Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services

Robustness Analysis and Capacity Management of the KPN (PS) Mobile Core Network

R.M.A. Imamdi
(1535676)

Committee members:
Supervisor: Dr. Ir. F.A. Kuipers
Mentor: Ing. J. Kromjong (KPN)
Member: Prof. Dr. Ir. P.F.A. Van Mieghem
Dr. E. Onur

September 3, 2010
M.Sc. Thesis No: PVM 2010 – 065

Preface

In the 4th quarter of 2009, KPN (”Koninklijke PTT Nederland NV”) launched a project, to boost up the capacity management of its packet switched (PS) mobile core network. The fastest growing PS services are already running on a core network and most new services should be accommodated as well. KPN’s capacity management project consists of both research and development aspects. The M.Sc. graduation project of the author is part of the research aspects. This thesis primarily focuses on robustness analysis and capacity management, where the PS mobile core network is treated as a case study. A more elaborate report, called the KPN deliverable, which also focuses on the details of the network has been delivered to KPN. The boundaries of this thesis were based on the available network-specific information at KPN (during the time of the graduation project) and the time limit for the M.Sc. project set by TU Delft.

I am grateful to many people, who provided me with information, documentation and support during the course of the project. In the first place I would like to thank my university Professor Piet Van Mieghem, my supervisor at TU Delft Fernando Kuipers and my company supervisor Johan Kromjong. I am also thankful to Jeroen van Huessen, who always shared his knowledge and experience regarding the PS mobile core. Furthermore, I would like to thank the managers (Will Boesveld, Marco de Nooijer, and Ben Perk), who gave me the opportunity to participate in the capacity management group at KPN. Last but not least I would like to thank the whole Care Customer Department for their kind cooperation and for the fun times at KPN.

Note 1: This thesis is accompanied by a CD containing programming work, network drawings, tables with network specific information, network related matrices, the KPN deliverable, and this thesis.

Note 2: As this version of the thesis is meant to be publicly available, the author has deliberately removed some classified information in order to protect KPN’s investments. Complete or partial information has been removed from figure 2.6, table 4.2, figure B1, figure E1, table E3 and table E4.

Summary

This thesis treats **robustness analysis** and **capacity management**, such that the techniques and tools used (and devised) for research are applicable to any network, while KPN's near future PS mobile core network is used as a case study. As a necessary step, this core network has been made insightful in the form of network and graph theoretical drawings on the subnet level, but also on the level of the complete core.

Most of the academic work focuses on robustness analysis, where edge, vertex and algebraic-connectivity are treated as robustness measures. A survey is made based on these three connectivity types. For designed networks, consisting of several node types, the best way of increasing robustness is by solving either the edge-connectivity-augmentation or the vertex-connectivity-augmentation problem. These augmentation problems are to find a minimum set of new edges to be added to a graph, such that the resilience to link and node failures increases. The author has made use of existing algorithms for solving these respective problems. Both algorithms provide optimal solutions in the case of an unweighted graph. There is some freedom of choosing vertex pairs for adding new edges to increase a network's robustness, when increasing the edge and/or vertex-connectivity. Increasing the algebraic-connectivity is an NP-complete problem for the general case. The heuristic approaches submitted to the core network and its subnets gave less efficient solutions for increasing the algebraic connectivity. Some calculations have been done to see how the algebraic-connectivity of the core network changes as the edge as well as the vertex-connectivity is increased by edge addition.

The algorithm for increasing the edge-connectivity is quite abstract. The most difficult and time consuming part is to construct a cactus representation, which is a compact graph representation of all the minimum cuts of the network. An existing abstract cactus construction algorithm is analyzed and extended with 5 subroutines, such that it can be implemented. In general there are various cactus representations that can be used to represent the minimum cuts of a single network.

Finally, we address the capacity management issues from a research perspective, since no measurements were possible on the case study network, at the time of writing. Therefore, this part focuses on bandwidth management and vertex criticality of nodes. A first tool, called the CTA-edge-Betweenness program, has been programmed to compute what percentage of each edge will be loaded with traffic and to find out which edges are prone to congestion. This tool can be used as an indicator for dimensioning the bandwidth of the edges in the case where no measurements are available. A second tool, named the vertex-criticality program, computes how important each element is with respect to the network it belongs to. Both tools were tested with the PS mobile core network as the test network.

The tools discussed in the previous paragraph are useful in the process of performing proactive capacity management, especially when a network is still in its design phase. To use them effectively, complete network information and traffic routing schemes are necessary.

List of figures

List of tables

Table of Contents

1 Introduction

1.1 Background information

Within the Netherlands, KPN is the largest service provider for both fixed and mobile Telephony, and a big competitor for Television and Internet provisioning. In order to provide adequate service to its customers, KPN has to manage, maintain and expand a large network consisting of a large variety of components. The KPN mobile network is a complex network, which can be divided into several domains as shown in figure 1.1.

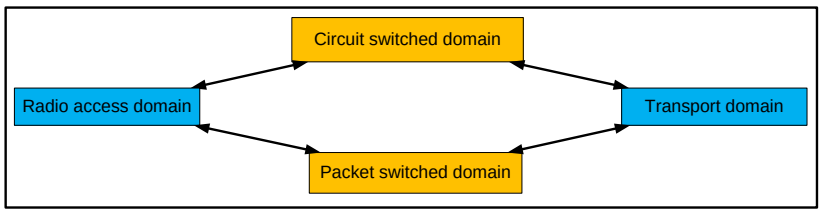

Figure 1.1: Domains of the KPN network

Currently, network and traffic information is gathered in an un-automated fashion from different tools. As a consequence the layout of the network architecture is not known in detail. Furthermore, partial management mechanisms are implemented for small parts of the core network. These "islands" are not interconnected and not completely structured yet. The data is only used when there is a problem or bottleneck somewhere in the network. This means that performance management (and capacity management) is done in a reactive way. To improve this situation, KPN has launched a capacity management project with the purpose of managing the capacity of the Packet Switched (PS) and the transmission domain [57], which form KPN's PS mobile core network.

1.2 Problem statement

The main problem that KPN wants to deal with is improving the manner in which the data traffic in the core network is currently managed. There is a danger that parts of the PS core network may get overloaded especially in busy periods. From a research perspective, the main problem is analyzing how the robustness of the network can be increased as efficient as possible.

1.3 Focus of the thesis

The **focus** of this M.Sc. thesis will be on robustness analysis and capacity management of the PS – and Transmission domain, which will often be referred to as the PS or Intelligent Edge mobile core.

Because the network architecture and topology were not known, an extra task was to make these two domains more insightful.

Within the scope of KPN's capacity management project the **main purpose** of the M.Sc. thesis is twofold. Firstly, it is important to deliver general methods and tools for increasing the robustness of a network. Secondly, general techniques and tools are required that provide initial insights in where capacity shortages and congestion are most likely to occur. For both topics, the packet switched (PS) mobile core network is treated as a case study.

1.4 Methodology and thesis outline

Methodology:

1.	Literature study regarding:	- The PS mobile core network (case study). - Graph theory. - Robustness of networks. - Connectivity and related algorithms. - Capacity management.
2.	Defining the M.Sc. thesis proposal.	
3.	Applying /formulating algorithms for increasing connectivity (and robustness).	
4.	Simulations regarding robustness analysis.	
5.	Modifying and applying algorithms for capacity management.	
6.	Simulations regarding capacity management.	
7.	Writing the thesis.	

Thesis outline:

Chapter 2 describes the network architecture and topology of the PS mobile core network, which is expected to be operational in the near future. The most important results are network drawings based on the Intelligent Edge design.

Chapter 3 focuses on robustness analysis, where edge, vertex and algebraic-connectivity are treated as measures. Research is also done regarding the relationship between the different connectivity types. Furthermore, an existing abstract algorithm is analyzed and extended, such that it can be used for writing a program that generates a representation of all min cuts (called cactus representation). This chapter contains most of the academic work.

Chapter 4 finally addresses the capacity management issues from a research perspective, where it is important to bear in mind that no measurements were possible on the target network, at the time of writing. Therefore the research is limited to bandwidth management on the network connections and a new term called vertex-criticality that indicates the importance of each network element.

2 Network Architecture of the PS mobile core network

KPN's PS mobile core network consists of several subnets, each with specific functionalities with respect to the services running on the network. The subnets are primarily responsible for processing and/or transmission of data.

Figure 2.1 shows a high level logical overview of how several networks are interconnected. The aqua-blue clouds on the right and below represent external networks from KPN's point of view. The Mobidata network is used for Operation and Maintenance purposes, while the Radio access network represents the combined GSM (2G)/UMTS (3G) cellular network of KPN in the Netherlands. The Mobile IP Backbone and the Intelligent Edge network, together form the PS mobile core network, which is the case study in this thesis.

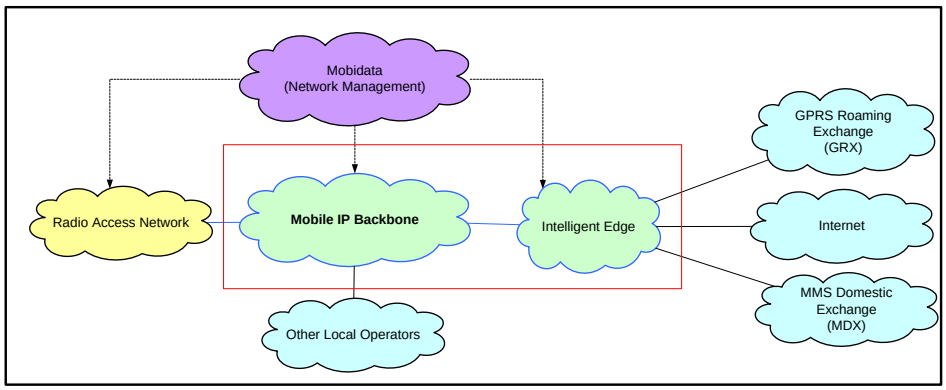

Figure 2.1: Interconnections between KPN's PS networks [2].

The physical connectivity between the IP BB and the Intelligent Edge network is quite different from the logical overview shown in figure 2.1. The Intelligent Edge network consists of 4 core PoP (Point of Presence) and 5 VRF (Virtual Routing Function) Lite locations, located in several Dutch cities, and are physically connected via the 10G mobile IP Backbone. As the name already reveals, this backbone is based on elements (routers) and connections that have a capacity of 10 Gbps. Figure 2.2 gives a high level view of the physical connectivity between the IP BB and Intelligent Edge network. The following sections describe the IP backbone and the Intelligent Edge network in more detail.

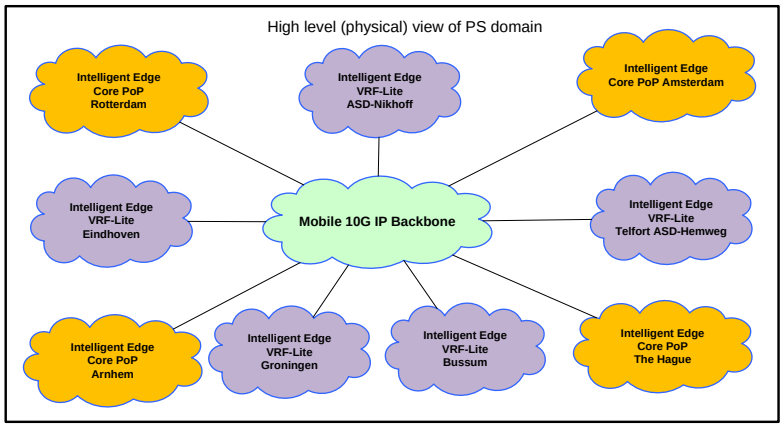

Figure 2.2: Physical connectivity 10G IP BB and Intelligent Edge [2].

2.1 The mobile IP backbone

At the time of writing, there are 2 IP Backbones, namely the currently active 1G Mobile IP Backbone and the new Mobile 10G IP Backbone[1]. In 2011, the traffic of PS services will migrate from the 1G to the new 10G backbone. The reason is that the older counterpart is getting saturated with both types of traffic (CS and PS), especially due to the exponential growth of the PS traffic. As a response on traffic growth, KPN has decided to separate these traffic types, each on its own backbone. This means that at the backbone level, the installed capacity for PS traffic increases from a shared 1 Gbps network to a dedicated 10 Gbps network, while CS traffic remains on the 1 Gbps network. As described in [37,2], the 10G IP BB provides Location redundancy, MPLS IPv4 VPNs, backbone redundancy, fast rerouting and interconnections to external networks.

The 10G IP BB will have a similar topology and functionality as its 1G counterpart, with the exception that only 4 (instead of 5) core PoP locations will constitute the new backbone. Figure 2.3 shows the 10G IP BB [58].

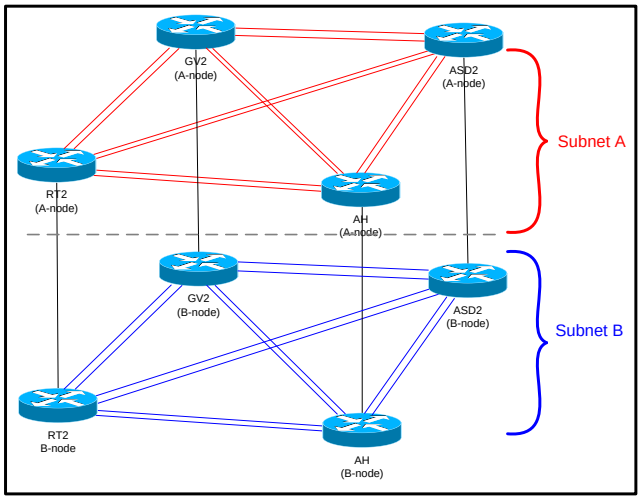

Figure 2.3: The Mobile 10G IP BB [58]

The 10G IP BB consists of 2 subnets, where the A and B collocated core routers are connected at each of the 4 core locations. The fully meshed A and B subnets, will provide the required redundancy, such that no single node (link) failure will affect the performance. All edge devices will be connected to both the A as well as the B subnet. Each of the two subnets should be able to support a maximum nominal load of 40% with respect to the entire backbone capacity.

Routing

A nice property of the 10G IP BB is that it supports load sharing between its two subnets. So in this configuration there is no primary and hot-standby subnet. Routing via transit locations is supported

[1] Note that in the future, when KPN steps over to an all IP network, there will be another migration from the Mobile 10G IP BB to the Generic IP Network (GIPN). This network is out of the scope of this thesis.

to prevent network disconnection in the case that multiple failures occur. When single failures occur, transit locations are avoided, which in turn avoids competition between single hop and multi-hop traffic paths. In order to be able to control traffic flows across the backbone and avoid unexpected behavior, all (OSPF) links are always assigned explicit link costs. There are no OSPF-enabled links on the backbone with default metrics. Furthermore, all links have the same metrics configured in both directions such that symmetry is preserved. Figure 2.4 (a) exemplifies how traffic is routed under normal conditions. Throughout the backbone the links connecting the A and B routers are assigned a weight of 100, while the cross links are configured to have a weight of 40 (to prevent routing via transit locations). Figure 2.4 (b) shows how the traffic flows in case of a link failure in the network. In case of a single backbone link failure, the cost of the best path between the two locations is 180. Therefore, paths passing through a transit location (like The Hague in the figure above) are prevented due to the higher required cost of 200. Due to a double failure in the Mobile IP Backbone, two locations may happen to have a best path via a third (transit) location.

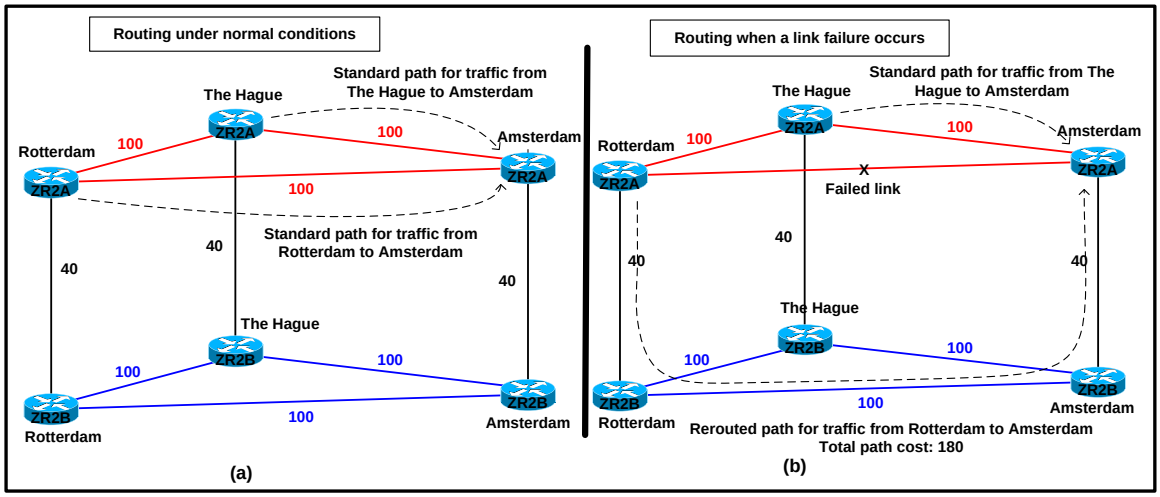

Figure 2.4: Traffic flow across the network during normal operation.

Link redundancy in the 10G IP BB

Between any combination of 2 core PoP locations two 10 Gbps connections are installed if they belong to the same subnet. If the primary link fails, the backup link will take over. With this configuration the probability of a 2 or 3 hop connection decreases.

Edge Devices

The (PS based) device types to be connected to the new Backbone are: Intelligent Edge routers, SGSNs, GGSNs, RNCs (at a later stage), MSTP signalling systems, DNS system, Li (Legal Intercept) and a CG (Charging Gateway). Clearly the voice related devices are left out in this device list, which emphasises the fact that only data traffic will be transmitted on the new IP Backbone. More detailed information about the new IP Backbone can be found in [37] and [58].

2.2 Background of the Intelligent Edge Network

The Intelligent Edge (IE) network is used to provide mobile users with (1) secure access to the Internet, (2) access to international roaming (via the GPRS Roaming Exchange) and (3) access to multimedia services. The redundancy implementation is generic for these connections. There are 2 core routers at each location, providing connectivity to an external network. When one router fails the other is able to carry the complete traffic load. There is also a notion of location redundancy, such that if one location would fail, another one would be able to take over the load. At the time of writing, Amsterdam and Rotterdam core PoP locations provide such redundancy, while the Arnhem core PoP is being upgraded, such that it can become the 3rd redundant location [2].

