include "config.h"
#include "error.h"
#include "disjoint.h"
#include "network.h"
#include "sbppmasterproblem.h"
#include "sbpp_sub_bhandari.h"
#include "sbpp_sub_mip.h"
#include "progress.h"

#define STATE_MIP    0
#define STATE_BHA    1

int main()
{
    debugMsg("Debug messages are ON.\n");
    debugMsg("Compile whitout DEBUGMSG defined to turn them off\n\n");

    // Create a network object based on a filename
    Network net("netdata/13n21s_2.top");

    // Specify the demand
    int nNodes = net.getNumNodes();
    int nlinks = net.getNumSpans();
    int nDemandSet = (nNodes*(nNodes-1)/2);
    int demand[nDemandSet];
    for(int n=0; n<nDemandSet; n++)
        demand[n] = 1;

    // Create the master- and subproblem objects we have to work on
    SBPPMasterProblem mp(&net, demand);
```

```
#ifdef WITH_BHANDARI
    SBPPSubProblem *sp = new SBPP_bhandari(&net, &mp);
    int state = STATE_BHA;
    double objval = 0;
    int samecount = 0;
#else
    SBPPSubProblem *sp = new SBPP_MIP(&net, &mp);
#endif

    // alocate mem for a disjoint path
    int *p1 = (int*)malloc(nlinks*sizeof(int));
    int *p2 = (int*)malloc(nlinks*sizeof(int));
    Dpath dpath(p1, p2, 0, 0);

    // open a logfile to receve data
    FILE *logfile = fopen("data.log", "w");
    fprintf(logfile, "# Progress logfile v1.4 (5 columns)\n");

    // enter the main column generation loop
    bool fNewPaths=1, fJumpIn=0, fTerminate=0;
    int iterations = 0;
    double gap=0;
    int klsave=0, ksave=0, lsave=0;
    int k=0, l=0, kl=0;
    int columnsAdded = 0;

    while(fNewPaths)
    {
        // Reset fNewPaths... we didn't find any yet
        fNewPaths = 0;

        // Solve the master problem
        mp.solve();

#ifdef WITH_BHANDARI
        if(state==STATE_BHA)
        {
            if(isEqual(mp.getObjValue(), objval))
                samecount++;
            else
            {
                samecount=0;
                objval = mp.getObjValue();
            }

            if(samecount>=3)
            {
                delete sp;
                sp = new SBPP_MIP(&net, &mp);
                state = STATE_MIP;
            }
        }
#endif

```

```
        // update the network with new prices
        for(int i=0; i<nlinks; i++)
            net.setSpanWeight(i, mp.getDualSpanCost(i));

        // Solve 'COL_PR_IT' of the subproblems the subproblems
        printf("Solving %i of the %i subproblems\n", COL_PR_IT, nDemandSet);
        for(gap=0, columnsAdded=0, fTerminate=0; columnsAdded<COL_PR_IT;
            fTerminate=1)
        {
            for(k=0, kl=0; k<nNodes-1 && columnsAdded<COL_PR_IT; k++)
                for(l=k+1; l<nNodes && columnsAdded<COL_PR_IT; l++, kl++)
                {
                    if(fJumpIn)
                    {
                        k  = ksave;
                        l  = lsave;
                        kl = klsave;
                        fJumpIn = 0;
                        continue;
                    }

                    // Solve the subproblem for the kl node-pair
                    sp->solve(&dpath, k, l);

                    // if this dpath can reduce the total cost of the
                    // masterproblem
                    if(mp.getDualDemandCost(kl)-dpath.getPrice() > 1e-4)
                    {
                        // Sum all the reduced costs to find a "gap"
                        gap += mp.getDualDemandCost(kl)-dpath.getPrice();

                        // Add the dpath to the master problem
                        mp.addPath(&dpath, kl, l, sp);

                        fNewPaths = 1;
                        fTerminate = 0;
                        columnsAdded++;
                    }

                    // Indicate activity to the user
                    blingbling();
                }

            if(fTerminate)
                break;

            // Save the k, l, and kl values so we can jump back in
            if(columnsAdded>=COL_PR_IT)
            {
                ksave   = k-1;
                lsave   = l-1;
                klsave  = kl-1;
                fJumpIn = 1;
            }
```

```
        }

        printf("gap = %g\n", gap);

        // Write to the logfile and update iteration counter
        fprintf(logfile, "%i %i %g %g %g\n", iterations,
            mp.getNumCols(), mp.getDualDemandCost(0), gap,
            mp.getObjValue());
        iterations++;
    }

    // print the solution
    printf("Final solution: %g\n", mp.getObjValue());

#ifdef INTEGER_SOLUTION
    printf("Final integer solution: %g\n",     mp.solveAsMIP());
#endif

    // close the logfile
    fclose(logfile);

    // Clean up
    free(p1);
    free(p2);
    delete sp;
    return 0;
}

```

A.2 sbppmasterproblem.cc

```
#include <assert.h>
#include "config.h"
#include "error.h"
#include "sbpp_sub_mip.h"
#include "progress.h"

SBPPMasterProblem::SBPPMasterProblem(Network *n, int *d)
{
#ifndef USE_CLP_SOLVER
    cplex    = new OsiCpxSolverInterface;
#else
    cplex    = new OsiClpSolverInterface;
#endif
    infinity = cplex->getInfinity();
    net   = n;
    numberOfColumns = 0;
    isFirstSolve = 1;
    demand = d;
    numColsCach = numNewColsCach = numRowsCach = 0;

```

```
    nnodes = net->getNumNodes();
    nlinks = net->getNumSpans();

    // Create the initial feaseble solution
    cpm = createInitialCoinPackedMatrix();

    // cach the demand row prices so they are
    // availeble afte we modify the problem
    rowPriceCach = new double[demandRows];
}

SBPPMasterProblem::~SBPPMasterProblem()
{
    delete cplex;
    delete cpm;
    delete[] rowPriceCach;
}


CoinPackedMatrix* SBPPMasterProblem::createInitialCoinPackedMatrix()
{
    debugMsg("[MasterProb] nodes: %i, links: %i\n", nnodes, nlinks);
    demandRows = (nnodes*(nnodes-1))/2;

    CoinPackedMatrix *matrix = new CoinPackedMatrix;

    // Create the 'y' columns

    //first create the values
    double yvalues[nlinks-1];
    for(int i=0; i<nlinks-1; i++)
        yvalues[i] = -1;

    // Then create the indexes
    int yindex[nlinks-1];
    for(int ij=0; ij<nlinks; ij++)
    {
        for(int qr=0; qr<nlinks-1; qr++)
            yindex[qr] = demandRows + qr + ij*(nlinks-1);

        // add the column to the matrix
        CoinPackedVector cpv(nlinks-1, yindex, yvalues);
        matrix->appendCol(cpv);
        numberOfColumns++;
    }

    // now we need enough paths (columns) to make a feaseble solution, but
    // we make them so expensive they won't be used in the final solution

    // create the values and indexes.
    int columnheight = nlinks*(nlinks-1)+1;
    double xvalues[columnheight];
    int    xindex [columnheight];

```

```
    for(int i=0; i<columnheight; i++)
    {
#ifdef RSBPP
        xvalues[i] = 2;
#else
        xvalues[i] = 1;
#endif
        xindex[i] = (demandRows-1) + i;
    }
    xvalues[0] = -1;

    // add a column for each demand
    for(int n=0; n<demandRows; n++)
    {
        // modify the first index to create a column for this demand
        xindex[0] = n;
        CoinPackedVector cpv(columnheight, xindex, xvalues);
        matrix->appendCol(cpv);
        numberOfColumns++;
    }

    return matrix;
}

int SBPPMasterProblem::isLinkInPath(int link, int *path, int plen)
{
    for(int i=0; i<plen; i++)
        if(path[i] == link)
            return 1;
    return 0;
}

void SBPPMasterProblem::addPath(Dpath *dp, int demidx, int dest,
    SBPPSubProblem *sp)
{
    int qr, icount=0, val;
#ifdef RSBPP
    int n1, n2;
#endif

    // Find the size of the Coin packed vector
    int primPathSize = dp->len1*(nlinks-1);
    int backPathSize = dp->len2*dp->len1;
    int vectorsize  = primPathSize + backPathSize +1;

    double value[vectorsize];
    int    index[vectorsize];

    // Add the indeces and values for the primary path.
    for(int a=0; a<dp->len1; a++)
    {
#ifdef RSBPP
        net->getNodesConnectedToSpan(dp->p1[a], &n1, &n2);
        (n1==dest || n2==dest) ? val=1 : val=2;
```