Intelligent Edge consists of the larger 4 core PoP subnets and the smaller 5 VRF Lite subnets (figure 2.2), where the latter serve as an extension for providing specific functionality to the core locations.

Figure 2.5 shows an example of the basic physical model of a core PoP subnet. An important implementation issue is that value added service related equipment are connected via an access switch to the core switch. When the access switch's maximum capacity is reached, another access switch is connected to the core switch [37, 56]. This basic model does not show the connection of Mobile Office Online (MOO) switches, because they are not implemented in every core location. The model is used as a building block for making the drawing of the PS mobile core network shown in figure B1 in appendix B. (See appendix A for the abbreviations).

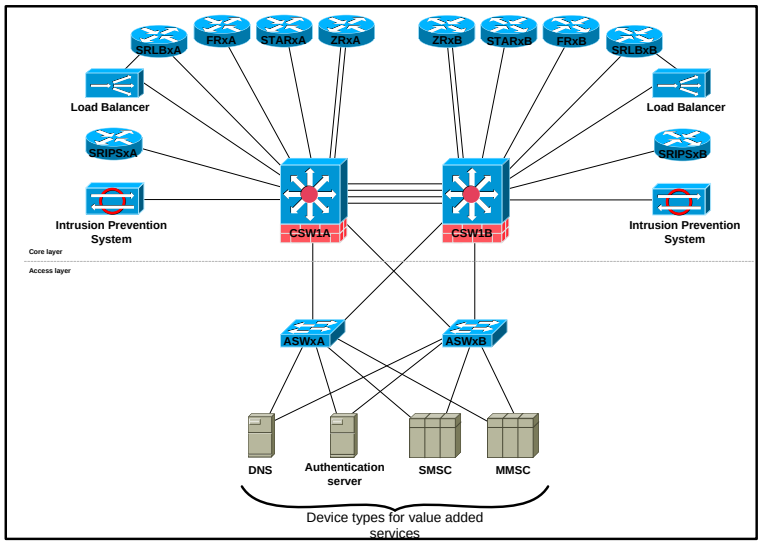

Figure 2.5: Physical model of an IE core PoP subnet [2].

Each IE core PoP consists of 3 security zones, which are logically separated by redundant firewalls. The separation into three zones is based on the level of security required to protect the network and thus leads to [37, 2]:

1) A trusted zone, which is the most secure environment and contains systems managed by KPN.
2) A semi-trusted zone, providing access to other networks/customers, based on Service Level Agreements (SLA's).
3) An untrusted zone, providing connectivity to "untrusted" networks (and customers), which are not based on SLA's. The best known examples are the connections with the Internet.

Each zone is equipped with 2 zone routers, which are connected to the IP backbone (figure 2.6). Only the zone routers of the untrusted zone in the core PoP of Amsterdam and Rotterdam provide a connection to the Internet and the GPRS Roaming Exchange (GRX) network [37,56]. There are only 2 VRF Lite routers per Lite subnet for all three zones. Each Lite router is configured to be aware of one, two or three security zones, depending on the tasks of its subnet. An important implementation issue is that information cannot be forwarded between different zones at a VRF Lite location itself. Instead such information, destined for a different zone, needs to be routed via the nearest PoP location. In the PS Mobile core network, each zone router is connected to a backbone core router, via a 1 Gbps or 10 Gbps link. VRF Lite subnets have their own physical connections with these backbone routers. Figure 2.6 exemplifies this.

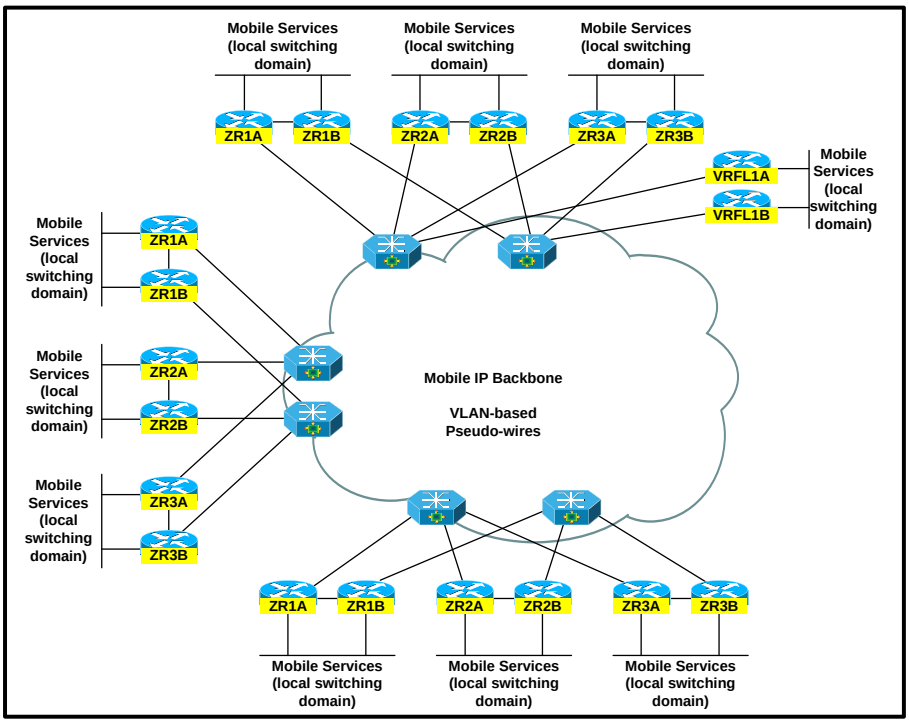

Figure 2.6: Physical connectivity of IE to IP BB [2].

2.3 The network model of Intelligent Edge with IP Backbone

An additional but necessary part of the M.Sc. project is to make a network drawing of the complete PS mobile core network according to the intelligent edge design discussed earlier in this chapter. From this network a graph $G(V, E)$ should be derived as well, such that it can be submitted to robustness and capacity management analysis.

The first suggestion was to install topology-mapping software (e.g. LanTopolog combined with LAN Surveyer) on a server and to run this server on the PS mobile core network. However this approach was cancelled, because of two issues:

1. The current network still consists of the older 1G IP BB and MIPnet[2] architecture, while the goal is to make the PS domain insightful according to the near future situation.
2. A second major issue is that running topology-mapping software may be risky.

Our approach was to use all the information in this chapter and discuss the situation with the design team of KPN. Figure B1 in appendix B shows the result and gives a good indication of how the new PS mobile core network will look like. The appendix contains a complete graph of the network as shown in figure B2. Finally figure B3 shows a relaxed graph of the core network where the clouds and (square) nodes are left out compared to figure B2. They are treated as external networks and specific functional nodes respectively. It is this relaxed graph which is treated as a case study for doing the analysis in the succeeding chapters.

The PS mobile core network can be viewed as to consist of an A and B network, which provide mutual redundancy. The load is shared between these networks. For each device (connection) in the A network, there is a similar device (connection) in the B network.

[2] MIPnet is the predecessor of the Intelligent Edge network discussed in section 2.2.

3 Robustness Analysis and Connectivity

This chapter focuses on the robustness analysis of the core network. The methods and tools are generic and can be applied to any network.

In this thesis connectivity is a term used to determine how well devices in a network are connected to each other in order to transfer data back and forth. Two network elements are said to be connected when there exist at least one (single hop or multi hop) path between them. A network is said to be connected when there exists a path between any pair of arbitrarily chosen nodes of that network. Good connectivity is achieved, when multiple paths exists between devices, especially those in the core of the network. The higher the number of disjoint paths between any node pair, the better the corresponding network will be connected.

Throughout this chapter Graph theory is used for describing and analyzing the KPN mobile core network. A graph is represented by $G(V, E)$, which consists of a set V of $|V|$ vertices and a set E of $|E|$ edges. All robustness and connectivity related simulations are done using MATLAB. All the matrices that are used as input can be found in the KPN deliverable. Before starting with the analysis, section 3.1 surveys the research already done with respect to algebraic, edge and vertex-connectivity.

3.1 A survey for connectivity as a measure for robustness

Edge and vertex-connectivity are important for the robustness due to the fact that they quantify the extent to which a graph $G(V, E)$ (and the network it represents) can accommodate to edge and vertex failures. Recent research [84, 41, 42] shows that the algebraic-connectivity is always non-decreasing (and usually increasing), with increasing edge – and/or vertex-connectivity. This implies that the algebraic-connectivity can be treated as a measure for the robustness. This survey focuses on all three connectivity types and its goal is to give a short description about the research already done by the scientific community. Furthermore, it briefly highlights a few potential topics that are still open to science.

Algebraic-connectivity: $a(G)$

Already in 1972 Miroslav Fiedler explained that the eigenvalues of the Laplacian matrix of a (simple) graph provide valuable information about the connectivity of the graph [15]. Fiedler explained that the number of eigenvalues equal to zero represents the number of connected components and that the 2nd smallest eigenvalue of a connected graph G is its algebraic-connectivity $a(G)$.

In [41, 42] the relationship between $a(G)$ and the robustness for complex networks is studied for E.R. random graphs, W.S. small world graphs and B.A. scale free graphs. It is shown that the speed with which $a(G)$ increases, by edge addition, is topology dependent and that in some cases the speed of increasing $a(G)$ is lower than increasing the vertex-connectivity $\kappa_v(G)$ by adding edges [41]. For other (complex) networks (such as (lognormal) geometric random graphs, regular lattice graphs and power law graphs) similar research is not yet conducted (to the knowledge of the author). Reference [4] also shows that $a(G)$ is a lower bound for $\kappa_v(G)$ if G is not a complete graph and that $\kappa_v(G) = |V| > a(G) = (|V| - 1)$ otherwise. Recently a study has been done, where the importance of a vertex (or edge) is quantified by the algebraic-connectivity of the remaining network after the removal of the vertex (or edge) [61]. Another different method to look at robustness is to increase $a(G)$ of a complex regular network, without adding vertices or edges, but by a technique called “random-rewiring” as described in [75].

The maximum algebraic-connectivity augmentation problem is proven to be **NP-complete** [69]. Therefore heuristic approaches have been devised to solve this problem [84].

An important point (not explicitly mentioned in the literature) is that even though $a(G)$ can be computed for weighted graphs [13, 54 and 51], it is usually the unweighted case that is treated when using $a(G)$ as a robustness measure. However, it is surely interesting to investigate the weighted $a(G)$ in relation to the robustness. In particular for the case that an edge weight represents the cost to remove the corresponding edge relative to the graph. In general the mean path length decreases as new edges are added to a graph, to increase $a(G)$. Another interesting open problem is to find an optimization that maximizes robustness and minimizes the delay time in a network as efficient as possible, when cost constraints limiting the number of vertices/edges are considered important [7].

There is a wealth of literature describing properties of the algebraic-connectivity in general, but also for special cases and/or topologies. There are already several surveys [68, 67, 66, 64 and 1] related to the algebraic-connectivity that have been published over the years, that discuss interesting topics

like the Laplacian spectrum, applications of Laplace eigenvalues, Congruence and Equivalence, weighted graphs and max-cuts, Optimal numberings, Classifications of bounds to $a(G)$ as a function of other graph invariants, applications of Fiedler vectors, etc. A lot of research has also been conducted on a special type of graph, namely the tree [28, 65, 14, 29, 54, and 53]. It is interesting to see if similar research can be applied on other topologies (e.g. directed and undirected cycle (or ring) graphs, hierarchical graphs etc.). It is also interesting to know that the properties of Fiedler's definition remain valid for digraphs [90]. Another important topic regarding $a(G)$ that has received attention is its upper and lower bound, which are summarized neatly in [1].

Edge-connectivity: $\kappa_e(G)$

Research regarding connectivity already started in the 70s when Eswaran and Tarjan introduced the terms bridge-connectivity and bi-connectivity for undirected graphs [12]. They also showed how to make a digraph strongly connected. Furthermore, they proved that solving these problems optimally for weighted directed and weighted undirected graphs is NP-complete. With this the foundation was put in place for edge-connectivity (and vertex-connectivity).

Most research has been done on **unweighted undirected** k-edge-connected graphs resulting in different approaches and many efficient algorithms are considered for special cases [12, 19] as well as the general case (for any augmentation value $\delta > 0$) [73, 88, 87, 17, 8, 25 and 72]. One of the most interesting findings is the one of Frank [17], who gave an $O(|V|^5)$ algorithm which also extends to the more general augmentation problem. Then Nagamochi and Ibaraki [72] used maximum adjacency ordering in their approach and by combining their minimum cut algorithm with the approach of Frank [17], they produced a (faster) $O(|V||E| + |V|^2 \log(|V|))$ time algorithm for augmenting a graph to achieve $(k + \delta)$-edge-connectivity. They also showed how to extend the problem for the situation where the connectivity target is a real value. Cai and Sun [8] also found an interesting algorithm that works for any multi-graph. In [44] it is indicated that the k-edge-augmentation problem, without introducing parallel new edges is NP-complete, but if the target connectivity is a predefined integer, this problem is solvable in polynomial time. In the author's opinion, understanding the construction of a cactus representation of a graph [71, 73] is important to understand the edge-augmentation problem. An algorithm for constructing a cactus that corresponds to the explanation in [73] can be found in [71]. However, this algorithm is not detailed enough to be used for programming purposes. In fact this algorithm is extended in section 3.3 to make it useful for programming purposes.

For **unweighted directed** graphs Gabow [24] found an $O(k|E|\log(|V|^2/|E|))$ time algorithm that finds the edge-connectivity, which runs slightly faster for an undirected graph. However this is not an algorithm that augments a digraph. On the other hand for unweighted directed trees (or digraphs whose underlying graph is a tree) there exists a polynomial time solution for increasing the edge-connectivity [47]. In [9] it is also shown that the successive augmentation property also holds for digraphs. Apart from the special cases (including strong connectivity), Frank [17] and Gabow [25] have found algorithms for the general digraph case with respective time complexities of $O(|V|^5)$ and $O(k(k+\delta)(|E| + \delta|V|)log^2|V|)$.

The **weighted edge-augmentation problem** is NP-complete for both undirected graphs and digraphs. However, several approximation algorithms have been devised [22, 50] for special cases. The special weighted cases for bi -, bridge – and strong connectivity have been studied by Frederickson and Já'Já'

[22] and they have found approximation algorithms with reasonable time bounds. Benczur [4] also found an algorithm that runs in $O(\min(|V|^3, \delta|V|^2))$ time. For the special weighted case, where edge costs arise from node costs, Frank [17] found a polynomial-time solution. For another special case where $G(V, E)$ is a spanning sub-graph of a 2-edge-connected weighted graph, it is possible to achieve 2-edge-connectivity in polynomial time [26]. Other special cases are treated in [89].

For **weighted digraphs** in general Jensen, Frank and Jackson [43] proved that the edge-augmentation is NP-complete. For mixed graphs (containing directed and undirected edges) they have obtained interesting results for 2 extreme cases (This includes the case of adding only directed edges or the other case of adding only undirected edges). It would be odd if no heuristic approaches would have been proposed for the weighted case. Based on maximum weight matching algorithms, Taoka and Watanabe [81] have obtained heuristic algorithms. Apart from [22, 43] not much attention has been paid to weighted digraphs. It is interesting to do research to see if there are polynomial-time or approximation algorithms for special topologies and the case of specific weight functions.

Vertex-connectivity: $\kappa_v(G)$

Vertex-connectivity-augmentation has also received much attention by the scientific world. Again most of the literature focuses on **unweighted undirected graphs**. The specific case to make a graph 2-connected is treated by Eswaran and Tarjan [12], Rosenthal and Goldner [79] and Hun and Ramachandran [38]. Watanabe and Nakamura [86] and Jordan [46] independently solved the case for achieving 3-connectivity, while Hsu [34] produced an algorithm to upgrade a 3-connected graph to a 4-connected one. Increasing the connectivity of a k-connected graph (where k can be any integer) by 1 has received most of the attention [46, 45, 56, 60, 55 and 10], which is no surprise as this is a case that is often targeted in practice. For this case Jordan [45, 46] gave an algorithm that finds an edge set larger than the optimum size by a value no more than $\lceil (k-2)/2 \rceil$. This result was extended by Jackson and Jordan [39] for the general connectivity augmentation to a set at most $\lceil (k(k+\eta-1)+4)/2 \rceil$ more than the optimum (where η is an integer augmentation value). With the general connectivity is meant that the target connectivity is $(k+\eta)$, with k indicating the connectivity of the original graph and η being any integer augmentation value. For this general augmentation, the known optimal result is an $O(|V|^5 + f(k)|V|^3)$ polynomial time algorithm by Jackson and Jordan [40], where $f(k)$ is an exponential function of k. For some special cases they prove even stronger results, such as the case where $d_{min} \geq 2k-2$. On the negative side the complexity of the vertex-connectivity augmentation problem is a longstanding open question [83, 39 and 40]. In [86] it is also mentioned that augmenting a graph G_0, with zero edges to become k-connected, where $k \geq 2$, is NP-complete. Vegh [83] has produced a polynomial time algorithm for finding an optimal solution to increase the connectivity of any k-connected unweighted undirected graph by $\eta = 1$. As it is important to understand the concept of k-shredders (k-separators) when discussing vertex-connectivity, an interesting result produced by Cheriyan [10] is not unimportant. In [10] a deterministic algorithm is treated, which finds all k-shredders in $O(k^2|V|^2 + k^3|V|^{1,5})$ time. The concept of shredders (and separators) is important for solving the vertex-connectivity-augmentation problem.

The vertex-connectivity for **digraphs** has been treated by Frank and Jordan [20]. They found a min-max formula that finds the minimum number of required new edges to make an unweighted digraph $(k+\eta)$-connected. Frank and Vegh [21] came up with an optimal polynomial time algorithm to

make a k-connected digraph $(k+1)$-connected. For the specific case of rooted directed trees Masuzuwa, Hagihara and Tokura [62] have devised an optimal sequential algorithm to achieve k'-connectivity, where k' is the target connectivity.

As the **weighted** vertex-connectivity-augmentation problem for graphs is NP-complete [83, 74 and 76], alternative approaches are required to deal with the problem. The 1st way is to consider algorithms for special cases, as discussed in [18, 34, 38, 35, 36, 46, 83 and 85]. Most of these references discuss specific connectivity targets (η and/or k are specific values) and the tree topology has also received much attention. Despite the NP-completeness, it remains interesting to find and solve more special cases (e.g. special topologies and special weight functions) for which an optimal polynomial-time solution does exist. To exemplify, such a solution is found for the special case if each edge weight is characterized by a node induced cost function [83]. A 2nd way is to develop heuristic algorithms and for this it is recommended to read [30] (for a review). Finally approximation algorithms can be designed [76, 48, 49, 50, 78 and 23] that still produce an acceptable outcome in polynomial time. To mention a good example, Khuller and Thurimella [50] have found an $O(|V|klog|V|(|E|+|V|\log|V|))$ approximation algorithm, which augments any k-edge-connected weighted graph to achieve a $(k+1)$ or even a $(k+2)$-connected graph. They also show that techniques can be used that run in $O(k|V|^2)$ time, provided that k is odd.

3.2 Increasing the algebraic-connectivity

This section focuses on increasing $a(G)$ (and therefore increasing the robustness) of the PS mobile core network by adding as few new edges as possible. As proposed by Fiedler $a(G)$ is the 2nd smallest eigenvalue of the Laplacian matrix (L) of the graph $G(V, E)$ [41], which is considered as a measure for the robustness of the graph.

Before discussing the algebraic-connectivity, the notations to be used are presented. The topology of a network is represented by a graph $G(V, E)$. The most important matrices are the Degree matrix D, the Adjacency matrix A and the Laplacian matrix $L = D - A$. D is a diagonal matrix where each entry (on the diagonal) represents the nodal degree. A is an unweighted representation of all the edges in a network. This matrix consists of 0's and 1's, where a 1 (0) at position $\{i, j\}$ indicates the existence (non-existence) of a link from node i to node j.

The set of all $|V|$ eigenvalues[3] of L is called the Laplacian spectrum of $G(V, E)$ and is represented as: $\mu_1 = 0 \leq \mu_2 \leq \cdots \leq \mu_n$. Two important pieces of information that can be obtained from this spectrum are:

1. If $\mu_2 = 0$, the graph is disconnected.
2. The number of eigenvalues equal to 0 is equal to the number of components or clusters of G.

Because multigraphs cannot be submitted to the analysis of the $a(G)$, the PS mobile core network is treated as a simple graph in this subsection. Another important point is that all the graphs to be analyzed consist of bidirectional links, which has the advantage that A and L are symmetric.

3.2.1 Strategies for increasing the algebraic-connectivity

Increasing $a(G)$ with the fewest links is proven to be NP-complete [69]. This is the reason that heuristic strategies are applied in the analysis of increasing $a(G)$, where the target graph is converted to a simple graph. The first strategy is an idea of the author, while the other two were obtained from chapter 5 of [84]. The strategies differ in the way that a "new" edge to be added is chosen.

Strategy 1:

This strategy starts out with computing L and the corresponding $a(G)$. After the 1st edge is added, the new L and corresponding $a(G)$ are computed. This procedure is done m' times, namely for all the $m' = (|V|(|V| - 1)/2 - |E|)$ possibilities to add a new edge. However, only that edge which gives the maximum $a(G)$ is stored along with its corresponding L. If there is a tie the first edge with the maximum $a(G)$ is chosen. If the "maximum $a(G)$" edge is found, then for the next iteration, the corresponding L becomes the starting situation for adding the 2nd new edge. The 2nd, 3rd till the last

[3] Eigenvalues are calculated using the following equation: det(L-μI), where I is an identity matrix with the same dimensions as the Laplacian matrix. More information regarding eigenvalues can be found in chapter 4 of [77].

new edge to be added are chosen the same way as the 1st one. The robustness is non-decreasing, because $a(G)$ cannot decrease by adding edges.

Strategy 2:

This strategy is based on increasing $a(G)$ by sequentially adding a link between a node of minimum degree and any other node. The first node found in D with minimum degree is chosen and the new link connecting this node and another node which yields maximum increase of $a(G)$, is added. A and D conforming to this addition are stored and the next iteration can start, using these matrices as its starting point. Note that this strategy attempts to increase the minimum degree of the overall graph and therefore it increases $a(G)$.

Strategy 3 (Fiedler vector strategy):

This strategy makes use of the Fiedler vector $\boldsymbol{b} = (b_1, \dots, b_{|V|})$, which is the vector corresponding to the 2nd smallest eigenvalue. The 2 vertices, to which the next new edge should be added to increase $a(G)$, is derived from this vector. The vector-indices, say i and j, of the 2 values b_i and b_j in $\boldsymbol{b}$, whose absolute difference $|b_i - b_j|$ is the maximum (compared to every other possible combination), correspond to the indices of these vertices (v_i and v_j) and are used to find them. The edge is added between these vertices, $a(G)$ is computed and A and D are modified accordingly. The next iteration (where the 2nd new edge is added) can start using the modified matrices as its input. Every next iteration proceeds in a similar way, until the predefined number of iterations is achieved.

The next subsection discusses the results obtained from simulations done in Matlab, according to the above strategies. The structure of the MATLAB source codes are presented in appendix C.

3.2.2 Results of increasing the algebraic-connectivity

This subsection presents the most important results obtained regarding the simulations for increasing $a(G)$. The results (regarding the 3 strategies) of subnet core PoP ASD/RT[4] and those of the complete network are presented. The reason for also doing the simulations on subnet level is because it yields results that are relatively cheap, as only indoor edges are to be added. The figures corresponding to the results of core PoP ASD/RT are shown in appendix B. For each figure representing a strategy, there is a summary of how many links are required to achieve $a(G) > 1$ and of course $a(G)$ itself. A value just higher than 1, means that the graph is at least 2-connected, as $a(G)$ is a lower bound of the (vertex) connectivity. Note that for a graph (which is not complete) the following holds:

$$a(G) \leq \kappa_v(G) \leq \kappa_e(G) \leq d_{min}(G) \qquad (3.1)$$

The target of at least 2-connectedness has physical meaning for the network, because any single link or node failure is not enough to disconnect the network.

[4] Results of the other IE subnets are similar and do not add anything extra from an academic perspective. They can be found in the KPN deliverable.

Increase of algebraic-connectivity of core PoP ASD/RT

Figure 3.1 represents the graph for the identical subnets in Amsterdam and Rotterdam. Figure 3.2 shows the increase in $a(G)$ as "new" edges are added according to the 3 strategies. The figure clearly indicates that for subnet ASD/ RT, strategy 3 (strategy 1) is the most (least) efficient in increasing $a(G)$[5]. For few "new" edges (less than 6) this is not always true, but even then the differences are minor.

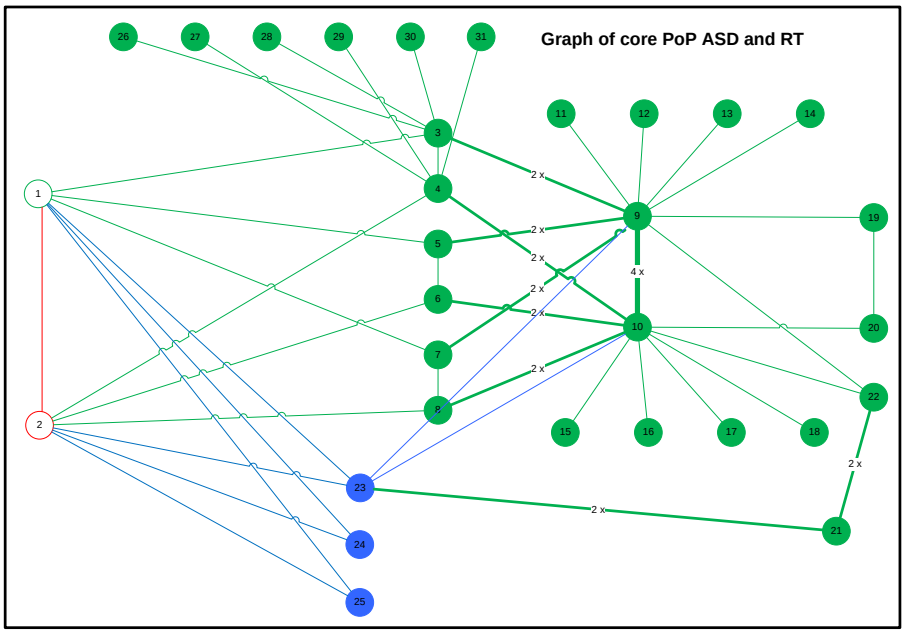

Figure 3.1: Current situation regarding core PoP ASD/RT and corresponding BB routers.

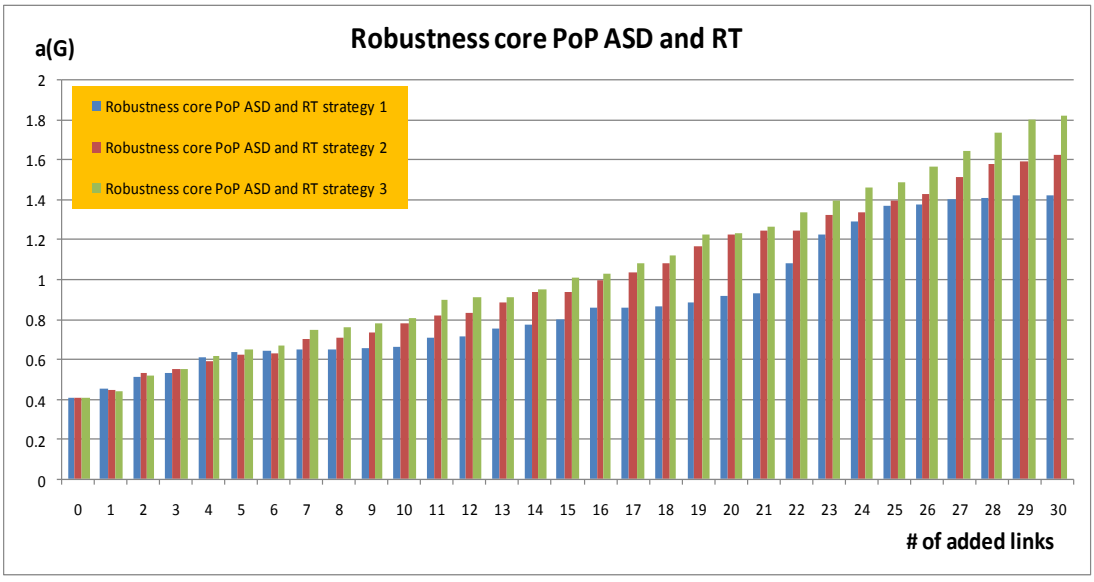

Figure 3.2: Comparison of suboptimal strategies for ASD/RT.

[5] This holds for all the other Intelligent Edge subnets.

Algebraic-connectivity of the IP BB

From figure 2.4 it is clear that $a(G)$ of the mobile 10G IP BB exceeds the target ($a(G) > 1$). It is therefore pointless in running the experiments for the backbone. What is interesting is the fact that $a(G)_{IP\ BB} = 2$, because this subnet is 4-connected. This corresponds with the leftmost inequality of equation 3.1.

Increase of the algebraic-connectivity of the entire graph

In reality it is not possible to add an edge between any two arbitrary vertices, due to the fact that not all network nodes perform the same functions or process the same traffic streams. Using any of the 3 strategies, the results (appendix D) are such that many “new” edges are incompatible with the way the network works and/or too expensive to be implemented. This is not unexpected, because firstly the strategies do not distinguish between different node types and secondly these strategies will most likely choose some expensive edges that connect vertices belonging to different subnets as they tend to increase $a(G)$ the most.

3.3 A programmable algorithm for cactus construction

Before it is explained how the edge-connectivity can be increased, it is important to understand how a cactus representation, (R, φ), is constructed. (R, φ) represents every minimum cut of the original graph $G(V, E)$, in a compact form, where $R = R(W, F)$ is the cactus graph consisting of node-set W and link-set F and $\varphi: V \rightarrow W$ is a mapping of the vertices $v \in V$ of G to the nodes $w \in W$ of R [6]. Cactus construction is quite difficult and often a time consuming task, especially if $G(V, E)$ represents a large network. This section focuses on extending and explaining Nagamochi's algorithm [70] that can be used for the construction of (R, φ). In the knowledge of the author, and via contact with Nagamochi it is believed that no implementation presently exists.

Nagamochi's algorithm (figure 3.3 and 3.4) does not indicate how to perform a couple of essential steps or how to obtain some information. The main contribution here is to develop and test programmable algorithms to help perform these key steps, which are then integrated into Nagamochi's algorithm, such that the latter can be used for programming purposes. The contributions are the following:

1. An algorithm for choosing an edge $\{s, t\} \in E(G)$ for performing line 3 in algorithm 3.2.
2. An algorithm for the construction of an *st*-MC-partition (a partition of the graph as a result of all the min cuts separating s and t), required in line 11.
3. An algorithm for the construction of an *st*-Cactus representation $(R_{(s,t)}, \varphi_{(s,t)})$, required in line 4.
4. An algorithm for merging multiple cacti and joining their mappings to construct an integral cactus representation (R, φ), as indicated in line 16.
5. An algorithm for converting a cactus representation into a cycle-type normal cactus (CNC) representation.

It is relevant to understand the parent and child algorithm (figures 3.3 and 3.4, respectively) that constitute Nagamochi's algorithm, such that it becomes clear to the reader how the contributed algorithms fit in the former to finally produce algorithms 3.3 and 3.4 (figure 3.5 and 3.6, respectively). Algorithms 3.1 and 3.2 are explained next, while the 5 new algorithms (which are considered to be subroutines of algorithm 3.2) are explained in subsection 3.3.1 to 3.3.5.

Algorithm 3.1 is used to compute the minimum cut value of the target graph G^* (line 1) and to initialize a list V^{old} (line 2) that should keep track of already found minimum cuts, such that they can be identified as old. Once identified they will be prevented from being used more than once in algorithm 3.2, which is the child of algorithm 3.1 and called in line 3 of the latter. The parent-child approach is used, because algorithm 3.2 is recursive as it may call itself as shown in lines 8 and 14.

[6] From now on the entities of a cactus are referred to as nodes and links, while those of a target graph are referred to as vertices and edges. When vertices are contracted together, the element containing these vertices is also referred to as a node. This may prevent confusion and ambiguity from the reader's perspective.

If the graph, of which a cactus representation should be constructed, consists of only 1 vertex, the cactus will be trivial ($R(w, \emptyset)$), meaning that it consists of 1 node and no edges. In this case the 1st line of algorithm 3.2 does the job. It is more interesting to see what happens if G consists of multiple vertices connected by edges. In this case an edge is chosen between vertices s and t (line 3).

Algorithm 3.1 Construct

Input: An edge weighted graph G^*.

Output: The Cycle-type Normal Cactus (CNC) representation (R, φ) for $\mathcal{C}(G^*)$.

1. Compute $\lambda := \lambda(G^*)$; *(% λ is the minimum cut value of G^*)*
2. $V^{old} := \emptyset$; *(% initialize an empty list)*
3. $(R, \varphi) :=$ Cactus(G^*, V^{old}, λ) *(% make a call to the subroutine named Cactus)*

Figure 3.3: Parent algorithm for constructing the cactus representation.

Algorithm 3.2 Cactus(G^*, V^{old}, λ)

Input: A graph G, a subset $V^{old} \subset V(G)$, and a real number λ.

Output: A cactus representation (R, φ) for a set $\mathcal{C}' \subseteq \mathcal{C}(G)$ of minimum cuts.

1. **if** |V(G)| = 1 **then return** the trivial cactus (R, φ).
2. **else**
3. Choose an edge $e = \{s, t\} \in E(G)$, with capacity $c_G(e) > 0$.
4. **if** $\lambda(s, t; G) > \lambda$ or the (s, t)-cactus representation $(R_{(s,t)}, \varphi_{(s,t)})$ represents no minimum cut other than those $\{\bar{v}, V(G) - \bar{v}\}, \bar{v} \in V^{old}$. *(% $\bar{v}$ represents an already found minimum cut)*
5. **then**
6. $G := G/\{s, t\}$;
7. $V^{old} := V^{old} - \{s, t\}$;
8. $(R, \varphi) :=$ Cactus(G, V^{old}, λ) *(% recursive call)*
9. **return** (R, φ)
10. **else**
11. **for** each V_i in the (s, t)-MC-partition $MC_{st} = (V_1, V_2, \dots, V_r)$ **do**
12. $G_i := G/(V(G) - V_i)$, denoting by $x_{\overline{V_i}}$ the vertex obtained by contracting $(V(G) - V_i)$;
13. **if** $d(V_i; G) = \lambda$ **then** $V_i^{old} := \left(V^{old} \cap V_i\right) \cup \{x_{\overline{V_i}}\}$ **end**.
14. $(R_i, \varphi_i) :=$ Cactus$(G_i, V_i^{old}, \lambda)$ *(% recursive call)*
15. **end**
16. $(R, \varphi) := merge\{\left(R_{(s,t)}, \varphi_{(s,t)}\right), (R_1, \varphi_1), \dots, (R_r, \varphi_r)\}$;
17. Convert (R, φ) into CNC representation.
18. **return** (R, φ)
19. **end**
20. **end**

Figure 3.4: Child algorithm for constructing the cactus representation.

Line 4 it tests if the chosen edge is critical or if the *st*-cactus-representation $(R_{(s,t)}, \varphi_{(s,t)})$ (explained in section 3.3.3) contains only old cuts (of V^{old}). If the edge is not critical or if all the cuts are elements of V^{old}, the algorithm executes lines 6-9. Line 6 actually shows that a contraction[7] of *s* and *t* should take place, which has the result that the target graph G changes. Therefore, the list V^{old} should be updated as such, that $\{s, t\}$ is removed from it as shown in line 7. With this modification a

[7] Contraction: A subset of vertices is merged together into a single new node and all self loops are removed. All the edges that were previously connecting the contracted vertices with other vertices are connecting these other vertices with the new node, after the contraction. This type of contraction is sometimes called edge-contraction.

new minimum cut can be detected or the number of vertices (of G) decreases after each recursive call. In line 8 such a recursive call is made and line 9 returns the cactus representation.

If none of the conditions in line 4 are satisfied, the algorithm continues from line 11. The *st*-MC-partition ($MC_{st} = (V_1, V_2, \dots, V_r)$) is necessary for executing the for-loop from lines 11-15. This is a partitioning of the graph based on all the min cuts separating *s* and *t* and is explained in more detail in section 3.3.2. The loop is required to find the indivisible cuts C_{ind}, which are not compatible with $MC_{st} = (V_1, V_2, \dots, V_r)$. These cuts are called indivisible because they do not separate *s* and *t*. The compatible cuts $\in C_{comp}$ are the ones separating *s* and *t*. They are compatible with $MC_{st} = (V_1, V_2, \dots, V_r)$ and are represented by the *st*-cactus-representation $(R_{(s,t)}, \varphi_{(s,t)})$.