```
#else
        val = 1;
#endif

        for(qr=0; qr<(nlinks-1); qr++)
        {
            index[icount  ] = demandRows + dp->p1[a]*(nlinks-1) + qr;
            value[icount++] = val; // sbpp => val=1, rsbpp => val={1,2}
        }
    }

    // Add indeces and values for the backup path
    for(int b=0, c=0; b<dp->len2; b++, c=0)
        for(qr=0; qr<nlinks; qr++)
        {
            if(qr==dp->p2[b]) {c=1; continue;}
            if(isLinkInPath(qr, dp->p1, dp->len1))
            {
                index[icount  ] = demandRows + dp->p2[b]*(nlinks-1) + qr-c;
                value[icount++] = 1;
            }
        }

    // Add the index and value for the demand
    index[icount  ] = demidx;
    value[icount++] = -1;

    // These will differ if a link is used more than once
    assert(icount==vectorsize);

    // Add the column to the problem
    try
    {
        CoinPackedVector cpv(vectorsize, index, value);
        cplex->addCol(cpv, 0, infinity, 0);
    }
    catch (CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred when adding a new path\n");
        fprintf(stderr, "The error happened in class %s in %s\nError "
            "Message: %s\n", e.className().c_str(), e.methodName().c_str(),
            e.message().c_str());
        fprintf(stderr, "(%i) prim links   : ", dp->len1);
        for(int i=0; i<dp->len1; i++)
            fprintf(stderr, "%i ", dp->p1[i]);
        fprintf(stderr, "\n(%i) backup links : ", dp->len2);
        for(int i=0; i<dp->len2; i++)
            fprintf(stderr, "%i ", dp->p2[i]);
        fprintf(stderr, "\n");
        ((SBPP_MIP*)sp)->test();
        assert(EXIT_DUE_TO_COINERROR);
    }
}
```

```
void SBPPMasterProblem::solve()
{
    if(!isFirstSolve)
        return resolve();

    int numberOfRows = nlinks*(nlinks-1)+demandRows;

    // Set the column bounds and objective values
    debugMsg("[MasterProb] Seting the col bounds.\n");
    double collb[numberOfColumns], colub[numberOfColumns], obj[numberOfColumns];

    for(int c=0; c<numberOfColumns; c++)
    {
        collb[c] = 0;
        colub[c] = infinity;
        obj  [c] = (c<nlinks) ? net->getSpanCost(c) : 0;
    }

    // set the row bounds
    debugMsg("[MasterProb] Seting the row bounds.\n");
    double rowlb[numberOfRows], rowub[numberOfRows];

    for(int r=0; r<numberOfRows; r++)
    {
        rowlb[r] = -infinity;
        rowub[r] = (r<demandRows) ? -demand[r] : 0;
    }

    // load the problem into cplex
    cplex->loadProblem(*cpm, collb, colub, obj, rowlb, rowub);
    cplex->setObjSense(1); // '-1' maximize '1' minimize
    cplex->messageHandler()->setLogLevel(0);

    // do some checking
    assert( cplex->getNumRows() == numberOfRows    );
    assert( cplex->getNumCols() == numberOfColumns );

    // let cplex solve it
    debugMsg("[MasterProb] Solving the problem.\n");
    try
    {
        cplex->initialSolve();
        cacheValues();
        printf("(%i) Obj val: %g\n", cplex->getNumCols(), objValue);
        isFirstSolve = 0;
    }
    catch(CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred when doing the"
            " initial solve\n");
        fprintf(stderr, "The error happened in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
```

```
    }

    assert(cplex->isProvenOptimal() == 1 );
}

void SBPPMasterProblem::resolve()
{
    debugMsg("[MasterProb] Re-solving..\n");

    try
    {
        cplex->resolve();
        cacheValues();
        printf("(%i) Obj val: %g\n", cplex->getNumCols(), objValue);
    }
    catch(CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred when doing a resolve\n");
        fprintf(stderr, "The error happened in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }

    assert(cplex->isProvenOptimal() == 1 );
}

double SBPPMasterProblem::solveAsMIP()
{
    // set the index sepcifying which variables are integer
    int intidx[nlinks];
    for(int i=0; i<nlinks; i++)
        intidx[i] = i;
    try
    {
        cplex->setInteger(intidx, nlinks);
        cplex->branchAndBound();
        cacheValues();
        cplex->setContinuous(intidx, nlinks);
    }
    catch(CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred when doing solveAsMIP\n");
        fprintf(stderr, "The error happened in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }

    return objValue;
}

void SBPPMasterProblem::cacheValues()
{
```

```
    objValue = cplex->getObjValue();

    const double *rp = cplex->getRowPrice();
    for(int i=0; i<demandRows; i++)
        rowPriceCach[i] = rp[i];

    numNewColsCach = cplex->getNumCols() - numColsCach;
    numColsCach    = cplex->getNumCols();
    numRowsCach    = cplex->getNumRows();
}

int SBPPMasterProblem::getNumCols()
{
    return numColsCach;
}

int SBPPMasterProblem::getNumNewCols()
{
    return numNewColsCach;
}

double SBPPMasterProblem::getObjValue()
{
    return objValue;
}

double SBPPMasterProblem::getDualSpanCost(int r)
{
    assert( r < nlinks );
    const double *dp = cplex->getRowPrice() + demandRows + r*(nlinks-1);
    double sum=0;

    for(int i=0; i<(nlinks-1); i++)
        sum += dp[i];

    return -sum;
}

void SBPPMasterProblem::testError()
{
    try
    {
        throw CoinError("There is no problem :) You asked for this.",
            "testError", "SBPPMasterProblem");
    }
    catch(CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred when trying to provoke "
            "an error\n");
        fprintf(stderr, "The error happened in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }
```

```
}

double SBPPMasterProblem::getDualDemandCost(int d)
{
    return -rowPriceCach[d];
}

double SBPPMasterProblem::getLinkFailurePrize(int ij, int qr)
{
    const double * dualrowprice = cplex->getRowPrice();
    assert( ij != qr     );
    assert( qr <  nlinks );
    assert( ij <= nlinks );
    int index = ij*nlinks + qr + demandRows;
    if(qr>ij) index--;
    return isEqual(dualrowprice[index],0) ? 0 : ABS(dualrowprice[index]);
}

double SBPPMasterProblem::getLinkRowPrice(int i)
{
    const double * dualrowprice = cplex->getRowPrice();

    assert( i < nlinks*(nlinks-1) );
    assert( !isnan(dualrowprice[demandRows + i]) );

    return ABS(dualrowprice[demandRows + i]);
}

void SBPPMasterProblem::pruneColumns()
{
    int deadColumns[cplex->getNumCols()-nlinks];
    int ndead=0;

    debugMsg("[MasterProb] Pruning columns.\n");
    try
    {
        const double* reducedCost = cplex->getReducedCost();

        // delete all the columns with a 'too high' reduced cost
        for(int c=nlinks; c<cplex->getNumCols(); c++)
            if(reducedCost[c] > 0)
                deadColumns[ndead++] = c;

        cplex->deleteCols(ndead, deadColumns);

        numNewColsCach = cplex->getNumCols() - numColsCach;
        numColsCach    = cplex->getNumCols();
    }
    catch(CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred when pruning columns from "
            "the master problem\n");
        fprintf(stderr, "The error happened in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
```

```
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }
}

void SBPPMasterProblem::printDpathBasedOnColumn(int col)
{
    const CoinPackedMatrix *matrix = cplex->getMatrixByCol();
    int colstart = matrix->getVectorStarts()[col];
    int idx, last=-1, link, count=0;

    for(int i=0; i<matrix->getVectorSize(col); i++)
        if((idx=matrix->getIndices()[colstart+i]) >= demandRows)
        {
            link = (idx-demandRows)/(nlinks-1);

            if(link != last)
            {
                printf("%i ", link);
                count = 0;
            }

            if(++count == nlinks-1)
            {
                printf("\b* ");
                count = 0;
            }

            last = link;
        }
}

void SBPPMasterProblem::printSolutionColumns()
{
    int basiscols = nlinks+demandRows;
    int i=0;

    try
    {
        const double *sol = cplex->getColSolution();
        printf(" id col links (* prim)\n");
        printf("----------------------\n");
        for(int c=basiscols; c<cplex->getNumCols(); c++)
        {
            if(sol[c] > 1e-4)
            {
                printf("%3i %i [", i++, c);
                printDpathBasedOnColumn(c);
                printf("\b]\n");
            }
        }

    }
    catch(CoinError e)
```

```
    {
        fprintf(stderr, "\nA CoinError occurred when fetching the "
            "solution vector\n");
        fprintf(stderr, "The error happened in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }
}

int SBPPMasterProblem::getMatrixElement(int row, int col)
{
    const CoinPackedMatrix *matrix = cplex->getMatrixByCol();
    int colstart = matrix->getVectorStarts()[col];

    for(int i=0; i<matrix->getVectorSize(col); i++)
        if(matrix->getIndices()[colstart+i] == row)
            return (int)matrix->getElements()[colstart+i];
    return 0;
}

void SBPPMasterProblem::printSolutionMatrix()
{
    int basiscols = nlinks+demandRows;

    try
    {
        const double *sol = cplex->getColSolution();

        // create matrix header
        printf("price | ");
        for(int i=0; i<nlinks; i++)
            printf(" y ");
        for(int j=basiscols; j<cplex->getNumCols(); j++)
            printf("%s", isEqual(sol[j], 1)?"[x]":" x ");
        printf("\n");

        // create matrix rows
        for(int r=0; r<cplex->getNumRows(); r++)
        {
            printf("%5.2f | ", ABS(cplex->getRowPrice()[r]));
            for(int c=0; c<cplex->getNumCols(); c++)
                if(c<nlinks || c>=basiscols)
                {
                    int a = getMatrixElement(r,c);
                    if(a==0)
                        printf("   ");
                    else
                        printf("%2i ", a);
                }
            printf("\n");
        }
    }
    catch(CoinError e)
```

```
    {
        fprintf(stderr, "A CoinError occurred in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }
}

void SBPPMasterProblem::printSolutionBasis(bool fShowDemand)
{
    try
    {
        const double *sol = cplex->getColSolution();

        // create matrix header
        printf("        ");
        for(int i=0; i<nlinks; i++)
            printf("   ");
        for(int j=nlinks; j<cplex->getNumCols(); j++)
            if(isEqual(sol[j], 1))
                printf("%3i", j);
        printf("\nprice | ");
        for(int i=0; i<nlinks; i++)
            printf(" y ");
        for(int j=nlinks; j<cplex->getNumCols(); j++)
            if(isEqual(sol[j], 1))
                printf("[x]");
        printf("\n");

        // create matrix rows
        for(int r=fShowDemand?0:demandRows; r<cplex->getNumRows(); r++)
        {
            printf("%5.2f | ", ABS(cplex->getRowPrice()[r]));
            for(int c=0; c<cplex->getNumCols(); c++)
                if(c<nlinks || (c>=nlinks && isEqual(sol[c], 1)))
                {
                    int a = getMatrixElement(r,c);
                    if(a==0)
                        printf("   ");
                    else
                        printf("%2i ", a);
                }
            printf("\n");
        }
        printf("\n");
    }
    catch(CoinError e)
    {
        fprintf(stderr, "A CoinError occurred in class %s in %s\nError "
            "Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }
}
```

```

void SBPPMasterProblem::writeRowPricesToFile(FILE *stream)
{
    int bytes = fwrite(rowPriceCach, sizeof(double), demandRows, stream);
    assert( bytes == demandRows );
}

void SBPPMasterProblem::readRowPricesFromFile(FILE *stream, double *buf,
    int maxread)
{
    int numRead = MIN(maxread, demandRows);
    int bytes = fread(buf, sizeof(double), numRead, stream);
    assert( bytes == numRead );
}

void SBPPMasterProblem::writeEuklidicalDistanceFromValue(FILE *stream,
    double *optimalPrice, int iteration)
{
    double sum=0;
    fprintf(stream, "%i ", iteration);
    for(int i=0; i<demandRows; i++)
    {
        sum += (rowPriceCach[i]-optimalPrice[i])*
               (rowPriceCach[i]-optimalPrice[i]);
        fprintf(stream, "%g ", (rowPriceCach[i]-optimalPrice[i])*
                                (rowPriceCach[i]-optimalPrice[i]));
    }
    fprintf(stream, "%g \n", sqrt(sum));
}
```

A.3 sbpp_sub_mip.cc

```
#include <assert.h>
#include "config.h"
#include "sbpp_sub_mip.h"
#include "error.h"

CoinPackedMatrix* SBPP_MIP::createInitialCoinPackedMatrix()
{
    int nNodes = net->getNumNodes();
    int nSpans = net->getNumSpans();
    int basewidth = nNodes*nNodes-nNodes;
    CoinPackedMatrix *matrix = new CoinPackedMatrix;

    numberOfColumns = 2*(nNodes*nNodes-nNodes) + nSpans*nSpans-nSpans;
    numberOfRows    = 2*nNodes + nSpans*nSpans;

    double linkok[2] = {1,-1};
    double nolink[2] = {0, 0};
    double *islink;
    int    findex[2];

```

```
    // Flow constraint for the X'es
    for(int i=0; i<nNodes; i++)
        for(int j=0; j<nNodes; j++)
        {
            if(i==j) continue;

            findex[0] = i;
            findex[1] = j;

            // if there is a link between node i and j
            islink = (net->getLink(i, j)>=0) ? linkok : nolink;

            CoinPackedVector cpv(2, findex, islink);
            matrix->appendCol(cpv);
        }

    // Flow constraint for the Y's
    for(int i=0; i<nNodes; i++)
        for(int j=0; j<nNodes; j++)
        {
            if(i==j) continue;

            findex[0] = nNodes + i;
            findex[1] = nNodes + j;

            // if there is a link between node i and j
            islink = (net->getLink(i, j)>=0) ? linkok : nolink;

            CoinPackedVector cpv(2, findex, islink);
            matrix->appendCol(cpv);
        }

    assert( matrix->getNumRows() == 2*nNodes    );
    assert( matrix->getNumCols() == 2*basewidth );

    double values[4] = {1,1,1,1,};
    int    index [4];

    // No-share constraint
    for(int s=0; s<nSpans; s++)
    {
        int n1, n2;
        // get the index of the nodes this span connects
        net->getNodesConnectedToSpan(s, &n1, &n2);

        // find, based on n1 and n2, the index'es.
        int i = MIN(n1, n2);
        int j = MAX(n1, n2);
        index[0] = (nNodes-1)*i + j - 1;
        index[1] = (nNodes-1)*j + i;
        index[2] = (nNodes-1)*i + j - 1 + basewidth;
        index[3] = (nNodes-1)*j + i     + basewidth;

        CoinPackedVector cpv(4, index, values);
```

```
        matrix->appendRow(cpv);
    }

    assert( matrix->getNumRows() == 2*nNodes+nSpans );
    assert( matrix->getNumCols() == 2*basewidth     );


    // Aux constraint
    for(int ij=0; ij<nSpans; ij++)
    {
        int n1, n2, n3, n4, i, j;
        // get the index of the nodes the ij span connects
        net->getNodesConnectedToSpan(ij, &n3, &n4);

        for(int qr=0; qr<nSpans; qr++)
        {
            if(ij==qr) continue;

            // get the index of the nodes the qr span connects
            net->getNodesConnectedToSpan(qr, &n1, &n2);

            // find, based on n1, n2, n3 and n4, the index'es.
            i = MIN(n1, n2);
            j = MAX(n1, n2);
            index[0] = (nNodes-1)*i + j - 1;
            index[1] = (nNodes-1)*j + i;
            i = MIN(n3, n4);
            j = MAX(n3, n4);
            index[2] = (nNodes-1)*i + j - 1 + basewidth;
            index[3] = (nNodes-1)*j + i     + basewidth;

            // add the row to the matrix. It would be cool if you could
            // just add the -1 for the z columns by setting index[4]
            // but that results in a segmentation fault.
            // Apperently you can't add columns when appending rows.
            CoinPackedVector cpv(4, index, values);
            matrix->appendRow(cpv);

        }
    }

    assert( matrix->getNumRows() == numberOfRows );
    assert( matrix->getNumCols() == 2*basewidth  );

    // The z's
    double zval[1] = {-1};
    int    zidx[1];
    int baseheight = 2*nNodes + nSpans;

    for(int z=0; z<nSpans*(nSpans-1); z++)
    {
        zidx[0] = baseheight + z;
        CoinPackedVector cpv(1, zidx, zval);
        matrix->appendCol(cpv);
```

```
    }

    assert( matrix->getNumRows() == numberOfRows    );
    assert( matrix->getNumCols() == numberOfColumns );

    return matrix;
}

void SBPP_MIP::createMatrixBoundsAndObj()
{
    int nNodes = net->getNumNodes();
    int basewidth = nNodes*nNodes-nNodes;

    //
    // Set the column bounds and objective values
    //

    // We don't need collb. The default values are 0
    // but here we set them anyway. Mmmmmmmm, memory
    collb = new double[numberOfColumns];
    colub = new double[numberOfColumns];
    obj   = new double[numberOfColumns];

    for(int c=0; c<numberOfColumns; c++)
    {
        collb[c] = 0;
        colub[c] = 1;
        obj  [c] = 0;
    }