Contraction is used again in line 12, with respect to all vertices, except the ones in $V_i \in MC_{st}$ (for each $i = \{1,2, \dots, r\}$). The contracted node is referred to as $x_{\overline{V_i}} = (V(G) - V_i)$. In line 13 it is shown that a new list V_i^{old} is created if the degree of V_i, denoted as $d(V_i; G)$, is equal to the min cut value λ. Notice that $V_i^{old} = x_{\overline{V_i}}$ if V^{old} is empty. It is important to understand this, especially due to the fact that a recursive call is made in line 14 to obtain a cactus representation (R_i, φ_i) of graph G_i. If the old cuts were not managed in this way, the algorithm would result in an infinite number of recursions and never terminate, because a contracted graph (the child of the originating graph) would always give an old cut, even if it were already detected in its parent graph. For a more elaborate explanation the author refers to section 5.3.2 of [71].

Line 16 indicates that the *st*-Cactus representation $(R_{(s,t)}, \varphi_{(s,t)})$ representing the set of compatible cuts C_{comp} should be merged together with each cacti (R_i, φ_i) for each $i = \{1,2, \dots, r)$, to obtain a complete cactus representation (R, φ). All the cuts taken together from each of the aforementioned cacti represent the set of indivisible cuts C_{ind}. From lemma 5.21 of [71] the union of C_{comp} and C_{ind} represent all minimum cuts of graph G ($C(G) = C_{comp} \cup C_{ind}$). This means that (R, φ) represents all minimum cuts of G.

However, the aforementioned cactus representation is not necessarily a unique representation of $C(G)$ for G. This means that there are more possible cactus representations, which adequately represent all the min cuts of the same graph. In line 17 (R, φ) is converted to a cycle-type normal cactus (CNC) representation. The latter is simplified and still represents each and every min cut of G. There may also be multiple possible CNC representations for the same graph. Think of the case when a different *s* and/or a different *t* is chosen.

Figures 3.5 and 3.6 show the modified Construct and Cactus algorithms, respectively, where the contributions of the author are written in blue en explained in the succeeding subsections. By combining these explanations with those of algorithm 3.1 and 3.2, it should be possible to understand algorithm 3.3 and 3.4 and to write a program accordingly.

Time complexity of algorithm 3.3 and 3.4

Line 6 of algorithm 3.4 calls subroutine “Update st-MC-partition”, which has a complexity of $O(|V|^6)$, as computed in section 3.3.2. This is the part with the highest complexity of algorithm 3.4. Let n_{calls} be the number of times that algorithm 3.4 is invoked due to its recursive nature. Lemma 5.27 of [71] shows that $n_{calls} \leq 9|V| - 7$. The complexity of algorithm 3.4 is therefore $O(n_{calls}|V|^6) = O(|V|^7)$. This results in the fact that the complexity of algorithm 3.3 is also $O(|V|^7)$.

Algorithm 3.3 Construct

Input: An edge weighted graph G^*.

Output: The CNC representation (R, φ) for $C(G^*)$.

1. Compute $\lambda := \lambda(G^*)$;
2. $V^{old} := \emptyset$;
3. $(R, \varphi) := \text{Cactus}(G^*, V^{old}, \lambda)$
4. **set** special-recursive-call = false;

Figure 3.5: Resulting parent algorithm for constructing cactus representations.

Algorithm 3.4 Cactus(G^*, V^{old}, λ)

Input: A graph G, a subset $V^{old} \subset V(G)$, and a real number λ.

Output: A cactus representation (R, φ) for a set $C' \subseteq C(G)$ of minimum cuts.

1. **if** |V(G)| = 1 **then return** the trivial cactus (R, φ).
2. **else**
3. Choose vertex pair {s,t}, such that $\{s, t\} \in E(G)$; *(% usually s (t) has the smallest (largest) label)*
4. **call** → subroutine choose s and t;
5. **call** → subroutine Construct *st*-MC- partition;
6. **call** → subroutine Update *st*-MC- partition;
7. **call** → subroutine Constructing *st*-cactus-representation;
8. **if** $\lambda(s, t; G) > \lambda$ or the (s, t)-cactus representation $(R_{(s,t)}, \varphi_{(s,t)})$ represents no minimum cut other than those $\{\bar{v}, V(G) - \bar{v}\}, \bar{v} \in V^{old}$.
9. **then**
10. $G := G/\{s, t\}$;
11. $V^{old} := V^{old} - \{s, t\}$;
12. **set** special-recursive-call = false;
13. $(R, \varphi) := \text{Cactus}(G, V^{old}, \lambda)$ *(% recursive call)*
14. **return** (R, φ)
15. **else**
16. **for** each V_i in the (s, t)-MC-partition $MC_{st} = (V_1, V_2, \dots, V_r)$ **do**
17. $G_i := G/(V(G) - V_i)$, denoting by $x_{\overline{V_i}}$ the vertex obtained by contracting $(V(G) - V_i)$;
18. **if** $d(V_i; G) = \lambda$ **then** $V_i^{old} := (V^{old} \cap V_i) \cup \{x_{\overline{V_i}}\}$ **end**.
19. **set** special-recursive-call = true;
20. $(R_i, \varphi_i) := \text{Cactus}(G_i, V_i^{old}, \lambda)$ *(% recursive call)*
21. **end**
22. **call** → subroutine Merger of cacti;
23. **call** → subroutine Construct CNC representation;
24. **return** (R, φ)
25. **end**
26. **end**

Figure 3.6: Resulting child algorithm for constructing cactus representations.

3.3.1 An algorithm for choosing an edge when constructing a cactus

This subsection focuses on the 1st sub-algorithm, called algorithm 3.5 (figure 3.7). In line 3 of algorithm 3.2 it is shown that an edge $e = \{s, t\} \in E(G)$ with a capacity $c_G(e) > 0$ should be chosen. According to this line any two vertices s and t may be arbitrarily chosen, as long as they are

connected by an edge[8]. One way to do this is to simply call the first and the last vertex s and *t*, respectively. However, one could use a strategy, such that the choice is made in a "smart" way, as is done by algorithm 3.5. This "smart" way of choosing has the advantage that the minimum cuts are found in less recursive calls and therefore makes the algorithm faster.

Algorithm 3.5: Choose s and t

Input: special-recursive-call *(%true or false)*
G_i and map_i *(%map_i maps the vertices of the original graph G^* to those of G_i)*
$x_{\overline{Vi}}$
$\lambda = \lambda(G^*)$
Node s and node t
Output:s and t, such that {s,t} is an edge

```
1.  if special-recursive-call = false
2.      return s and t;
3.  else
4.      for each v ∈ V(G_i)
5.          if degree(v) = λ
6.              add v to mdn; (% mdn is a vector of minimum degree nodes)
7.          end
8.      end
9.      if # of elements of mdn ≥ 2
10.         for each i, j ∈ mdn, ( i ≠ j)
11.             if there is an edge {i, j}, such that i ≠ x_Vi ≠ j
12.                 s: = i; t: = j;
13.             elseif there is an edge {i, j}, such that i ≠ x_Vi and j = x_Vi.
14.                 s: = i; t: = j;
15.             else
16.                 s: = i; t ∈ Γ(s)(preferably t ≠ j);
                    (%Γ(s) is the neighbor list of vertex s)
17.             end
18.         end
19.     elseif # of elements of mdn =1
20.         s ∈ mdn; t ∈ Γ(s);
21.     else
22.         Choose an arbitrary node as s; t ∈ Γ(s);
23.     end
24. end
```

Figure 3.7: An algorithm for choosing an edge {s,t}.

Algorithm 3.5 verifies if it is invoked during a recursive call in line 14 (not the one in line 8) or by the parent algorithm for the first time. If it is the latter, it just takes the *s* and *t* that is treated as input by algorithm 3.5. When algorithm 3.5 is invoked during a recursive call (line 14), the algorithm makes a

[8] Algorithm 3.2 says that an edge should be chosen, but actually it is the vertices connected by the edge that are important, because the *st*-MC-partition and *st*-cactus-representation use s and t as input.

proper choice according to the network's topology, such that the new cuts can be found in less recursive calls. It first examines and stores all the vertices of degree equal to the minimum cut value (lines 4-8). After that it verifies if there is more than 1 (or just 1) of such vertices, by examining how many elements are contained in the minimum degree nodes (mdn) vector. If there are at least 2 vertices in the mdn vector (line 9-18), and there exists an edge between them, one of them is chosen as *s* and the other as *t*. Notice that there is a preference not to choose $x_{\overline{V_i}}$. This is due to the fact that $x_{\overline{V_i}}$ is an old cut and by choosing a different vertex (if possible), there is a bigger chance of finding a new cut. Figure 3.8 (a) and (b) show examples when (not) to choose $x_{\overline{V_i}}$. Notice that the colored vertices belong to the mdn vector. The third case (as exemplified by figure 3.8 (c)) occurs when there exists no edge between the vertices in the mdn vector (line 16). Preferably, a vertex $s \neq x_{\overline{V_i}}$ of the mdn vector is chosen and a neighbor is chosen as *t*. If there is only one vertex in the mdn vector, it is chosen as *s* and a neighbor as *t* as shown in (line 20 and figure 3.8 (d)). Finally, if there are no vertices in the mdn vector, an arbitrary connected vertex pair may be chosen (line 22 and figure 3.8 (e)).

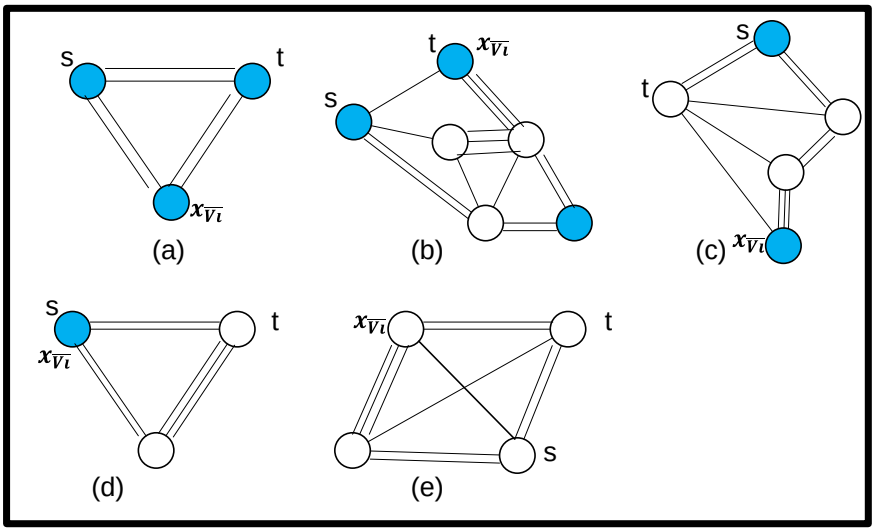

Figure 3.8: Examples of choosing s and t where the min cut value is 4.

Time complexity of algorithm 3.5

The time complexity of algorithm 3.5 (and those in the succeeding 4 subsections) will be derived by analyzing the lines of the pseudo code that have a significantly high complexity order. Therefore they can be used to determine the worst case time performance, for high values of the variable $n = |V|$. Lines with relatively low complexity order have no significant impact in the determination of the complexity order and are not discussed in detail. (E.g. lines 1-3 of algorithm 3.5 do not contribute a significant amount of time to determine the worst case time bound). Lines 4-8 has a worst case running-time function of $T_{(4,8)}(n) = c_{(4,8)}n$, where the constant $c_{(4,8)}$ is chosen sufficiently large[9]. To determine $T_{(9,24)}(n)$, the following lines (line 10, 16, 20 and 22), with significantly high order of computation time are analyzed. The result is: $T_{10}(n) = c_{10}n^2$, $T_{16}(n) = c_{16}n^2$, $T_{20}(n) = c_{20}n$ and $T_{22}(n) = c_{22}n$. Notice that the power of n depends on the amount of iterations that the respective

[9] Every constant c_i or $c_{(i,j)}$ is chosen sufficiently large in its respective worst case function, whenever the complexity of an algorithm is analyzed.

line of code undergoes. For these 4 lines, the number of loops (nested in each other) determines the number of iterations. We find $T_{(9,24)}(n) = c_a n^2$ if lines 10-17 are executed or $T_{(9,24)}(n) = c_b n$ otherwise. The worst case running-time function of algorithm 3.5 is found to be $T_{alg\ 3.5}(n) = cn^2$, by adding $T_{(4,8)}(n)$ with $T_{(9,24)}(n)$. This means that the complexity is $O(|V|^2)$. (For information regarding complexity analysis, it is advised to read [11]).

3.3.2 Constructing the *st*-MC-partition

This section describes two algorithms that when used together are able to compute the *st*-MC-partition of an arbitrary connected graph. An *st*-MC-partition is a partitioning of the target graph G^* into smaller subsets of vertices $V_i \in st$-MC-partition, where $(i = 1, \dots, r)$. This partitioning is based on the $r-1$ compatible minimum cuts. Because they separate s from t they are said to be compatible with the *st*-MC-partition. The next figure gives an example of such a partition on a graph consisting of 9 vertices. There are 4 min cuts that are compatible with the *st*-MC-partition $= (V_1, V_2, V_3, V_4, V_5) = ([1,2], [3], [4,5,6], [7,8], [70])$.

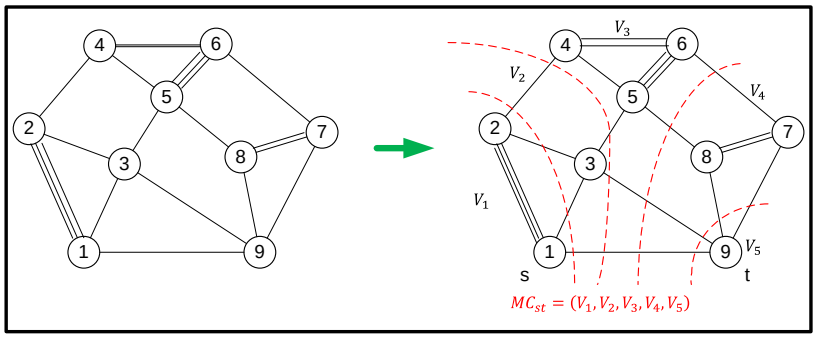

Figure 3.9: Example of st-MC-partition

It is tricky to formulate an algorithm that works for every connected graph. The author did not find any algorithm that was able to do the job[10], and thus started with the formulation of algorithm 3.6. Later it would seem that not all partitions would be found by this algorithm for some graphs and algorithm 3.7 was added as an extension to solve this problem as well. Both these algorithms[11], shown in figure 3.11 and 3.13 respectively, are explained next.

Algorithm 3.6 starts with creating the contraction-list, containing all the vertices of the target graph and copying its contents to a list called original-list (lines 1-4). The former list is used to keep track of how the vertices will be contracted in the succeeding parts of the algorithm, while the latter is used in step 4. The algorithm tries to find all the compatible minimum cuts, by using a min-cut subroutine that outputs the flow value λ_{st} (or minimum cut value), but also 1 cut, namely the one closest to the

[10] Professor Nagamochi was contacted and according to his opinion it is quite difficult to write a program that generates such an *st*-MC-partition. He advised to read [71] to obtain some insight, but also indicated that there was no algorithm in his work for achieving this.

[11] Due to the lack of time, the author was not able to proof that these algorithms are exact.

source *s*. For this subroutine various algorithms can be used, such as a maxflow-mincut algorithm (e.g. based on Goldberg's push-relable algorithm) or Nagamochi's min-cut algorithm [71] (based on maximum adjacency ordering).

In step 2, subroutine-min-cut outputs the first cut, which is stored in the Cut_list (line 5). After that all the vertices in Cut_list are contracted into a single node and the target graph and contraction-list are updated (lines 6-7).

Step 3 is a while loop that runs until the number of vertices in G is equal to 2, because at that stage the last compatible cut (the one closest to t) is already found. This step consists of two major parts, which are both illustrated with examples in figure 3.10. In the first part (lines 9-20) the algorithm searches the node containing vertex s, called *Snode* and generates a neighbor list of this node, denoted as $\Gamma_G(Snode)$. Then it contracts each element of $\Gamma_G(Snode)$ separately with the *Snode* (line 13) and tries to find the next cut (using subroutine-min-cut) closest to *Snode* (and therefore closest to s). Figure 3.10 (b) and (c) show this for each neighbor. Notice that after each contraction the graph is restored and the cut is only stored if it satisfies the conditions of line 17. When it has finished doing this for each neighbor, it starts with the second part of step 3. In this part the algorithm starts using the same reference graph as was the case for each neighbor in the first part (from figure 3.10 (a) producing 3.11 (d)). Basically the second part is identical to the first part except that the contraction is based on the entire neighbor list and that the contracted graph and contraction-list are now stored to be used for the next iteration of the while loop.

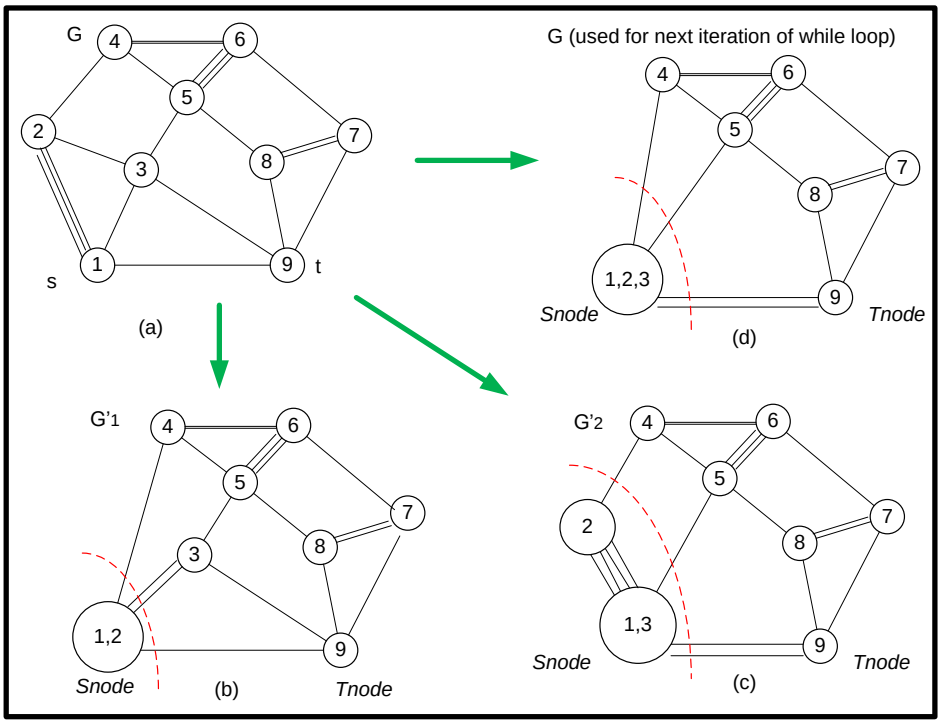

Figure 3.10: Examples to explain algorithm 3.6.