    // change the Y upper bound to 0 if the link dosn't exist.
    for(int i=0, ij=0; i<nNodes; i++)
        for(int j=0; j<nNodes; j++)
        {
            if(i==j) continue;
            colub[ij          ] = net->getLink(i, j)<0 ? 0: 1;
            colub[ij+basewidth] = net->getLink(i, j)<0 ? 0: 1;
            ij++;
        }

    // We do not yet know the dual prices of the marster problem
    // so just set the X-prices and do the rest later.
    for(int i=0, ij=0; i<nNodes; i++)
        for(int j=0; j<nNodes; j++)
        {
            if(i==j) continue;
            obj[ij++] = net->getLinkCost(i, j);
        }

    //
    // set the row bounds
    //

    // We do not yet know witch node is start and witch is end, so we'll
```

```
    // just set that later.
    rowlb = new double[numberOfRows];
    rowub = new double[numberOfRows];

    for(int r=0; r<numberOfRows; r++)
    {
        rowlb[r] = 0;
        rowub[r] = (r<2*nNodes) ? 0 : 1;
    }

    // set the index sepcifying which variables are integer
    intidx = new int[2*basewidth];
    for(int i=0; i<2*basewidth; i++)
        intidx[i] = i;
}

void SBPP_MIP::solve(Dpath* dpath, int k, int l)
{
    int nNodes = net->getNumNodes();
    int basewidth = nNodes*nNodes-nNodes;

    // change some values in the row bounds
    rowub[k       ] = rowlb[k       ] =  1;
    rowub[k+nNodes] = rowlb[k+nNodes] =  1;
    rowub[l       ] = rowlb[l       ] = -1;
    rowub[l+nNodes] = rowlb[l+nNodes] = -1;

    // set the parameters of the objective function
    setObjParameters(l);

    // load the problem into cplex
    cplex->loadProblem(*cpm, collb, colub, obj, rowlb, rowub);
    cplex->setInteger(intidx, 2*basewidth);
    cplex->setObjSense(1); // '-1' maximize '1' minimize
    cplex->messageHandler()->setLogLevel(0);

    // do some checking
    assert( cplex->getNumRows() == numberOfRows    );
    assert( cplex->getNumCols() == numberOfColumns );

    // let cplex solve it
    try
    {
        cplex->branchAndBound();
        iterations++;
        objValue = cplex->getObjValue();
    }
    catch(CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred doing the initial solve\n");
        fprintf(stderr, "The error happened in class %s in %s\n"
            "Error Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
```

```
    }

    // If something went wrong, write some debug output
    if(!cplex->isProvenOptimal())
        writeDebugOutput(k, l);

    // restore the row-bounds
    rowub[k       ] = rowlb[k       ] = 0;
    rowub[k+nNodes] = rowlb[k+nNodes] = 0;
    rowub[l       ] = rowlb[l       ] = 0;
    rowub[l+nNodes] = rowlb[l+nNodes] = 0;

    // create dpath
    createDpathFromSolution(dpath);
}

void SBPP_MIP::writeDebugOutput(int k, int l)
{
    printf("\nPANIC!\n");
    printf("Sub problem %i<->%i was not proven optimal, "
        "here is some debug info:\n", k, l);
     // Are there a numerical difficulties?
    printf("isAbandoned()                   : %s\n",
        cplex->isAbandoned()?"true":"false");
    // Is optimality proven?
    printf("isProvenOptimal()               : %s\n",
        cplex->isProvenOptimal()?"true":"false");
    // Is primal infeasiblity proven?
    printf("isProvenPrimalInfeasible()      : %s\n",
        cplex->isProvenPrimalInfeasible()?"true":"false");
    // Is dual infeasiblity proven?
    printf("isProvenDualInfeasible()        : %s\n",
        cplex->isProvenDualInfeasible()?"true":"false");
    // Is the given primal objective limit reached?
    printf("isPrimalObjectiveLimitReached() : %s\n",
        cplex->isPrimalObjectiveLimitReached()?"true":"false");
    // Is the given dual objective limit reached?
    printf("isDualObjectiveLimitReached()   : %s\n",
        cplex->isDualObjectiveLimitReached()?"true":"false");
    // Iteration limit reached?
    printf("isIterationLimitReached()       : %s\n",
        cplex->isIterationLimitReached()?"true":"false");

    printf("\nThe crash occured after %i iterations\n", iterations);
    printf("Here are the bounds:\nrowlb: ");
    for(int i=0; i<numberOfRows; i++)
        printf("%g ", rowlb[i]);
    printf("\nrowub: ");
    for(int i=0; i<numberOfRows; i++)
        printf("%g ", rowub[i]);
    printf("\nobj  : ");
    for(int i=0; i<numberOfColumns; i++)
        printf("%g ", obj[i]);
    printf("\n");
```

```

    assert( cplex->isProvenOptimal() == 1 );
}


SBPP_MIP::SBPP_MIP(Network *network, SBPPMasterProblem *master)
{
    net      = network;
    mp       = master;
    cpm      = createInitialCoinPackedMatrix();
#ifndef USE_CLP_SOLVER
    cplex    = new OsiCpxSolverInterface;
#else
    cplex    = new OsiClpSolverInterface;
#endif

    createMatrixBoundsAndObj();
    iterations = 0;
}

SBPP_MIP::~SBPP_MIP()
{
    // Free memory for the Coin packed matrix
    delete cpm;

    // Free memory for the row and column, upper-
    // and lower bounds and objetive function
    delete[] collb;
    delete[] colub;
    delete[] obj;
    delete[] rowlb;
    delete[] rowub;

    // free memory for the integer variable index
    delete[] intidx;

    // Free memory for the cplex interface (osi)
    delete cplex;
}

void SBPP_MIP::setObjParameters(int dest)
{
    // for X
    int nNodes = net->getNumNodes();
    for(int i=0, ij=0; i<nNodes; i++)
        for(int j=0; j<nNodes; j++)
        {
            if(i==j) continue;
#ifdef RSBPP
            obj[ij++] = (j==dest) ? net->getLinkCost(i, j) :
                2*net->getLinkCost(i, j);
#else
            obj[ij++] = net->getLinkCost(i, j);
#endif
```

```
        }

    // for Z
    int basewidth = nNodes*nNodes-nNodes;
    for(int i=2*basewidth; i<numberOfColumns; i++)
    {
        obj[i] = mp->getLinkRowPrice(i-2*basewidth);
        if(isEqual(obj[i],0))
            obj[i] = 1e-4;
    }
}

void SBPP_MIP::createDpathFromSolution(Dpath* dp)
{
    int nNodes = net->getNumNodes();
    int basewidth = nNodes*nNodes-nNodes;
    int len1=0, len2=0;

    try
    {
        const double *sol = cplex->getColSolution();

        for(int i=0, ij=0; i<nNodes; i++)
            for(int j=0; j<nNodes; j++)
            {
                if(i==j) continue;

                // primary path
                if(isEqual(sol[ij], 1))
                    dp->p1[len1++] = net->getSpan(i, j);

                // backup path
                if(isEqual(sol[ij+basewidth], 1))
                    dp->p2[len2++] = net->getSpan(i, j);

                ij++;
            }
        dp->len1 = len1;
        dp->len2 = len2;

        // The objective value is the price
        dp->price = objValue;
    }
    catch(CoinError e)
    {
        fprintf(stderr, "\nA CoinError occurred fetching the "
            "solution vector\n");
        fprintf(stderr, "The error happened in class %s in %s\n"
            "Error Message: %s\n\n", e.className().c_str(),
            e.methodName().c_str(), e.message().c_str());
        assert(EXIT_DUE_TO_COINERROR);
    }
}

```

```
int SBPP_MIP::test()
{
    int nNodes = net->getNumNodes();
    int basewidth = nNodes*nNodes-nNodes;

    const double *sol = cplex->getColSolution();
    printf("X: ");
    for(int i=0; i<basewidth; i++)
        printf("%i ", isEqual(sol[i], 1)?1:0);
    printf("\nY: ");
    for(int i=basewidth; i<2*basewidth; i++)
        printf("%i ", isEqual(sol[i], 1)?1:0);
    printf("\nZ: ");
    for(int i=2*basewidth; i<cplex->getNumCols(); i++)
        printf("%g ", sol[i]);
    printf("\n");

    return 0;
}
```

A.4 sbpp_sub_bhandari.cc

```
#include <malloc.h>
#include <assert.h>
#include <string.h> // for memset
#include "config.h"
#include "sbpp_sub_bhandari.h"

#define INF            1000000
#define INVALID        -2
#define END_OF_PATH    -1

void SBPP_bhandari::breathFirstSearch(int *nodes, int nnodes,
    Network *net, int start, int end)
{
    double d[nnodes];
    int p[nnodes], i, j, o;

    // Create some sets and pointers to them. The reason for the
    // pointers is to make it quik and easy to flip sets.
    Set s1, s2, s3;
    Set *currentLayer=&s1, *neighbors=&s2, *relabeled=&s3, *tmp;

    // init distance (d) and predesesor (p)
    for(int i=0; i<nnodes; i++)
    {
        d[i] = INF;
        p[i] = start;
    }
    d[start] = 0;

    // init currentLayer
```

```
    currentLayer->add(start);

    // main BFS loop
    while(!currentLayer->isEmpty())
    {
        // for all the nodes in this layer
        while((j=currentLayer->getNext())>=0)
        {
            // check all the neighbors
            net->getNeighborSet(neighbors, j);
            while((i=neighbors->getNext())>=0)
            {
                double lji = net->getLinkCost(j,i);
                assert(d[j] + lji >= 0);
                if((d[j] + lji < d[i]) && (d[j] + lji < d[end]))
                {
                    d[i] = d[j] + lji;
                    p[i] = j;
                    if(i!=end) relabeled->add(i);
                }
            }
        }

        // make the relabeled nodes the next layer
        tmp = currentLayer;
        currentLayer = relabeled;
        relabeled = tmp;
    }

    memset(nodes, INVALID, nnodes*sizeof(int));
    for(o=-1, i=end; i!=start; o=i, i=p[i])
        nodes[i] = o;
    nodes[start] = o;
}

void SBPP_bhandari::printPath(int *nodes, int start)
{
    for(int i=start; i>=0; i=nodes[i])
        printf("%i ", i);
    printf("\n");
}

void SBPP_bhandari::createDisjointPath(Dpath* dpath, int *nodes1,
    int *nodes2, int start, int end)
{
    int *nodes;
    int len1=0, len2=0;

    dpath->price = 0;
    nodes = nodes1;
    for(int s=start, d=nodes[start]; d>=0; s=d, d=nodes[d])
    {
        dpath->p1[len1++] = net->getSpan(s,d);
        dpath->price += mp->getDualSpanCost(net->getSpan(s, d));
```

```
#ifdef RSBPP
        if(d != end)
            dpath->price += mp->getDualSpanCost(net->getSpan(s, d));
#endif
        if(nodes[d] == INVALID)
            nodes = (nodes==nodes1) ? nodes2 : nodes1;
    }

    nodes = nodes2;
    for(int s=start, d=nodes[start]; d>=0; s=d, d=nodes[d])
    {
        dpath->p2[len2++] = net->getSpan(s,d);
        for(int qr=0; qr<len1; qr++)
            dpath->price += mp->getLinkFailurePrize(net->getSpan(s, d),
                dpath->p1[qr]);

        if(nodes[d] == INVALID)
            nodes = (nodes==nodes1) ? nodes2 : nodes1;
    }

    dpath->len1 = len1;
    dpath->len2 = len2;
}

void SBPP_bhandari::bhandariDisjointPath(Dpath *dpath, int start, int end)
{
    int link;
    int nnodes = net->getNumNodes();
    int nodes1[nnodes], nodes2[nnodes];

    // find the shortest path through the network
    breathFirstSearch(nodes1, nnodes, net, start, end);

    // change the weights of the links on that path
    for(int n=start; nodes1[n]>=0; n=nodes1[n])
    {
        // set all the backward links to -w
        link = net->getLink(nodes1[n], n);
        net->setLinkWeight(link, -net->getLinkCost(link));

        // set all the forward links to (w + INF)
        link = net->getLink(n, nodes1[n]);
        net->setLinkWeight(link, net->getLinkCost(link)+INF);
    }

    // find the shortest path through the network again
    breathFirstSearch(nodes2, nnodes, net, start, end);

    // restore the weights of the links we changed before
    for(int n=start; nodes1[n]>=0; n=nodes1[n])
    {
        // set all the backward links to -w
        link = net->getLink(nodes1[n], n);
        net->setLinkWeight(link, -net->getLinkCost(link));
```

```

        // set all the forward links to (w - INF)
        link = net->getLink(n, nodes1[n]);
        net->setLinkWeight(link, net->getLinkCost(link)-INF);
    }

    // remove the overlaps betwen the two paths
    for(int i=start; i>=0; )
        if(nodes2[nodes1[i]] == i)
        {
            int next = nodes1[i];
            nodes1[i] = INVALID;
            nodes2[next] = INVALID;
            i = next;
        }
        else
            i=nodes1[i];

    // create a Dpath created by grouping the remaining links
    createDisjointPath(dpath, nodes1, nodes2, start, end);
}

SBPP_bhandari::SBPP_bhandari(Network *network, SBPPMasterProblem *master)
{
    net = network;
    mp  = master;
}

SBPP_bhandari::~SBPP_bhandari()
{
}

void SBPP_bhandari::solve(Dpath *dpath, int k, int l)
{
    bhandariDisjointPath(dpath, k, l);
}

```

A.5 network.cc

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include "error.h"
#include "network.h"
#include "utill.h"
#include "topfilereader.h"


Network::Network(const char* filename)
{
```

```
    // Get object designed to read the file format
    infile = new TopFileReader(filename);

    // Point data pointers to the data.
    nodes  = infile->getNodesData();
    nNodes = infile->getNumNodes();
    spans  = infile->getSpansData();
    nSpans = infile->getNumSpans();
    nodeToLinkMap = infile->getNodeToLinkMap();
}

Network::~Network()
{
    delete infile;
}

void Network::setWeights(int n, int *links, double *weights)
{
    assert(n<=nSpans/2);
    for(int i=0; i<n; i++)
    {
        spans[links[i]*2  ]->length = weights[i];
        spans[links[i]*2+1]->length = weights[i];
    }
}

void Network::setSpanWeight(int link, double weight)
{
    spans[link*2  ]->length = weight;
    spans[link*2+1]->length = weight;
}

void Network::setLinkWeight(int link, double weight)
{
    spans[link]->length = weight;
}

int Network::getNumNodes()
{
    return nNodes;
}

int Network::getNumSpans()
{
    // we devide by 2 because there are two direcred spans for each
    // undirected link.
    return nSpans/2;
}

double Network::getLinkCost(int i)
{
    return spans[i]->length;
}

```

```
double Network::getSpanCost(int i)
{
    // we multiply by 2 because there are two direcred links for each
    // undirected span. (i.e. link 0 and 1 is span 0, link 2 and 3 is
    // span 1, ...)
    return spans[2*i]->length;
}

double Network::getLinkCost(int i, int j)
{
    if(nodeToLinkMap[i][j]<0)
        return 999999;
    return spans[nodeToLinkMap[i][j]]->length;
}

int Network::getLink(int i, int j)
{
    return nodeToLinkMap[i][j];
}

int Network::getSpan(int i, int j)
{
    // we devide by 2 because there are two direcred links for each
    // undirected span. (i.e. link 0 and 1 is span 0, link 2 and 3 is
    // span 1, ...)
    assert( nodeToLinkMap[i][j]>=0 );
    return nodeToLinkMap[i][j]/2;
}

void Network::getNodesConnectedToSpan(int s, int *n1, int *n2)
{
    // we multiply by 2 because there are two direcred links for each
    // undirected span. (i.e. link 0 and 1 is span 0, link 2 and 3 is
    // span 1, ...)
    *n1 = spans[s*2]->orig;
    *n2 = spans[s*2]->dest;
}

void Network::displayNodesAndSpans()
{
    Link *li;
    for(int n=0; n<nNodes; n++)
    {
        li = nodes[n]->links;
        printf("node[%s] ", nodes[n]->nodeid);
        while(li != NULL)
        {
            printf("%s ", spans[li->pos]->spanid);
            li = li->next;
        }
        printf("\n");
    }
    for(int n=0; n<nSpans; n++)
        printf("span[%s] (%s --> %s) {%g}\n", spans[n]->spanid,
```

```
            nodes[spans[n]->orig]->nodeid, nodes[spans[n]->dest]->nodeid,
                spans[n]->length);
}

void Network::displayInternalMatrix()
{
    for(int i=0; i<nNodes; i++)
    {
        for(int j=0; j<nNodes; j++)
            nodeToLinkMap[i][j] == -1 ? printf(" - ") :
                printf("%2i ", nodeToLinkMap[i][j]);
        printf("\n");
    }

}

void Network::getNeighborSet(Set *set, int n)
{
    assert(n>=0);
    assert(n<nNodes);

    set->clear();
    Link *li = nodes[n]->links;

    while(li != NULL)
    {
        set->add(spans[li->pos]->dest);
        li = li->next;
    }
}

void Network::writeGAMSNetwork()
{
    // Nodes
    printf("SET NODE /");
    for(int n=0; n<nNodes; n++)
        printf("%s, ", nodes[n]->nodeid);
    printf("\b\b/;\n\n");

    // Spans
    for(int l=0; l<nSpans; l++)
        printf("SPAN('%s','%s')=1\n", nodes[spans[l]->orig]->nodeid,
            nodes[spans[l]->dest]->nodeid);
    printf("\n");

    // Spancost
    for(int l=0; l<nSpans; l++)
        printf("SPAN_COST('%s','%s')=%g\n", nodes[spans[l]->orig]->nodeid,
            nodes[spans[l]->dest]->nodeid, spans[l]->length);
    printf("\n");

}

void Network::writeGnuplotNetwork()
```

```
{
    int xmax=0, xmin=999999;
    int ymax=0, ymin=999999;

    // Nodes
    for(int n=0; n<nNodes; n++)
    {
        printf("set label %i \"%s\" at %i,%i center\n", n+1,
            nodes[n]->nodeid, nodes[n]->x, nodes[n]->y);
        xmax = MAX(xmax, nodes[n]->x);
        xmin = MIN(xmin, nodes[n]->x);
        ymax = MAX(ymax, nodes[n]->y);
        ymin = MIN(ymin, nodes[n]->y);
    }

    // Spans
    for(int i=0; i<nSpans; i+=2)
        printf("set arrow %i from %i,%i to %i,%i nohead\n", i+1,
            nodes[spans[i]->orig]->x, nodes[spans[i]->orig]->y,
            nodes[spans[i]->dest]->x, nodes[spans[i]->dest]->y);

    printf("set noborder; set noxtics; set noytics\n");
    printf("plot [%i:%i][%i:%i] 0 notitle lt 0 lw 0\n", xmin, xmax,
        ymin, ymax);
}