In the last step Cut_list and original-list are used to construct the *st*-MC-partition. First they are added in the *st*-MC-partition as shown in line 21 (see the above partition of figure 3.12, which corresponds to the running example). Then all elements of V_j, already stored in its predecessors $\{V_1, \ldots, V_{j-1}\}$, should be removed from V_j. And finally all empty V_i, if any, should be removed from the *st*-MC-partition (see the resulting partition at the bottom of figure 3.12).

Algorithm 3.6: Construct st-MC-partition

Input: Graph $G^*(V^*, E^*)$ *% Adjacency matrix, Adjacency list or some other representation of G.*
Source node s
Destination node t
Output: st-MC-partition
Cut_list between s and t
Initialization: MC-partition ← empty list
$G(V, E) \leftarrow G^*(V^*, E^*)$
Contraction-list ← empty list

(% Step 1: Construction of a contraction-list)
1. **for** i = 1 to $|V|$
2. **add** i as a singleton set {i} to the Contraction-list;
3. **end**
4. Original-list ← Contraction-list;

(% Step 2: Find the first cut and contract all the nodes on the source side)
5. (λ_{st}, Cut_list) := **Subroutine-min-cut**(G, s, t) *(%Cut_list contains vertices on the s-side of G)*
6. G := $G \backslash (Cut\_list)$; *(% Apply contraction)*
7. **update** Contraction-list ;

(% Step 3: Constructing the Cut_list)
8. **while** $|V| > 2$
9. **find** *Snode* in Contraction-list; *(% find the node containing vertex s in the Contraction-list)*
10. **Construct** a neighbor set $\Gamma_G(Snode)$of *Snode*;
11. **foreach** n $\epsilon\ \Gamma_G(Snode)$ and $(n \neq t)$
12. Temp-list := Contraction-list;
13. G' := $G \backslash \{Snode, n\}$;
14. **update** Temp-list;
15. **find** *Snode* and *Tnode* in Temp-list;
(% find the nodes containing vertices s and t (respectively) in the contraction-list)
16. (flowvalue, Cut) := **Subroutine-min-cut**($G', Snode, Tnode$);
17. **if** $(flowvalue == \lambda_{st})$ **and** $(Cut \notin Cut\_list)$
18. **add** Cut to Cut_list;
19. **end**
20. **end**
21. **find** *Snode* in Contraction-list;
22. **Construct** a neighbor list $\Gamma_G(Snode)$of Snode, where $(t \notin \Gamma_G(Snode))$;
23. G := $G \backslash \{Snode, n \in \Gamma_G(Snode)\}$;
24. **update** Contraction-list;
25. **find** *Snode* and *Tnode* in Contraction-list;
(% find the nodes containing vertices s and t (respectively) in the contraction-list.)
26. (flowvalue, Cut) := **Subroutine-min-cut**($G, Snode, Tnode$)
27. **if** $(flowvalue == \lambda_{st})$ **and** $(Cut \notin Cut\_list)$
28. **add** Cut to Cut_list;
29. **end**
30. **end**

(% Step 4: Construct st-MC-partition out of Cut_list)
31. st-MC--partition = (Cut_list ∪ Original-list);
32. **for** $V_i, V_j \in st$-MC-partition, $(j = i + 1)$
33. **remove** all elements common to both V_i and V_j from V_j;
34. **end**
35. **for** $V_i \in st$-MC-partition
36. **if** $V_i = empty$
37. **remove** V_i;
38. **end**
39. **end**

Figure 3.11: Algorithm st-MC- partition

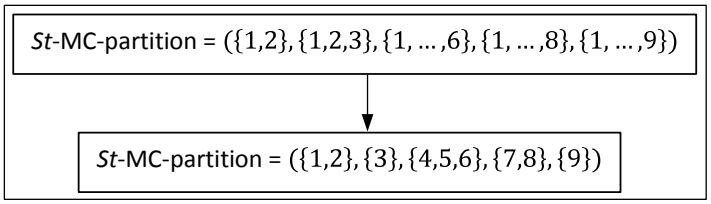

Figure 3.12: Example step 4 of algorithm 3.6.

However, algorithm 3.6 has a serious flaw in that it misses certain partitions in the case that merging the neighbor list $\Gamma_G(Snode)$ (2nd part of step 3) does not produce a new cut, while merging with at least one of the neighbors (1st part of step 3) of the same list does produce such a cut. To rectify this flaw algorithm 3.7 (figure 3.13) serves as an extension.

Time complexity of algorithm 3.6

The time complexity of the 4 steps of algorithm 3.6 are first analyzed/derived and then used to obtain the overall complexity of the algorithm. The worst case running-time function for step 1 is $T_{(1,4)}(n) = c_{(1,4)}n$. In step 2, line 5 makes a call to Subroutine-min-cut, where it is chosen to use the minimum cut algorithm derived by Nagamochi[12] [71] that has complexity $O(|V||E| + |V|^2 log|V|)$. If $m = |E|$, then $T_5(n) = c_5 n(m + nlog(n))$. The contraction in line 6 requires a modification of a representation (usually an adjacency matrix) of G. Because such a representation is 2-dimensional, which requires 2 loops for programming, $T_6(n) = c_6 n^2$. The contraction list can be modeled as a 2-level nested list, which also requires a 2-level nested loop. Because each level has at most $O(n)$ elements (in the worst case), $T_7(n) = c_7 n^2$. Combining $T_5(n), T_6(\mathrm{n})$ and $T_7(n)$ gives $T_{(5,7)}(n) = c_{(5,7)}n(m + nlog(n))$ for step 2. By looking at lines 8-30 in step 3, one can see that lines 12-19 will require relatively large running time, because it runs in the for loop (line 11) that in turn runs in the outer while loop (line 8). Line 16 has the largest order running time, as it calls subroutine-min-cut . For the worst case it is assumed that the contraction in line 23 is such, that in each successive iteration the target graph is smaller by 1 vertex. This yields: $T_{16}(n) = c_{16}\sum_{j=0}^{n-3}(n-j)((n-j)m^* + (n-j)\log(n-j))$ and $m^* = f\big(n - \alpha_j j\big)\big) \leq c'(n)^2$ and α_j is an integer (where the while and for loop correspond to $\sum_{j=0}^{n-3}(n-j)$ in T_{16}). Further manipulation and rewriting results in:
$T_{16}(n) \leq c_{16}\sum_{j=0}^{n-3}(n)^2(m^* + n\log(n)) \leq c_{16}(n-2)n^2(m + nlogn)$.
This means that the complexity of step 3 is represented by $T_{(8,30)}(n) \leq c_{(8,30)}(n-2)n^2(m + nlogn)$, where $c_{(8,30)}$ is chosen large enough. Finally, the complexity of step 4 can be represented by $T_{(31,39)}(n) = c_{(31,39)}n^2$, because lines 32-34 can be programmed by a 2-level nested for loop. Adding the worst case running-time functions (of the 4 steps) and choosing a large enough constant results in $T_{alg\ 3.6}(n) = c_{alg\ 3.6}(n-2)n^2(m + nlogn)$, which means that algorithm 3.6 has complexity $O((|V|-2)|V|^2(|E| + |V|\log|V|))$.

Algorithm 3.7 uses the output *st*-MC-partition and the Cut_list of algorithm 3.6 as its most important input parameters. It zooms into each partition $V_i \in st$-MC-partition and tries to find compatible cuts that might have been missed. Because partitions consisting of a single vertex cannot produce such a

[12] This minimum cut algorithm is one of the faster algorithms that can be used to obtain the minimum cut value λ between source s and a destination t, as well as the minimum cut closest to the source.

cut, they are filtered out by line 2. As an example the graph of figure 3.14 is perfect, because when applying algorithm 3.6 it produces the following incomplete *st*-MC-partition: $MC_{(1,19)} = (\{1\},\{2,\dots,5\},\{6,\dots,12\},\{13\},\{14\}.\{15,\dots,18\},\{19\})$. The 3rd partition should actually be split up into three partitions as shown in figure 3.14. In this subsection we now jump from our running example to this example and we return to the former again in the next subsection.

Algorithm 3.7: Update (s,t)-MC-partition

Input: st-MC-partition = $(V_1, V_2, \dots, V_r)$ *(% as computed by algorithm 3.6)*
Cut_list *(% as computed by algorithm 3.6)*
Graph $G^*(V^*, E^*)$ and $\lambda = \lambda(G^*)$
Node s and node t *(% from algorithm 3.6)*
Output: st-MC-partition *(% updated)*
Cut_list *(% updated)*

1. **for each** $V_i \in st$-MC_partition
2. **if** $|V_i| > 1$ **do**
3. Merge all elements before V_i into a single node: $V_{(1,i-1)}$; *(% skip if $i = 1$)*
4. Merge all elements after V_i into a single node: $V_{(1,i-1)}$; *(% skip if $i = r$)*
5. **if** $V_i = V_1$ **do**
6. $\hat{G} := G^*/\{V_{i+1}, V_{i+2}, \dots, V_r\}$;
7. **elseif** $V_i = V_r$ **do**
8. $\hat{G} := G^*/\{V_1, V_2, \dots, V_{i-1}\}$;
9. **else**
10. $\hat{G} := G^*/\{V_1, V_2, \dots, V_{i-1}\}$;
11. $\hat{G} := \hat{G}/\{V_{i+1}, V_{i+2}, \dots, V_r\}$;
12. **end**
13. **Create** a mapping of vertices of G^* into nodes: map;
14. $Snode$ ← node that contains s in map;
15. $Tnode$ ← node that contains t in map;
16. **Create** a neighbor list of $Snode$ in $\hat{G}$, excluding $Tnode$ from it: $\Gamma(Snode)$;
17. (Cut_list) := **Procedure update Cut_list** (Cut_list, $\hat{G}$, map, $Snode$, $\Gamma(Snode)$)
(% Procedure update Cut_list is a subroutine of this algorithm)
18. **end**
19. **end**
(% Now the st-MC-partition is created out of Cut_list)
20. **Create** a list of all the vertices in G^*: Original-list;
21. **Clear** st-MC--partition
22. st-MC--partition = (Cut_list ∪ Original-list);
23. **for** $V_i, V_j \in st$-MC-partition, $(i < j)$
24. **remove** all elements common to both V_i and V_j from V_j;
25. **end**
26. **for** $V_i \in st$-MC-partition
27. **if** $V_i = empty$
28. **remove** V_i;
29. **end**
30. **end**

Figure 3.13: Algorithm for updating st-MC-partition.

Lines 3–12 indicate that the input graph is submitted to a contraction that contracts all the vertices before and after partition V_i. Partition V_3 is now interesting to use as an example for this discussion

and for this partition the upper left graph of figure 3.16 can demonstrate the result of this code. Note that $V_{(k,l)} = V_k \cup \ldots \cup V_l \ (where\ k < l)$. The map created in line 13 maps the vertices of the input graph G^* to the nodes of this graph in figure 3.16. Using this map $Snode$, $Tnode$ and $\Gamma(Snode)$ can be derived. When this is done, algorithm 3.7 calls upon its subroutine in line 17, which is shown as algorithm 3.8 in figure 3.15. This subroutine produces an updated Cut_list, which now includes each and every compatible cut with respect to *s* and *t*. This complete list is now used to construct a new updated *st*-MC-partition as indicated by lines 20–30. These lines will not be explained here, because it is similar to the code corresponding to step 4 of algorithm 3.6 and therefore the explanation of the latter suffices here as well.

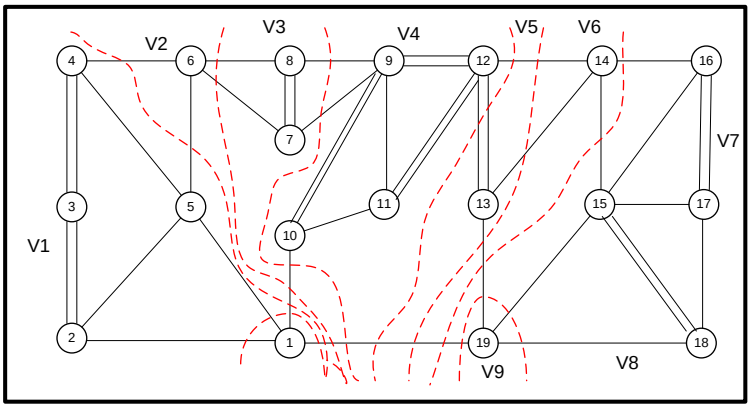

Figure 3.14: Example of a graph that requires algorithm 3.6 and 3.7 to find all cuts.

Steps 1 and 2 of algorithm 3.8 are very similar to step 3 of algorithm 3.6. In both cases $Snode$ is first merged with each element of the neighbor list $\Gamma(Snode)$ and then merged with the whole list. In the case of algorithm 3.8 there is a minor difference in the fact that the contracted graphs are stored at a later stage (step 3 of algorithm 3.8), if a min cut is found. In this case $\hat{G} \coloneqq G'$ (line 29) or $\hat{G} \coloneqq G$" (line 36. This means that the contracted graph is the new target graph in the next recursion. Figure 3.16–3.18 illustrates how the contraction in step 1 and 2 takes place. The mapping is also done similarly to that of algorithm 3.7. The main goal of step 3 of algorithm 3.8 is to stop the recursive process if no new cut is found in V_i or to store the correct contracted graph and update if at least one new cut is found. In the latter case, the subroutine makes a recursive call to itself, such that it can compute other new cuts, if any (or stop if none). In the example of figure 3.16 a new cut is found when vertex 6 is merged with $Snode$. Therefore the lower left graph is the starting graph in the next recursive call. The algorithm continues working in this recursive fashion, until no more new cuts are detected in partition V_i. Figures 3.17 and 3.18 show the results of the 1st and 2nd recursive call for V_3 of the example. When algorithm 3.8 has finished running for each V_i $(i = 1, \ldots, r)$ it returns the updated Cut_list to algorithm 3.7, which processes it as explained before. Information about the structure of the source code of this subsection can be found in figure C5 and C6 of appendix C.

Time complexity of algorithm 3.8

First the complexity of algorithm 3.8 is derived and after that this result is used to do the same for algorithm 3.7. In a similar (but less involved) way as the $T(n)$ function was derived for step 3 of algorithm 3.6, the following is obtained for step 1 of algorithm 3.8: $T_{(1,12)}(n) = c_{(1,12)}(n-1)n(m + nlog(n))$. This time there is a for loop (line 2), which is considered to be upperbounded by $n-1$ and therefore the latter is multiplied with something in the order of the complexity of subroutine-min-cut

to obtain $T_{(1,12)}(n)$. Because the order of $T_{(1,12)}(n)$ is higher than that of any line in step 2, $T_{(1,22)}(n) = c_{(1,22)}(n-1)n(m + nlog(n))$, if $c_{(1,22)}$ is chosen large enough. In step 3 it is shown that the algorithm is recursive, which means that recurrent equation is required, in which something of the order of $O(T_{(1,22)}(n)) = O((\mathrm{n}-1)\mathrm{n}(m + nlog(n))$ appears in each recursive call. The worst case would be if during each recursive call, the target graph would become smaller by one vertex (after contraction by line 29 or 36) and the obtained result would be the input for the next recursion. So the number of vertices would decrease as follows: $n \to n-1 \to \cdots \to 1$. (The recursion would surely stop if $n = 1$, because a trivial graph cannot produce a min cut (lines 23-25)). By combining the results (and neglecting subscripts) of the three steps, the following equation is achieved: $T(n) = T(n-1) + c'(\mathrm{n}-1)\mathrm{n}(m + nlog(n)$. To simplify this equation, $m + nlog(n) \leq c"n^2$ is integrated in the above and after some simplification (for a large number of vertices $(n-1) \approx n$), the following is obtained: $T(n) = T(n-1) + cn^4$.

Substitution gives the following:

$T(n-1) = T(n-2) + c(n-1)^4$.

$T(n-2) = T(n-3) + c(n-2)^4$.

⋮

$T(2) = T(1) + c(2)^4$.

$T(1) = O(1)$, because no minimum cut can be found in a trivial graph. Therefore no recursive call can be invoked anymore (lines 23-25) and the recursion stops. By applying some math to the above, the recursive equation can be written as: $T(n) = c\sum_{j=0}^{n-2}(n-j)^4$.

From this we find that $T(n)$ is upperbounded as follows: $T(n) = c\sum_{j=0}^{n-2}(n-j)^4 \leq c(n-1)n^4$. This means that algorithm 3.8 has complexity $O(|V|^5)$.

Time complexity of algorithm 3.7

The recursive subroutine Update_Cut_list in line 17 of algorithm 3.7 has the highest complexity of all the other lines of its pseudo code. Because this subroutine is called r times (r is the number of partitions in *st*-MC-partition) and because $r \leq |V|$, the complexity of algorithm 3.7 is $O(|V|^6)$.