```

A.6 lex.cc

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include "error.h"
#include "lex.h"

LexicalAnalyser::LexicalAnalyser(const char* filename)
{
    // Open the file
    if((ff = fopen(filename, "r"))==NULL)
        fatalError("Error: file \"%s\" not found\n", filename);

    // init some stuff
    nextChar();
    line = 0;

    // Get first token
    nextToken = yylex();
}

LexicalAnalyser::~LexicalAnalyser()
{
    fclose(ff);
```

```
}

char LexicalAnalyser::nextChar()
{
    ch = fgetc(ff);
    if(ch == 0)
    {
        /* A NULL-char was found inside the file try the next one */
        return nextChar();
    }
    return ch;
}

int LexicalAnalyser::isAlpha(char c)
{
    return ((c>='a') && (c<='z')) || ((c>='A') && (c<='Z')) || c=='_';
}

int LexicalAnalyser::isWspace(char c)
{
    return ((c==' ') || (c=='\r') || (c=='\t'));
}

int LexicalAnalyser::isDigit(char c)
{
    return ((c>='0') && (c<='9'));
}

int LexicalAnalyser::yylex()
{
    int n;

    // ignore whitespace
    if(isWspace(ch))
    {
        for(nextChar(); isWspace(ch); nextChar());
        return yylex();
    }

    // Count lines
    if(ch=='\n')
    {
        line++;
        nextChar();
        return yylex();
    }

    // numbers
    if(isDigit(ch))
    {
        n=0;
        for(yylval=0; isDigit(ch); nextChar())
        {
            yytext[n++] = ch;
```

```
            yylval = yylval*10 + (ch-'0');
        }
        if(ch=='.')
        {
            yytext[n++] = '.';
            nextChar();
            for(int i=10; isDigit(ch); nextChar(), i*=10)
            {
                yytext[n++] = ch;
                yylval = yylval + (double)(ch-'0')/i;
            }
            yytext[n] = '\0';
            return T_FLOAT;
        }
        else
        {
            yytext[n] = '\0';
            return T_INT;
        }
    }

    // negative numbers
    if(ch=='-')
    {
        nextChar();
        yylex();
        if(nextToken!=T_INT && nextToken!=T_FLOAT)
            fatalError("'-' found in front of a  non-number\n");
        yylval = -yylval;
        return nextToken;
    }

    // Strings
    if(isAlpha(ch))
    {
        for(n=0; isAlpha(ch) || isDigit(ch); nextChar())
            yytext[n++] = ch;
        yytext[n] = '\0';
        return T_STR;
    }

    // default
    nextToken = ch;
    yytext[0] = ch;
    yytext[1] = '\0';
    nextChar();
    return nextToken;
}

const char* LexicalAnalyser::tok2Str(int tok)
{
    static const char *tstr[] = {"UNKNOWN", "STRING", "INT", "FLOAT"};
    static char buf[] = ".";

```

```
    if(tok>=300 || tok<-1)
        return "TOK-ERR";
    if(tok>=256)
        return tstr[tok-256];

    buf[0] = tok;
    return buf;
}

void LexicalAnalyser::accept(int t)
{
    if(t!=nextToken)
        fatalError("Error: expected [%s(%i)], found [%s(%i)] on line %i\n",
            tok2Str(t), t, tok2Str(nextToken), nextToken, line+1);

    nextToken=yylex();
}

```

A.7 basefilereader.cc

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>

#include "basefilereader.h"

BaseFileReader::BaseFileReader(const char* filename)
{
    lex = new LexicalAnalyser(filename);
    nArraySize = 30;
    sArraySize = 30;
    nodes = (Node**)malloc(nArraySize*sizeof(Node*));
    spans = (Span**)malloc(sArraySize*sizeof(Span*));
    nNodes = 0;
    nSpans = 0;
}

BaseFileReader::~BaseFileReader()
{
    delete lex;
}

void BaseFileReader::addNode(char *nid, int x, int y, int size)
{
    // if the array is full, enlarge it
    if(nNodes >= nArraySize)
    {
        nArraySize*=2;
        nodes = (Node**)realloc(nodes, nArraySize*sizeof(Node*));
    }
```

```

    // add the node to the array
    nodes[nNodes++] = new Node(nid, x, y, size);
}

void BaseFileReader::addSpan(char *sid, int n1, int n2, double length)
{
    // if the array is full, enlarge it
    if(nSpans >= sArraySize)
    {
        sArraySize*=2;
        spans = (Span**)realloc(spans, sArraySize*sizeof(Span*));
    }

    // add the span to the array, there is two since this
    // is an undirected graph
    spans[nSpans++] = new Span(sid, n1, n2, length);
    spans[nSpans++] = new Span(sid, n2, n1, length);

    // add the span to the two nodes it connects
    nodes[n1]->links = new Link(nSpans-2, nodes[n1]->links);
    nodes[n2]->links = new Link(nSpans-1, nodes[n2]->links);

    // add the span to the map for easy lookup
    nodeToLinkMap[n1][n2] = nSpans-2;
    nodeToLinkMap[n2][n1] = nSpans-1;
}

int BaseFileReader::findNode(char *nid)
{
    for(int n=0; n<nNodes; n++)
        if(strcmp(nodes[n]->nodeid, nid)==0)
            return n;
    return -1;
}

int** BaseFileReader::newEmptyMatrix(int size)
{
    int *data, **index;

    // allocate the memory
    data = (int*)malloc(size*size*sizeof(int));
    index = (int**)malloc(size*sizeof(int*));

    // create the index for easy access
    for(int i=0; i<size; i++)
        index[i] = data + i*size;

    // make sure it's empty
    memset(data, -1, size*size*sizeof(int));

    return index;
}

```

```
Node** BaseFileReader::getNodesData()
{
    return nodes;
}

int BaseFileReader::getNumNodes()
{
    return nNodes;
}

Span** BaseFileReader::getSpansData()
{
    return spans;
}

int BaseFileReader::getNumSpans()
{
    return nSpans;
}

int** BaseFileReader::getNodeToLinkMap()
{
    return nodeToLinkMap;
}

```

A.8 topfilereader.cc

```
#include <strings.h>
#include "topfilereader.h"
#include "error.h"

void TopFileReader::file()
{
    header();
    nodepart();
    spanpart();
}

void TopFileReader::header()
{
    // for now just bypass the header
    while(strcmp(lex->yytext, "NODE")!=0)
        lex->accept(lex->nextToken);
}

void TopFileReader::nodepart()
{
    char nid[10];
    int x, y, nsize;

    // Process node header
```

```
    lex->accept(T_STR); // NODE
    lex->accept(T_STR); // X
    lex->accept(T_STR); // Y
    lex->accept(T_STR); // SIZE

    // read all the nodes
    while(strcmp(lex->yytext, "SPAN")!=0)
    {
        strcpy(nid, lex->yytext);
        lex->accept(T_STR);

        x = (int)lex->yylval;
        lex->accept(T_INT);

        y = (int)lex->yylval;
        lex->accept(T_INT);

        nsize = (int)lex->yylval;
        lex->accept(T_INT);

        // add node to list
        addNode(nid, x, y, nsize);
    }
    nodeToLinkMap = newEmptyMatrix(nNodes);
}

void TopFileReader::spanpart()
{
    char sid[10], orig[10], dest[10];
    double length;
    int n1, n2;

    // Process span header
    lex->accept(T_STR); // SPAN
    lex->accept(T_STR); // O
    lex->accept(T_STR); // D
    lex->accept(T_STR); // LENGTH
    lex->accept(T_STR); // UNITCOST

    // read all the spans
    while(lex->nextToken != T_EOF)
    {
        strcpy(sid, lex->yytext);
        lex->accept(T_STR);

        strcpy(orig, lex->yytext);
        lex->accept(T_STR);

        strcpy(dest, lex->yytext);
        lex->accept(T_STR);

        length = lex->yylval;
        if(lex->nextToken==T_INT)
            lex->accept(T_INT);
```

```
        else
            lex->accept(T_FLOAT);

        // unitcost is missing from the file????
        //unitcost = la->yylval;
        //lex->accept(T_NUMBER);

        n1 = findNode(orig);
        n2 = findNode(dest);
        if(n1<0) fatalError("Error: Span '%s' is between nodes '%s' "
            "and '%s', but '%s' is not found in the node list\n",
            sid, orig, dest, orig);
        if(n2<0) fatalError("Error: Span '%s' is between nodes '%s' "
            "and '%s', but '%s' is not found in the node list\n",
            sid, orig, dest, dest);

        // add the span to the spanlist
        addSpan(sid, n1, n2, length);
    }
}

TopFileReader::TopFileReader(const char *filename) :
    BaseFileReader(filename)
{
    file();
}
```

Appendix B

Network formats

This appendix briefly describes the different network formats used.

B.1 Top format

The "top" format is a home made format. Not that I am the inventor, but it is not standardized in any way, and not widely accepted (or known) outside the institute of Informatics and Mathematical Modeling (IMM), DTU.

The data file consists of 3 blocks: *the header*, *the nodes* and *the spans*. The header begins at the top of the file and ends with the reserved word "NODE". The nodes begin with the reserved word "NODE" and ends with the reserved word "SPAN". The Spans begins with the reserved word "SPAN" and ends with the end of the file.

```
NAME: testnet2.top
DATE LAST MODIFIED: 20050513
MODIFIED BY: Kasper Bonne Rasmussen

NODE    X       Y       SIZE
N1      1       1       -1
N2      3       1       -1
N3      6       1       -1
N4      9       1       -1

SPAN    O       D       LENGTH  UNITCOST
S01     N1      N2      1
S02     N1      N3      1
S03     N2      N3      1
S04     N2      N4      1
S05     N3      N4      1
```

B.2 xy2 format

The xy2 format is also a format used primarily on IMM, DTU. It consists of 3 lines describing the number of nodes, edges and demands respectively, and two lines that describe the physical size of the network. (Used to display the network graphically).

The rest of the file are lines describing individual nodes, edges and demands. The different types of lines need not be in any particular order, but it is customary to order them by type. Below is an example representing the `testnet2` network.

```
NumberOfNodes    4
NumberOfEdges    5
NumberOfDemands  10

Dimx   120
Dimy   140

Node     0     20   33     0
Node     1     35   26     0
Node     2     44   24     0
Node     3     60   22     0

Edge     0      1     -1
Edge     0      2     -1
Edge     1      2     -1
Edge     1      3     -1
Edge     2      3     -1

Demand     0      1     1
Demand     0      2     1
Demand     0      3     1
Demand     1      2     1
Demand     1      3     1
Demand     2      3     1
```

The initial declarations of the size of the network, makes this format somewhat easier to parse than the top format, but the fact that you can't give the nodes and edges descriptive names makes it less suited for display.

Finally it is my opinion that the demand of the network is not a property of the network, but a result of the current situation and therefore should not be included in the network description.

Consequently all the networks are defined in the top format, and the xy2 format is avoided when ever possible.

Appendix C

Network structures

The following is a visual representation of the networks used in this thesis.

A gnuplot-script is generated using the `Network::writeGnuplotNetwork()` function and gnuplot is then used for generating the images. The networks `testnet` and `testnet2` are considered debug/test networks and are not shown.

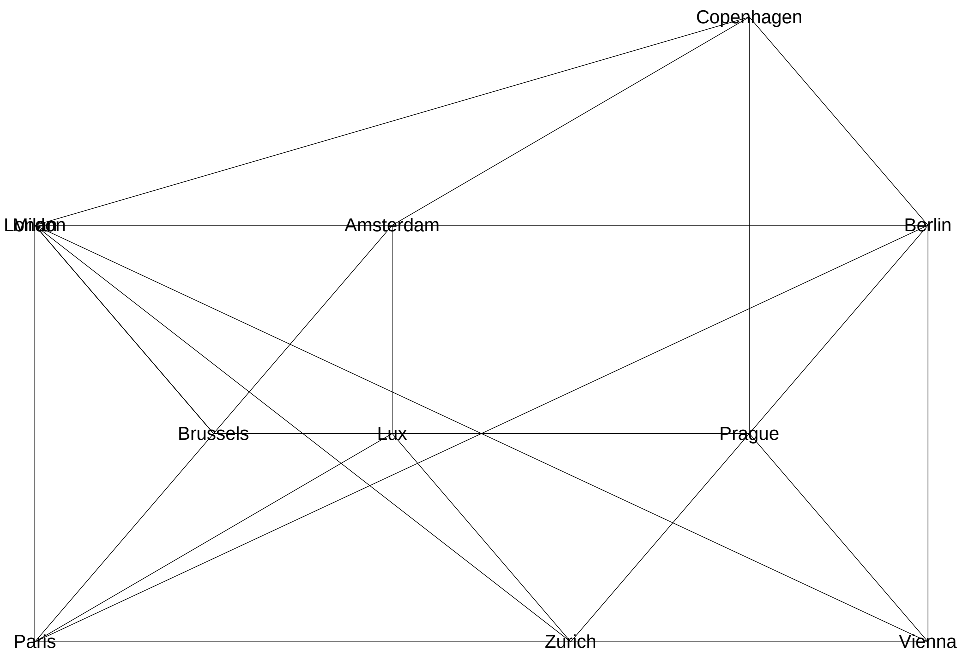

Figure C.1: The `11n26s` network. (Also known as `cost239`)

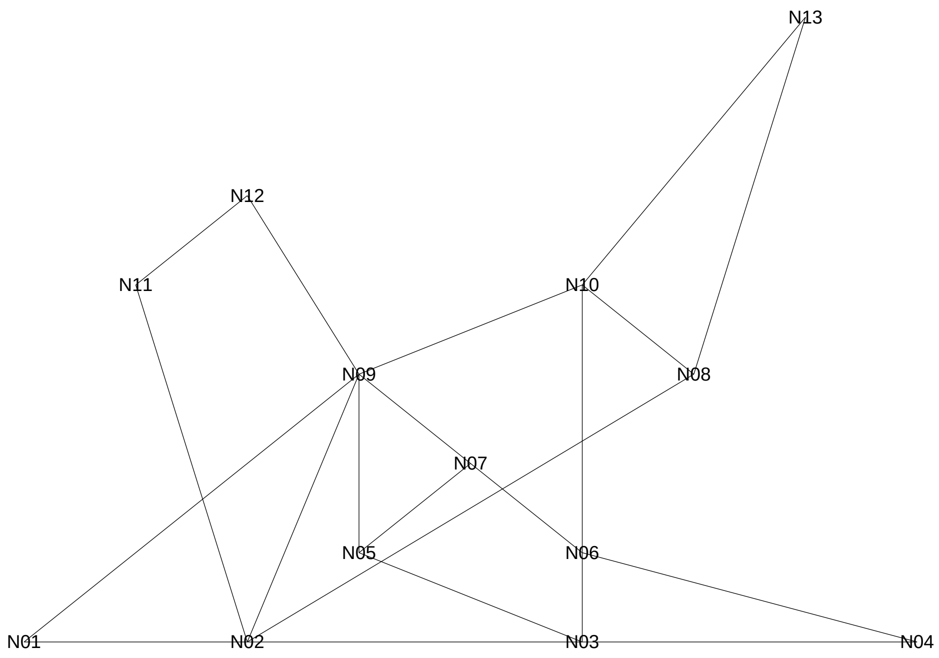

Figure C.2: The `13n21s` network. (Also known as `PanEuropean`)