Algorithm 3.8: Update Cut_list

Input: Graph $\hat{G}(V,E)$ and map
Cut_list
$\lambda = \lambda(G^*)$
Node s and node t
$Snode$ and $\Gamma(Snode)$

Output: Graph $\hat{G}(V,E)$ and map *(% updated)*

(%Step1: Contract each neighbor of Snode and store all new min cuts)

Cut_list *(% updated)*
$Snode$ and $\Gamma(Snode)$ *(% updated)*

1. **Create** j_list and Cut_list_1; *(% These are empty lists)*
2. **for each** $j \in \Gamma(Snode)$ **do**
3. $G'_j := \hat{G}/\{Snode, j\}$;
4. **Create** map'_j; *(%mapping the vertices of G^* to the nodes of G'_j)*
5. $Snode'_j$ ← location of s in map'_j;
6. $Tnode'_j$ ← location of t in map'_j;
7. (flowvalue, Cut) := **subroutine-min-cut**($G'_j, Snode'_j, Tnode'_j$)
8. **if** (flowvalue = λ) and (Cut $\notin$ Cut_list)
9. **add** Cut to Cut_list_1; **add** j to j_list;
10. **end**
11. **Create** a neighbor list of $Snode'_j$, excluding $Tnode'_j$: $\Gamma'(Snode'_j)$;
12. **end**

(%Step2: Contract the whole neighbor list of Snode and store a new min cut, if any)

13. $G'' := \hat{G}/\{Snode, j\}$;
14. **Create** $\boldsymbol{map}''$; *(%mapping the vertices of G^* to the nodes of G'')*
15. $Snode''$ ← location of s in map'';
16. $Tnode''$ ← location of t in map'';
17. (flowvalue, Cut) := **subroutine-min-cut**($G'', Snode'', Tnode''$)
18. **Create** Cut_list_2;
19. **if** (flowvalue = λ) and (Cut $\notin$ Cut_list)
20. **add** Cut to Cut_list_2;
21. **end**
22. **Create** a neighbor list of $Snode''$, excluding $Tnode''$: $\Gamma''(Snode'')$;

(% Step 3: Add cuts to Cut_list and start recursive call if required)

23. **if** (Cut_list_1 = empty) **and** (Cut_list_2 = empty)
24. **return** Cut_list
25. **end**
27. **if** (Cut_list_1 ≠ ∅) **and** (Cut_list_2 = ∅)
28. **for each** $k \notin j\_list$ **do**
29. $\hat{G} := G'_k$; $map := map'_k$;
30. $Snode := Snode'_k$; $\Gamma(Snode) := \Gamma'_k(Snode'_k)$;
31. **Move** k[th] element from Cut_list_1 to Cut_list;
32. (Cut_list, $\hat{G}$, map, $Snode$, $\Gamma(Snode)$)) := **Procedure update Cut_list** (Cut_list, $\hat{G}$, map, $Snode$, $\Gamma(Snode)$) *(%This is a recursive call)*
33. **end**
34. **end**
35. **if** Cut_list_2 ≠ ∅)
36. $\hat{G} := G''$; $map := map''$;
37. $Snode := Snode''$; $\Gamma(Snode) := \Gamma''(Snode'')$;
38. **Move** all elements from Cut_list_1 (if any) and the element of Cut_list_2 to Cut_list;
39. (Cut_list, $\hat{G}$, map, $Snode$, $\Gamma(Snode)$)) := **Procedure update Cut_list** (Cut_list, $\hat{G}$, map, $Snode$, $\Gamma(Snode)$) *(%This is a recursive call)*
40. **end**

Figure 3.15: Algorithm update Cut_list.

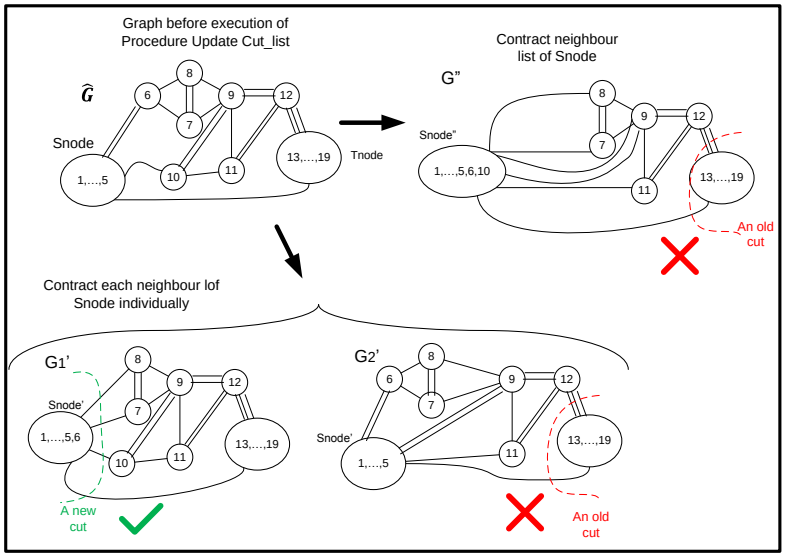

Figure 3.16: Example of the process of updating the Cut_list (1).

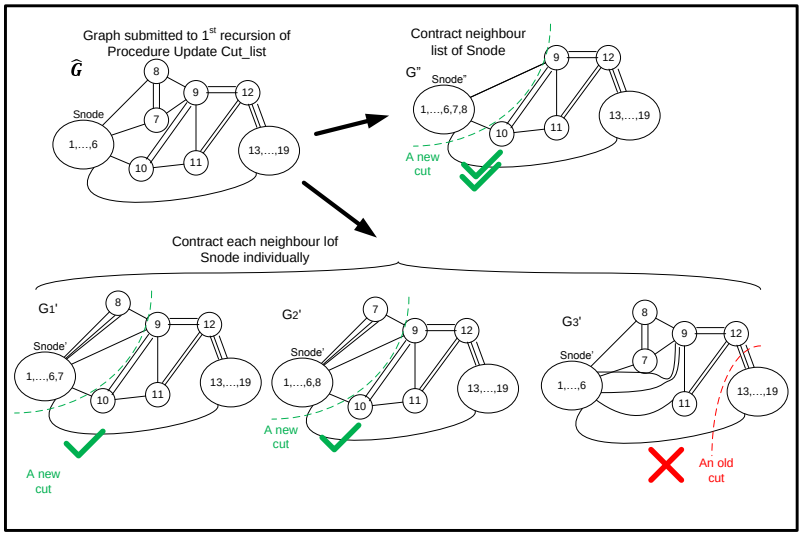

Figure 3.17: Example of the process of updating the Cut_list (2).

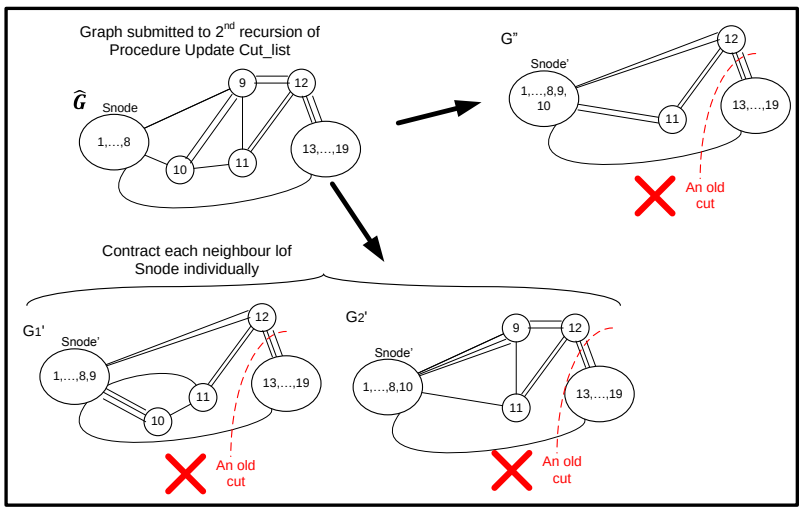

Figure 3.18: Example of the process of updating the Cut_list (3).

3.3.3 Constructing the *st*-cactus-representation

One of the most crucial parts of constructing a cactus representation, is the construction of an *st*-cactus-representation $(R_{(s,t)}, \varphi_{(s,t)})$, which represents all compatible cuts separating s and t. Algorithm 3.9 (in figure 3.22, 3.23 and 3.24) is able to construct such a representation in 5 steps. Steps 2 and 5 were already algorithmically available in [71]. The other steps and the integration of all 5 steps are new. The explanation will be done with the aid of the example, which was called the running example in the previous subsection.

Step 1 creates a contracted graph G_{st_MC} out of the original graph G^*, where all the vertices in the same partition $V_i \in st$-MC-partition are contracted into a single node and this is done for each and every partition. If the graph in figure 3.9 were to be treated as G^*, then G_{st_MC}, with $s = 1$ and $t = 9$, would become the graph shown in the next figure.

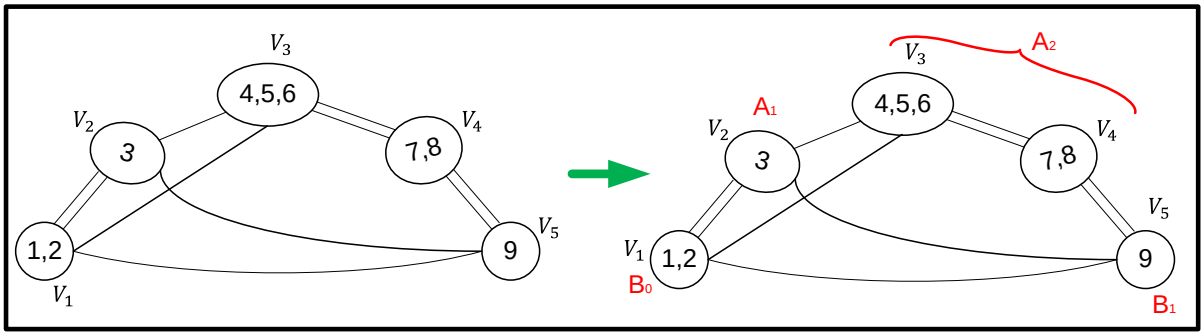

Figure 3.19: Example of contracted graph according to st-MC-partition.

Steps 2 to 5 are only invoked if the target graph consists of more than 2 vertices. Notice that this is an interesting situation, because if the target graph consists of only 1 or 2 vertices the *st*-cactus-representation is a trivial cactus or two nodes connected by 2 edges, respectively. In the case of 1 vertex, Nagamochi's algorithm will never invoke algorithm 3.9, due to the if statement in line 1 of algorithm 3.4. In the case of 2 vertices, $R_{(s,t)}$ consists of only 2 nodes connected by 2 links. Notice that $\varphi_{(s,t)} = st$-MC-partition. Now the explanation of the more interesting case, steps 2-5, will be done with the help of the running example.

Step 2 creates segments of the vertices of G_{st_MC} that will later on be used to construct circular-minimum-cut partitions CMC_{Ak}[13] and minimum-cut partitions MC_{Bj}. These partitions are then used to create cycles and chains, respectively, which are the building blocks of $(R_{(s,t)}, \varphi_{(s,t)})$. But as step 2 is only concerned with the segmentation it is now relevant to explain how the segments are derived and which segments generate the chains (cycles). Chain segments are derived from partitions that are included in $B = \{V_1, V_r\} \cup \{V_i | d(V_i, \overline{V_i}) > \lambda, 1 < i < r\}$, while cycle segments are derived form the partitions included in $A = \{V_i | d(V_i, \overline{V_i}) = \lambda, 1 < i < r\}$. Each element B_j of B can be treated as a chain segment. Out of the elements of A the cycle segments can be constructed as follows: $A_k = V_{(a_k,l)}$ if $V_l \in A$ and $d\left(V_{(a_k,l)}, \overline{V_{(a_k,l)}}\right) = \lambda, (a_k, a_k + 1, \dots, b_k)$ holds [71]. a_k is equal to the first

[13] A circular-minimum-cut partition is a partition that results in a cycle graph, as will be shown later in this section. For detailed information it is recommended to read section 5.1 of [71].

and b_k is equal to the last V_i in a segment A_k. The right graph of figure 3.19 shows the segments of the running example. Step 2 stores the chain and cycle segments in B-list and A-list, respectively. Lines 6-15 are concerned with constructing the cycle segments, while the remaining lines in step 2 of the algorithm are devoted for constructing the chain segments.

Step 3 is concerned with constructing the chains out of the B_j segments. But before such a chain graph can be constructed, it is first necessary to construct MC_{Bj} for each B_j segment (lines 27-61). This is done by storing all the vertices of all the V_i appearing before (after) the partition corresponding to B_j, say V_j, into a single sub-list and placing it in front of (behind) V_j in MC_{Bj}. If there is no V_i before (after) V_j, no sub-list is placed before (after) V_j. After constructing MC_{Bj}, this minimum cut partition is used to construct the chains (lines 62-68). If B_j is the first or last element of MC_{Bj}, then a chain consisting of 2 nodes is constructed. Otherwise, a chain consisting of 3 nodes is constructed. In all cases the MC_{Bj} serves as a mapping for the respective chains and each node is connected with 2 links to each neighbor in the chain. This means that a chain representation $(R_{Bj}^{chain}, \varphi_{Bj}^{chain})$ is constructed for each MC_{Bj}. The chains on the right in figure 3.20 demonstrate this for the running example.

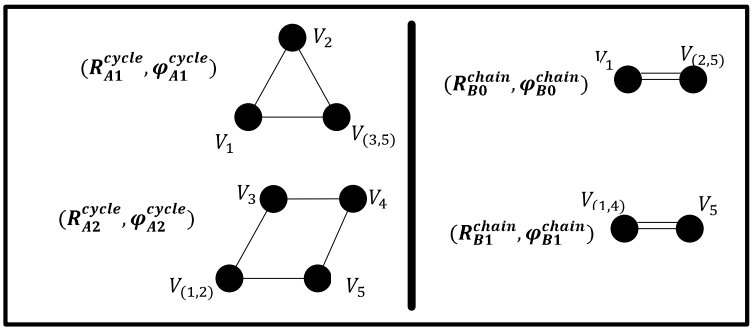

Figure 3.20: Chains and cycles of running examples.

Step 4 first constructs the circular-minimum-cut partitions CMC_{Ak} (lines 68-85). In this phase the values a_k and b_k are also computed and stored in two separate vectors (lines 74-76). A segment A_k does never include V_1 or V_r and is therefore always between other partitions of *st*-MC-partition. The vertices of each V_i before (after) each cycle segment A_k are stored into a sub list and placed as the first (last) element in CMC_{Ak} for each A_k. From each CMC_{Ak}, a cycle is constructed where each node is connected by one link to each of its two neighbors (lines 86-89). Step 4 generates a cycle representation $(R_{Ak}^{cycle}, \varphi_{Ak}^{cycle})$ for each CMC_{Ak} as step 3 did for the situation regarding the chains. The cycles on the left in figure 3.20 demonstrate the result of step 4 for the running example.

Step 5 finaly merges the chains and cycles together in the correct order to construct $(R_{(s,t)}, \varphi_{(s,t)})$. The chain corresponding to B_0 is the one that the construction was started with (lines 91 and 92), as it contains vertex *s*. This chain is initially stored as $(R_{(s,t)}, \varphi_{(s,t)})$. Then lines 95 and 96 are used for merging/unifying the first cycle representation $(R_{A1}^{cycle}, \varphi_{A1}^{cycle})$ with the last stored $(R_{(s,t)}, \varphi_{(s,t)})$ and the result in turn is stored as $(R_{(s,t)}, \varphi_{(s,t)})$. Then comes the tricky part where during the same iteration a chain will be combined with the last $(R_{(s,t)}, \varphi_{(s,t)})$ if the condition in line 97 is satisfied. If not, the next iteration of the for loop starts without the addition of a chain in the current iteration. Figure 3.21 exemplifies how the chains and cycles are combined to construct the *st*-cactus-representation. The Matlab code structure is shown in figure C7 of appendix C.

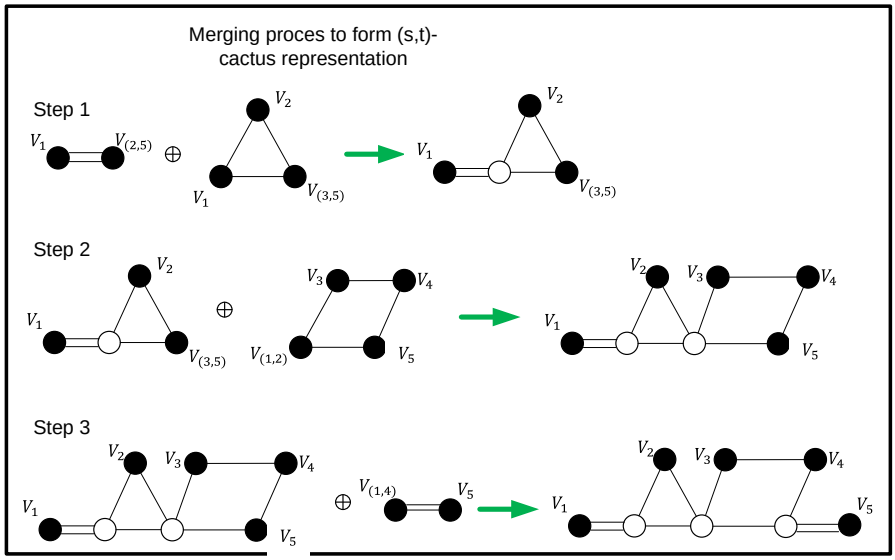

Figure 3.21: Example of constructing st-cactus-representation out of chains and cycles.