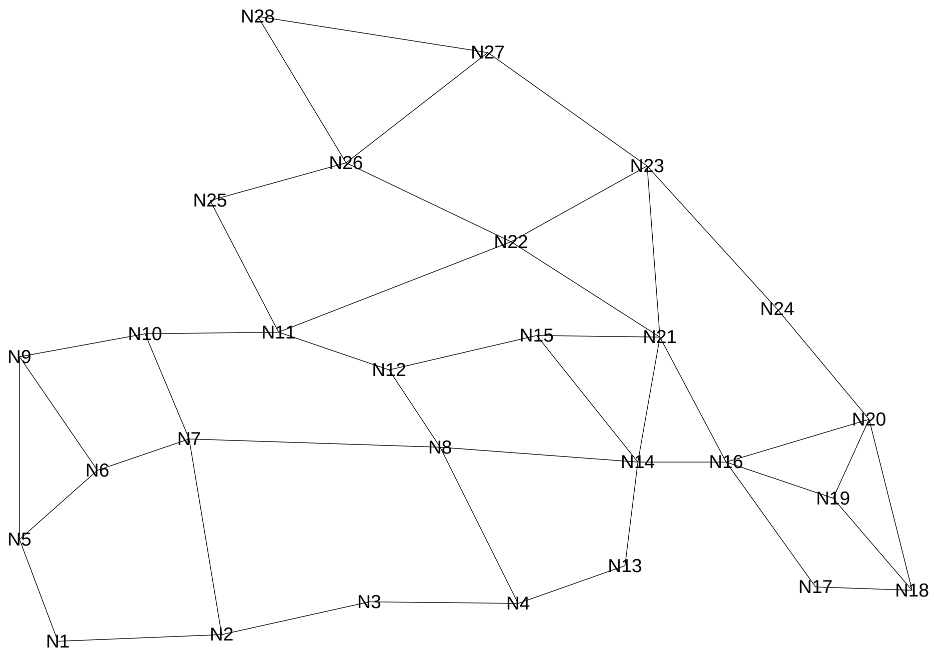

Figure C.3: The `28n45s` network. (Also known as `USANetwork`)

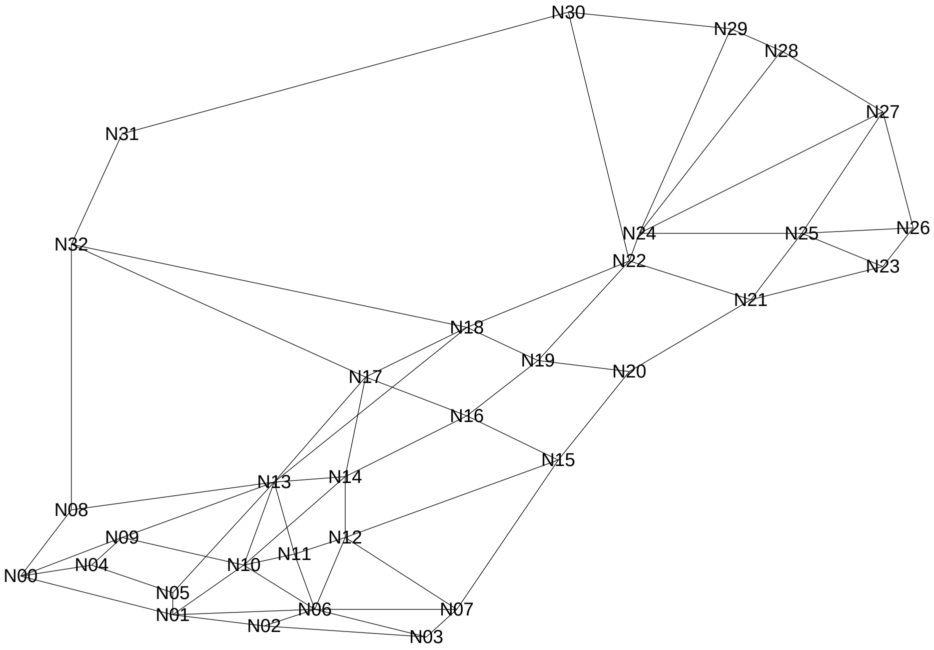

Figure C.4: The `33n68s` network. (Also known as `Italy`)

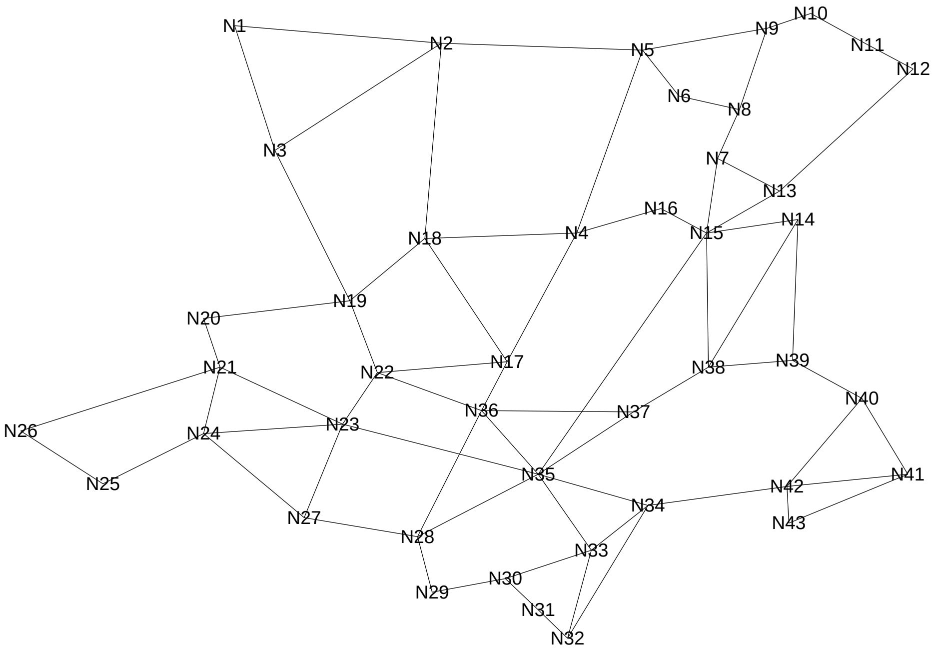

Figure C.5: The `43n71s` network. (Also known as `France`)

Bibliography

[1] COmputational INfrastructure for Operations Research (COIN–OR) http://www.coin-or.org/

[2] Eusebi Calle, José L. Marzo, Anna Urra, Lluís Fàbrega *Enhancing Fault Management Performance of Twostep QoS Routing Algorithms in GMPLS Networks*, IEEE Communications Society, 2004.

[3] Eusebi Calle, José L. Marzo, Anna Urra, *Protection performance components in MPLS networks*, Computer Communications 27 (2004) 1220–1228.

[4] Frederick S. Hillier, Gerald J. Lieberman, *Introduction to Operations Research*, 7th edition, McGraw-Hill, 2001.

[5] J.W.Surballe, R.E.Tarjan, *A Quick Method for Finding Shortest Pairs of Disjoint Paths*, John Wiley & Sons, Inc., 1984.

[6] Laurence A. Wolsey, *Integer programming*, John Wiley & Sons, Inc., 1998, ISBN: 0-471-28366-5

[7] Mikkel Sigurd, David Ryan, *Stabilized Column Generation for Set Partitioning Optimization*, Aug 15, 2003

[8] Olivier du Merle, Daniel Villeneuve, Jacques Desrosiers, Pierre Hansen, *Stabilized column generation*, Discrete Mathematics, 1999.

[9] Paul E. Black, Dictionary of Algorithms and Data Structures, NIST. http://www.nist.gov/dads/HTML/complexityClass.html

[10] Ramesh Bhandari, *Survivable Networks, Algorithms for Diverse Routing*, 1999

[11] Richard Kipp Martin, *Large Scale Linear and Integer Optimization*, Kluwet Academic Publications, 1999.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 2nd edition, MIT Press, 2001.

[13] Thomas Stidsen, Peter Kjærulff, *Complete Rerouting Protection*, Informatics and Mathematical Modeling, Technical University of Denmark, 2005.

[14] Thomas Stidsen, Tommy Thomadsen, *Joint Routing and Protection Using p-cycles*, Informatics and Mathematical Modeling, Technical University of Denmark, 2005.