Algorithm 3.9: Constructing st-cactus-representation

Input: Graph $G^*(V^*, E^*)$ *(% Adjacency matrix, Adjacency list or some other representation of G)*
$\lambda := \lambda(G^*)$
st-MC-partition *(% of size r)*
Output: $(R_{(s,t)}, \varphi_{(s,t)})$

(% Step 1: Merge all the vertices of G^* into nodes according to the st-MC-partition)
$G_{(s,t)-MC} := G^*$;
for each $V_i \in$ st-MC-partition **do**
  $G_{(s,t)\_MC} := G_{(s,t)-MC} \backslash \{V_i\}$;
end
if $\left|V\left(G_{(s,t)\_MC}\right)\right| > 2$
(% Step 2: Segment the obtained graph, such that it can be converted into chains or cycles later on)
  $a := 2;\ l := 2; delta := 0; k := 0;$ *(%initialization values)*
  create A-list and B-list
  while $a < r$ **do**
    while $d\left(V_{(a,l)}, \overline{V_{(a,l)}}\right) = \lambda$ **do** $delta := 1; l := l + 1;$ **end**
    if $delta := 1$ **then**
      $delta := 0; k := k + 1; A_k := V_{(a,l-1)}; l := l - 1;$
      add A_k to A-list;
    end
    $a := l + 1; l := l + 1;$
  end
  $B_0 := V_1;$
  add B_0 to B-list;
  $j := 1; b := 2;$
  while $b < r - 1$
    if $d(V_b, \overline{V_b}) > \lambda$
      $j := j + 1; B_j := V_b;$ **add** B_j to B-list;
    end
    $b := b + 1;$
  end
  $B_q := V_r;$
  add B_q to B-list; *(% B-list = $(B_0, B_1, \ldots, B_q)$)*
(% Step 3: Construct the chains corresponding to the segments in B-list)
  create Chain-partitions and chain-cacti;
  for each $B_i \in B$-list **do**

Figure 3.22: Algorithm for constructing st-cactus-representation (part 1).

```
29.                     create MC_{B_j}; (% an empty list)
30.                     create before-list, between-list and after list;
31.                     if B_j = B_0
32.                         for each V_i ∈ st-MC-partition do
33.                             if V_i = B_0
34.                                 add all elements of V_i to before-list;
35.                             else
36.                                 add all elements of V_i to after-list;
37.                             end
38.                             add before-list to MC_{B_j}; add after-list to MC_{B_j};
39.                         end
40.                     elseif B_j = B_q
41.                         for each V_i ∈ st-MC-partition do
42.                             if V_i = B_q
43.                                 add all elements of V_i to after-list;
44.                             else
45.                                 add all elements of V_i to before-list;
46.                             end
47.                             add before-list to MC_{B_j}; add after-list to MC_{B_j};
48.                         end
49.                     else
50.                         for each V_i ∈ st-MC-partition do
51                              if V_i appears before B_j
52.                                 add all elements of V_i to before-list;
53.                             elseif V_i = B_j
54.                                 add all elements of V_i to between-list;
55.                             else
56.                                 add all elements of V_i to after-list;
57.                             end
58.                             add before-list to MC_{B_j}; add between-list to MC_{B_j}; add after-list to MC_{B_j};
59.                         end
60.                     end
61.                     add MC_{B_j} to chain-partitions
                        (% now we construct the graphs of the chain-partitions)
62.                     If B_j ∈ {B_0, B_q}
63.                         Construct R_j^chain consisting of 2 nodes, each one connected by 2 links to its neighbor;
64.                     else
65.                         Construct R_j^chain consisting of 3 nodes, each one connected by 2 links to its neighbor(s);
66.                     end
67.                     add R_j^chain to chain-graphs;
68.             end
(% Step 4: Construct the cycles corresponding to the segments in A-list)
69.             create Cycle-partitions and cycle-cacti; (% empty at initialization)
70.             create a_vector_k and b_vector_k; (% empty at initialization)
71.             for each A_k ∈ A-list do
72.                 create CMC_{A_k}; (% an empty list)
73.                 index-small := r; index-large := 1;
74.                 if k ≤ index-small then index-small:=k; end
75.                 if k ≥ index-large then index-large:=k; end
76.                 a_vector_k := index-small; b_vector_k := index-large;
77.                 create before-list and after list; (% empty at initialization)
78.                 for each V_i ∈ st-MC-partition do
79.                     if V_i appears before A_k
80.                         add all elements of V_i to before-list;
```

Figure 3.23: Algorithm for constructing st-cactus-representation (part 2).

```
81.                                 elseif V_i appears after A_k
82.                                     add all elements of V_i to after-list;
83.                                 end
84.                             end
85.                             add before-list, A_k and after-list in ordered fashion to CMC_{A_k};
                                (% now we construct the graphs of the chain-partitions)
86.                             Construct R_k^{cycle} consisting of size(CMC_{A_k}) nodes, each one connected by 1 link to each neighbor;
87.                             add R_k^{cycle} to cycle-cacti;
88.                             add CMC_{A_k} to cycle-partitions
89.                         end
(% Step 5: Construct R_(s,t) and φ_(s,t))
                            (% Initialization)
90.                         create a list of all the vertices of G*: vertex-list;
91.                         R_(s,t) := R_0^{chain}; (% from chain-graphs)
92.                         φ_(s,t) := MC_{B_0}; (% from chain-partitions)
93.                         a_vector_{k+1} := inf; j := 1;
94.                         for each A_k ∈ A-list (% k is the index of A_k)
95.                             R_(s,t) := merge(R_(s,t) and R_k^{cycle});
96.                             φ_(s,t) := unify(φ_(s,t) and CMC_{A_k});
97.                             if (b_vector_k + 1) ≠ a_vector_{k+1}
98.                                 R_(s,t) := merge(R_(s,t) and R_j^{chain});
99.                                 φ_(s,t) := unify(φ_(s,t) and MC_{b_j});
100.                                j := j + 1;
101.                            end
102.                        end
103.    elseif |V(G_(s,t)_MC)| = 2
104.                        R_(s,t) is a chain consisting of 2 nodes, each one connected by 2 links to its neighbor;
105.                        φ_(s,t) := st-MC-partition;
106.    end
```

Figure 3.24: Algorithm for constructing st-cactus-representation (part 3).

Time complexity of algorithm 3.9

Only the lines of the pseudo code of algorithm 3.9, which require the largest processing time (for significantly large n) are considered. The running-time function is again derived for each step and then the corresponding results are added to compute the overall complexity order. For step 1, $T_{(1,5)}(n) = c_{(1,5)}rn^2$, because the contraction in line 3 requires a 2-level nested for loop, which should run for all r partitions in *st*-MC-partition. For line 8 (step 2) $T_8(n) = \sum_{a=2}^{r} t_a$, while $T_9(n) = c_9(\sum_{a=2}^{r} t_a)(r-2)$ applies for line 9. The last factor in the latter is due to the fact that an A_k-segment $V_{(a,l)}$ cannot be larger than $(r-2)$. This leads to $T_{(6,18)} = c'_{(6,18)}(r-2)(\sum_{a=2}^{r} t_a) \leq c_{(6,18)}(r-2)(r-1)$ for line 6-18. Similarly $T_{(19,24)} = c'_{(19,24)}\left(\sum_{b=2}^{r-1} t_b\right) \leq c_{(19,24)}(r-2)$. Adding the upperbounds of $T_{(6,18)}$and $T_{(19,24)}$results in $T_{(6,26)} = c_{(6,26)}(r-2)r$ for step 2. Line 8 in step 3, has running-time function $T_{28}(n) = c_{28}nr^{-1}$, because there can be no more than (nr^{-1}) B_j segments. This yields $T_{(27,61)} = c_{(27,61)}r(nr^{-1}) = c_{(27,61)}n$, which results in $T_{(27,68)} = c_{(27,68)}n$ for step 3. To obtain the function for step 4, we first state that $T_{71}(n) = c_{71}(r-2)$, $T_{78}(n) = c_{78}r(r-2)$ and $T_{86}(n) = c_{86}(r-2)^3$. (It takes a 2-level nested for loop to construct r_k^{cycle} (line 86)). This results in $T_{(69,89)}(n) = c_{(69,89)}(r-2)^3$ for step 4. In step 5, the construction of both $R_{(s,t)}$ and $\varphi_{(s,t)}$ (lines 94-95) require 2-level nested for loops, each of size r at most. When taking the for loop of line 94 into account, this results in $T_{(90,102)} = c_{(90,102)}r^2(r-2)$. For the remainder of the algorithm, $T_{(103,106)} = n^2c_{(103,106)} = 4c_{(103,106)}$ holds. Because this last part is insignificant, the worst case

complexity is derived by adding the complexities of step 1 to 5. When using $r \leq n$ as an upperbound the worst case running-time function of algorithm 3.9 becomes $T_{alg\ 3.9} = c_{alg\ 3.9} n^3$, This mean that the complexity is $O(|V|^3)$.

3.3.4 Merging multiple cacti

Algorithm 3.10 shows all the steps that are required for merging $(R_{(s,t)}, \varphi_{(s,t)})$, representing all the compatible cuts C_{comp}, with the cacti (R_i, φ_i) (where $i = 1, \dots, r$) that together represent all the indivisible cuts C_{ind}.

Algorithm 3.10: Merger of cacti

Input: $R_{(s,t)}$ and $\varphi_{(s,t)}$
R_i and φ_i, $(i = 1,2, \dots, r)$ *(% r is the number of elements of MC_{st})*
A complete list of the vertices of G^*: vertex-list.
Output: R and φ

1. $R^{"} \coloneqq R_{(s,t)}$; *(% Initialization)*
2. $\varphi^{"} \coloneqq \varphi_{(s,t)}$; *(% Initialization)*
3. **for** $i = 1$ **to** r **do**
4. **if** $|W(R_i)| > 2$ **or** $(|W(R_i)| = 2$ **and** the cut separating the last 2 nodes is not old)
5. **for each** $n^{"} \in \varphi^{"}$ **do**
6. **for each** $n' \in \varphi_i$ **do**
7. union-list := $n^{"} \cup n'$;
8. **if** union-list = vertex-list
9. $z^{"} = n^{"}; z = n'$; *(% these are the nodes to be joined together)*
10. **end**
11. **end**
12. **end**
13. **Update** $\varphi^{"}(z^{"})$, such that $\varphi^{"-1}(z^{"}) \coloneqq \varphi^{"-1}(z^{"}) \cap \varphi_i^{-1}(z)$ and that the mapping of the nodes $W_i(R_i) - z$ is added appropriately to the map $\varphi^{"-1}(z^{"})$;
14. $R^{"} \coloneqq merge(R^{"} and\ R_i)$, such that z is removed and that its incident links are connected directly to $z^{"}$;
15. **end**
16. **end**
17. $R \coloneqq R^{"}$;
18. $\varphi \coloneqq \varphi^{"}$;

Figure 3.25: Merging cacti to form the cactus representation.

Figure 3.26 exemplifies the contracted graphs G_i on the left, produced by line 12 of algorithm 3.4. The graphs on the right in this figure illustrate the cacti (R_i, φ_i), produced by line 14. It is also shown that (R_2, φ_2) and (R_5, φ_5) are trivial cacti. Remember that they are trivial because the final cut separating the last two nodes in R_i is already marked old.

Algorithm 3.10 starts with the merging process by copying $(R_{(s,t)}, \varphi_{(s,t)})$ to $(R^{"}, \varphi^{"})$, as is done by lines 1 and 2. The latter is used further in the algorithm as a starting representation to which all the non-trivial cacti corresponding to indivisible cuts are added. Lines 3 and 4 together are used to select

all the non-trivial cacti for this merger. Lines 5-12 find the two nodes $z^{"} \in W^{"}(R^{"})$ and $z \in W_i(R_i)$ that need to be contracted as a new node in the merged graph. By repeatedly trying out the unification of one element of $\varphi^{"}$ and one of φ_i, one particular combination will be the same as the vertex-list, which is a list containing all the vertices of the target graph G^*. When these two so called junction nodes are found, the vertex to node map $\varphi^{"}$ is updated as described in line 13. In line 14 it is shown how the cactus graph $R^{"}$ is merged.

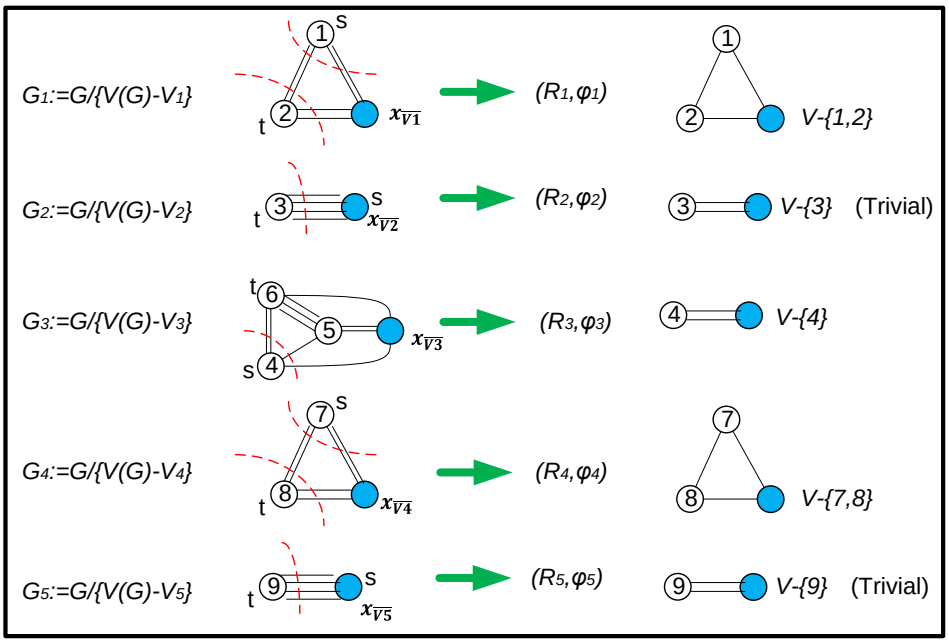

Figure 3.26: Examples of cacti to be merged with st-cactus-representation.

Figure 3.27 illustrates how the merging process, described in algorithm 3.10 takes place for the running example. Notice that the graph in the center of 3.27 (a) is the final graph of figure 3.21.

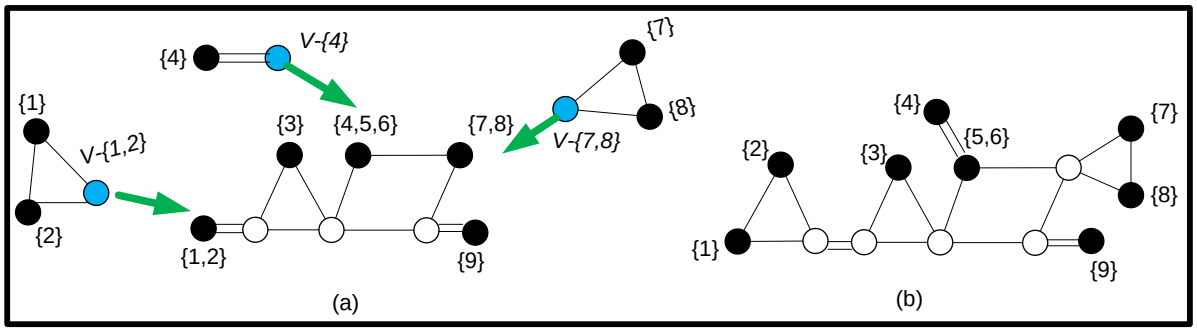

Figure 3.27: Example of merging cacti.

Time complexity of algorithm 3.10

The following functions are derived similarly as for the previous algorithms. The order is again determined by the amount of loops required to write a program according to the pseudo code of algorithm 3.10:

1. $T_3 = c_3 r$.
2. $T_5 = c_5 rn$, because the number of nodes in $R_{(s,t)}$ is $O(n = |V|)$.
3. $T_6 = T_5(c' nr^{-1}) = c_6 rn(nr^{-1}) = c_6 n^2$, because the number of nodes in R_i is $O(nr^{-1})$.

4. $T_{13} = T_3(c"nr^{-1}) = c_{13}n$, because the number of vertices mapped to node z is $O(nr^{-1})$.
5. $T_{14} = T_3(c^*n^2) = c_{14}rn^2$. It requires a 2-level nested for loop, each of size $O(n)$ to perform line 14.

Adding these equations and choosing a large enough constant $c_{(1,19)}$ results in $T_{(1,19)} = c_{(1,19)}rn^2$. Because $r \leq n$, the resulting complexity for algorithm 3.10 is $O(|V|^3)$.

3.3.5 Converting a cactus representation to a CNC cactus representation

For converting a cactus representation into a CNC representation algorithm 3.11 (figure 3.61) should be invoked. A CNC representation does not contain empty 3-junction nodes nor does it contain an empty 2-junction node belonging to a 2-cycle. By applying algorithm 3.11, just after algorithm 3.10 all the empty 3-junction nodes are first removed. Afterwards the empty 2-junction nodes of the 2-cycles are removed and the CNC representation is achieved.

Algorithm 3.11: Construct CNC representation

Input: $R'$and φ'
A complete list of the vertices of G^*: vertex-list.
Output: $R'$and φ'

```
1.  for each w ∈ φ' do
2.      if w is an empty node (% if n does not map to any vertex)
3.          add n to empty-list;
4.      end
5.  end
6.  for each k ∈ empty-list
7.      if k is an empty 3-junction node
8.          modify R' and φ' according to 3-cycle insertion;
9.      end
10. end
11. for each w ∈ φ' do
12.     if w is an empty node (% if n does not map to any vertex)
13.         add n to empty-list;
14.     end
15. end
16. for each k ∈ empty-list
17.     if k is a 2-junction node and at least one of the cycles containing k is a 2-cycle
18.         R' := R'\{k,2-cycle-neighbour(k)};
19.         move the contents of the 2-cycle neighbor of φ' to φ'(k);
20.         remove the 2-cycle neighbor of φ';
21.     end
22. end
```

Figure 3.28: Algorithm for converting to CNC representation.

Algorithm 3.11 initially starts to construct a list with all the empty nodes (lines 1-5). Then it looks if an element of this list is an empty 3-junction node. If so it applies a 3-cycle insertion and updates (R', φ') accordingly (lines 6-10). A 3-cycle insertion[14] is a process where an empty 3-junction node is replaced by a 3-cycle of empty nodes as illustrated in the next figure.

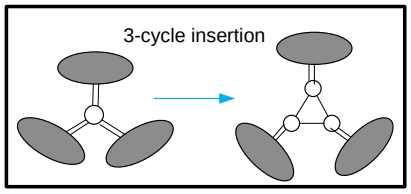

Figure 3.29: Example of 3-cycle insertion.

Because the graph is modified, the empty nodes in the new (R', φ') are stored in a new list (lines 11-15). In the remainder of the algorithm, all empty 2-junction nodes in a 2-cycle are removed to obtain the CNC representation. Figure 3.30 illustrates how this algorithm works for our running example.

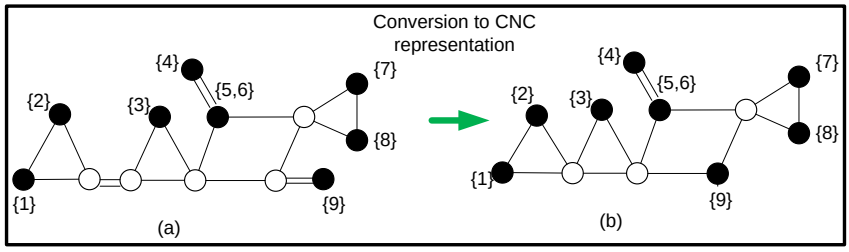

Figure 3.30: Example of constructing a CNC representation.

Time complexity of algorithm 3.11

For the first 5 lines of algorithm 3.11, $T_{(1,5)}(n) = c_{(1,5)}n$, because the number of nodes in the cactus R' is $O(n)$. In [71] it is shown that the number of empty nodes is also $O(n)$, which results in $T_6(n) = c_6 n$. If there is a 3-junction node in R', it should satisfy the following properties:

1. The degree of the node should be 6.
2. It should have at least 3 neighbors.
3. Each neighbor should be connected by 1 or 2 links to the 3-junction node.

To identify the 3-junction node (line 7), a for loop of $O(n)$ is required. By doing this for all 3-junction nodes and using the loop of line 6 to iterate through all these nodes, the worst case running-time function for line 7 becomes $T_7(n) = c_7 n^2$. Once identified, a 3-cycle insertion should take place (line 8). The modification of both R' and φ' requires a 2-level nested for loop. This results in $T_8(n) = c_7 n^3$ that can be extended to $T_{(6,10)}(n) = c_{(6,10)}n^3$, because line 8 has the highest order in the range 6-10. $T_{(11,15)}(n) = c_{(11,15)}n$ (similarly as the first 5 lines). Because both line 19 and 20 require a 2-level nested loop for the programming (excluding the for loop of line 16), the highest complexity order of line 16-22 is $O(n^3)$. This leads to $T_{(16,22)}(n) = c_{(16,22)}n^3$. Adding $T_{(1,5)}(n)$, $T_{(6,10)}(n)$, $T_{(11,15)}(n)$ and $T_{(16,22)}(n)$ results in $T_{alg\ 3.11} = c_{alg\ 3.11}n^3$. Thus, the complexity of the algorithm is $O(|V|^3)$.

[14] 3-cycle insertion in line 8 is actually a big step that can be programmed in various ways, depending in what format R' is represented and submitted to the code.

3.4 Increasing the edge-connectivity

This section introduces a technique to increase a graph's robustness, which focuses more on the weakly connected parts of the network, rather than on the complete topology of the network[15]. In this section an edge-augmentation algorithm is treated that focuses on optimally augmenting the edge-connectivity of the PS mobile core network. For increasing the edge-connectivity of a graph, it is required to solve the edge-augmentation problem for a graph $G(V, E)$. The edge-augmentation problem is defined as to find the smallest set of edges to be added to G, such that its edge-connectivity can be increased by an integer value δ [73]. Thus, G will become $(k + \delta)$-edge-connected after applying the edge-augmentation procedure.

Algorithm 3.12 (devised by D. Naor, D. Gusfield and C. Martel [73]) is used for augmenting core PoP ASD/RT and the PS mobile core network[16]. To understand this algorithm it is relevant to understand some processes which can be seen as subroutines of the algorithm. These processes are explained next and afterwards the algorithm is treated. For an elaborate description it is recommended to resort to the literature [8, 12 and 73].

The first process to be clarified is the modified DFS algorithm, consisting of two stages. The 1st stage defines different colors for the different cycles of R. The 2nd stage is characterized by a DFS traversal, which starts at an arbitrary node and obeys the following rule: If a node is visited for the first time via a link, which is part of a cycle (colored for example with red), then all other links incident to this node should be traversed, before traversing the other (red) link incident to that node. If the cactus is acyclic, the modified DFS procedure reduces to the standard DFS algorithm.

Furthermore, it is necessary to say something about the edge demand function $\Phi(P)$, where *P* may be any partition of the set *V* of vertices into disjoint subsets $P_1, P_2, \ldots\ldots, P_x$. This function is used to compute how many edges are required to achieve $(k + \delta)$-connectivity and is defined as:

$$\Phi(P) = \sum_{i=1}^{x} \max\{0, (k + \delta) - d(P_i)\} \qquad (3.2)$$

Equation (3.2)[17] takes the sum of the number of edges to be added to each P_i. If for subset P_i, $d(P_i) < (k + \delta)$, then at least $(k + \delta) - d(P_i)$ edges need to be added between P_i and another subset P_j (that also requires at least 1 edge, because $d(P_j) < (k + \delta)$). Since the subsets are disjoint, each edge satisfies at most 2 requirements. This means that over all partitions *P*, at least $max_p\{\lceil \Phi(P)/2 \rceil\}$ edges need to be added to a graph to make it $(k + \delta)$-connected [73].

Finally it is important to know how the Extreme Sets Tree (EST) is constructed. The definition of EST states that a set $U \subset V$ is q-extreme if and only if $d(U) = q$ is strictly smaller than each of its proper subsets. This definition states for any $W \subset U$ that $d(W) > q$, given that $d(U) = q$. Lemma 4.2 in

[15] When increasing $a(G)$ to increase the robustness, the focus is on the whole graph.

[16] Results involving other subnets are in the KPN deliverable.

[17] In equation 3.2 *k* is the edge-connectivity before edge augmentation, δ is the amount to increase the edge-connectivity and $d(P_i)$ is the degree of P_i.

[73] states that if X is q-extreme, $Y \neq X$ is l-extreme and $q \leq l$, then either $Y \subset X$ or X and Y are disjoint. This can be represented by the EST, where every leaf corresponds to a vertex $v \in V$, and the root to the entire set $V(G)$. Every other node in the tree corresponds to an extreme set. The construction of the EST is not trivial. Figure 3.31 is used to exemplify the EST construction. For any node x other than the root *r* of the Extreme Sets Tree, the edge demand of node *x* is:

$$\Phi(x) = \max\{0, (k+\delta) - d(X), \sum_{y\ child\ of\ x} \Phi(y)\} \qquad (3.3)$$

The edge demand of the root *r* is:

$$\Phi(r) = \sum_{y\ child\ of\ r} \Phi(y) \qquad (3.4)$$

The edge demand of the extreme set (ES) is equal to that of the root of the EST ($\Phi(ES) = \Phi(r)$). Theorem 4.4 in [73] shows that the algorithm given above is optimal. The point of this theorem is that only $\lceil \Phi(ES)/2 \rceil$ new edges are needed for increasing the edge-connectivity by δ. Remember that this was the minimum number of edges required for the augmentation.

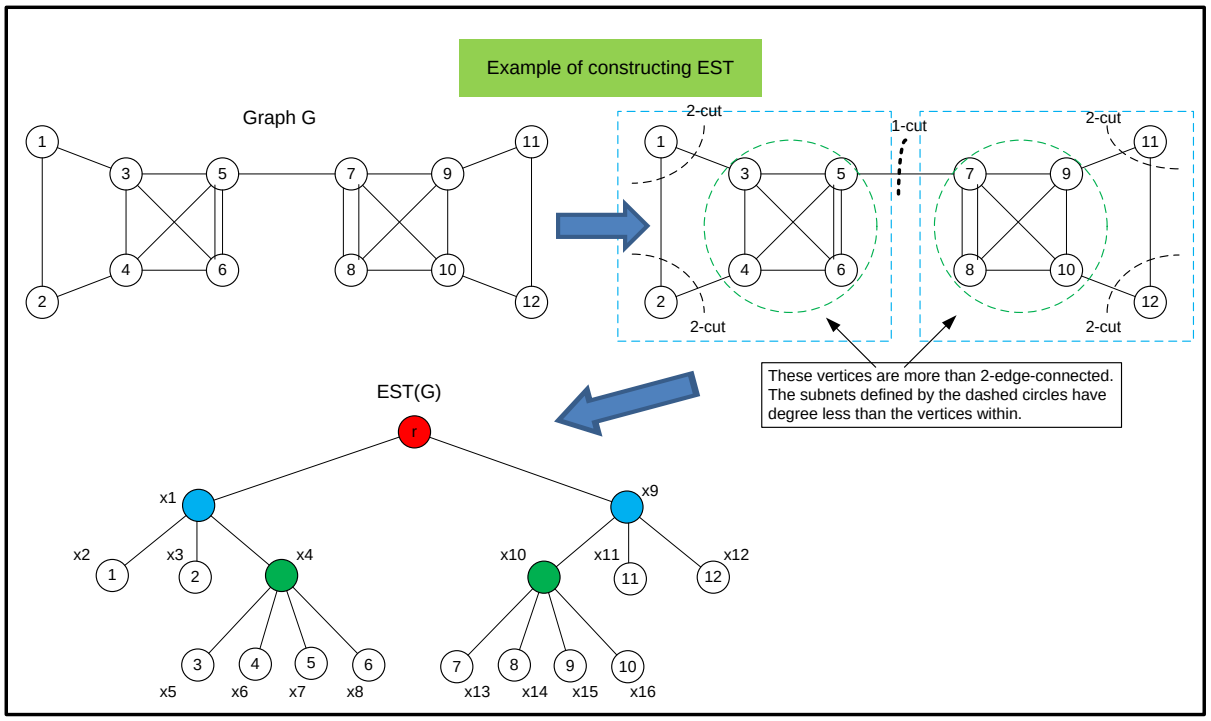

Figure 3.31: Example of EST construction.

Algorithm 3.12 consists of the following steps:

1. Construct (R, φ) (see section 3.3) of a k-connected graph, representing all its k-cuts.
2. Traverse $R = H(G)$ using a modified Depth First Search (DFS) algorithm and label the leaves ($as\ u_1; u_2; \ldots; u_x$) of the cactus in the order of the traversal.
3. Form the pairs ($\{U_i, U_{i+\lceil x/2 \rceil}\}$: $1 \leq i \leq \lfloor x/2 \rfloor$), where U_i is the set of vertices from G that is mapped to the leaf u_i of R.
4. For each pair (U_i,$U_{i+\lceil x/2 \rceil}$), arbitrarily pick 1 vertex from U_i and 1 from $U_{i+\lceil x/2 \rceil}$ and connect this pair of vertices with a single edge. If *k* is odd the process is completed by connecting the vertex $U_{\lceil x/2 \rceil}$ to an arbitrarily chosen different leaf U_j. If $\delta = 1$ the algorithm stops here. For augmenting the graph, such that $\delta \geq 2$, continue to step 5.

5. Construct the Extreme Sets Tree (EST) corresponding to graph G^i. For every leaf U_a in $H(G^i)$ find the node u_a in the tree EST^i, which corresponds to the set U_a. Because U_a is a $(k+i)$-extreme set in G^i the node u_a should exist. The idea is to find a node w_a of the EST^i in the subtree of u_a (possibly u_a itself), which satisfies the following properties, concerning the demand function $\Phi(x)$ for each node $x \in V$:
    a. $\Phi(w_{a)} > 0$
    b. $\Phi(z) = 0$ for every child vertex z of its parent w_a. (If u_a is a leaf in the EST^i then w_a=u_a.)
6. Now the graph G^{i+1} can be constructed from G^i by adding edges to the latter. For any pair (U_a,U_b) of leaves from the cactus $H(G^i)$ formed in step 1 till 3 two arbitrary vertices from *G* should be chosen, where the 1st corresponds to a leaf in the sub-tree of w_a and the 2nd to a leaf in the sub-tree w_b, in EST^i. Connect the chosen vertices by a new edge.
7. Compute the Extreme Sets Tree EST^{i+1} for the graph G^{i+1} by updating EST^i. If the number of leaves in $H(G^i)$ is odd at some phase other than the latest one ($i < \delta$), then at the end of step 3, there will be a leaf U_a that participates in two different pairs. Note that $\Phi(u_a) \geq 2$ since this is not the very last stage of the algorithm. For this particular leaf two nodes, w_{a1} and w_{a2}, need to be selected from the subtree of u_a and each one should be associated to a different pair. After that step 6 can be done as before. The following procedure explains how w_{a1} and w_{a2} can be selected:
    A node w_a should be found in the sub-tree of u_a such that (a) $\Phi(w_a) > 2$ and (b) $\Phi(z) < 2$ for every child z of w_a. If w_a is a leaf then define $w_{a1} = w_a$ and $w_{a2} = w_a$. Otherwise, let $z_1$and z_2 be two children of w_a, such that they have the largest edge demand among all the child nodes of w_a. Find a node w_{aj} (for j =1,2) in the subtree of z_j (possibly z_j itself) such that (a) $\Phi(w_{aj}) > 0$ and (b) $\Phi(z) = 0$ for every child z of w_a. There are only 3 cases possible, namely: (1) $\Phi(z_1)$= $\Phi(z_2)$=0, (2) $\Phi(z_1)$= 1 and $\Phi(z_2)$= 0 and (3) $\Phi(z_1)$= 1 and $\Phi(z_2)$=1.

Now that the edge-augmentation algorithm is explained, it will be applied on the graph of subnet ASD/RT as well as the entire graph representing KPN's PS mobile core network[18]. There are two important issues that have to be dealt with. The 1st point is that in reality the cost for adding new edges varies, depending on several factors like distance, edge-type, indoor or outdoor etc. The 2nd issue is that physically it may not be possible to add a new edge between any arbitrary pair of vertices. To deal with this, the above algorithm is modified, such that edge-augmentation is done in two stages. In the first stage the edge-connectivity is increased from 1 to 2, and the DFS procedure is done in such a way, that the output of the algorithm produces edges to be added that can also be implemented in reality. In the second stage, applied on the subnet only (and not on the entire graph), the edge-connectivity is increased to 3.

Increasing edge-connectivity of core PoP ASD/RT

Now algorithm 3.12 will be applied to $G_{ASD/RT}$ (the graph of core PoP ASD/RT), shown in figure 3.1. From $G_{ASD/RT}$ a cactus representation is constructed, according to the explanation of subsection 3.3,

[18] The application of the algorithm on other subnets is in the KPN deliverable.

which is shown in figure 3.32[19]. For the case that the target would be a 2 edge-connected network, a nice way of applying the modified DFS algorithm and related labeling would yield the 7 dashed links. As these new links (in the cactus) correspond to edges (in the target graph) that connect routers, which fulfill the same function, it is surely an implementable solution. Because the router pairs are collocated, this is a good solution for the 2-edge-connectivity target.

However, if the target would be ≥ 3-edge-connectivity, this solution is not necessarily the optimum one. Further analysis is required as described in algorithm 3.12. First the Extreme Sets Tree is constructed, as shown in figure 3.33. $EST(G_{ASD/RT})$ is actually a tree, where the root corresponds to set V and child nodes correspond to the leaves[20]. The edge demand $\Phi(x)$ of each element of the EST is calculated and denoted in table 3.1.

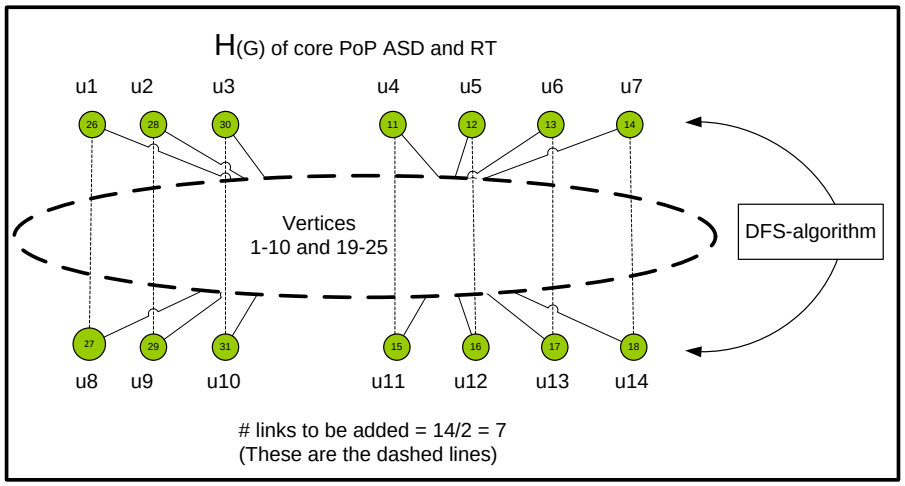

Figure 3.32: Constructing the cactus representation of core PoP ASD/RT.

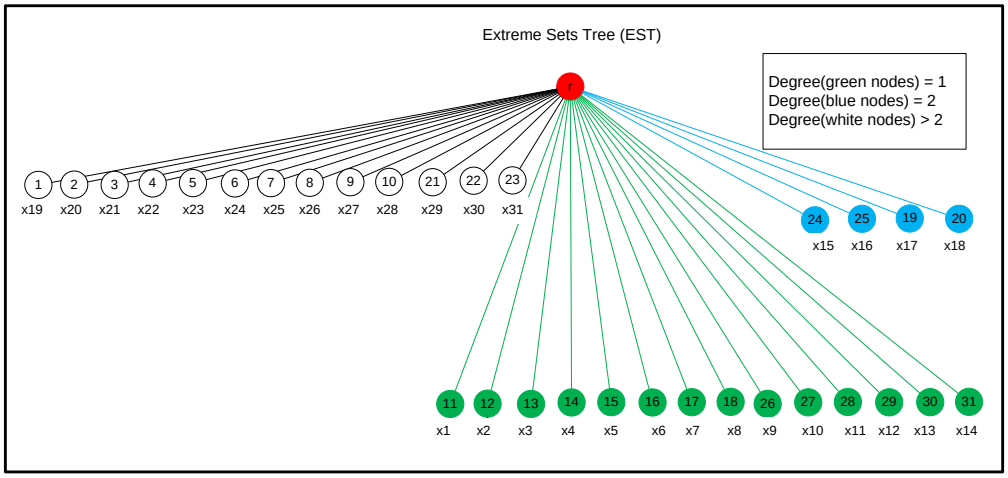

Figure 3.33: The Extreme Sets Tree of core PoP ASD/RT.

[19] Strictly taken, the dashed links are not part of the cactus.

[20] This is in agreement with lemma 4.2 of [73].

Node		Degree	Φ(.)	Node		Degree	Φ(.)	Node		Degree	Φ(.)	Node		Degree	Φ(.)
1	X19	7	0	10	X28	17	0	19	X17	2	1	28	X11	1	2
2	X20	7	0	11	X1	1	2	20	X18	2	1	29	X12	1	2
3	X21	7	0	12	X2	1	2	21	X29	4	0	30	X13	1	2
4	X22	7	0	13	X3	1	2	22	X30	4	0	31	X14	1	2
5	X23	4	0	14	X4	1	2	23	X31	6	0				
6	X24	4	0	15	X5	1	2	24	X15	2	1				
7	X25	4	0	16	X6	1	2	25	X16	2	1				
8	X26	4	0	17	X7	1	2	26	X9	1	2				
9	X27	17	0	18	X8	1	2	27	X10	1	2				

Table 3.1: Vertex degree and edge demand of core PoP ASD/RT.

For the calculations of the edge demand in the case of the above table, the following reduced equation is used: $\Phi(x) = \max\{0, (3) - d(X)\}$. Using these results, the minimum number of edges to obtain a 3 edge-connected graph is calculated.

$\Phi(r) = \Phi(ES) = \sum_{i=1}^{31} \Phi(x_i) = 32$ and $\#\ of\ links\ to\ be\ added = \left\lceil \frac{\Phi(ES)}{2} \right\rceil = 16$.

The minimum number of edges to be added to obtain 3-edge-connectivity from graph $G_{ASD/RT}$, which was 1-edge-connected is therefore 16. First the 7 previously found edges are added to obtain the 2-edge connected graph $G^1_{ASD/RT}$ as shown in the next figure.

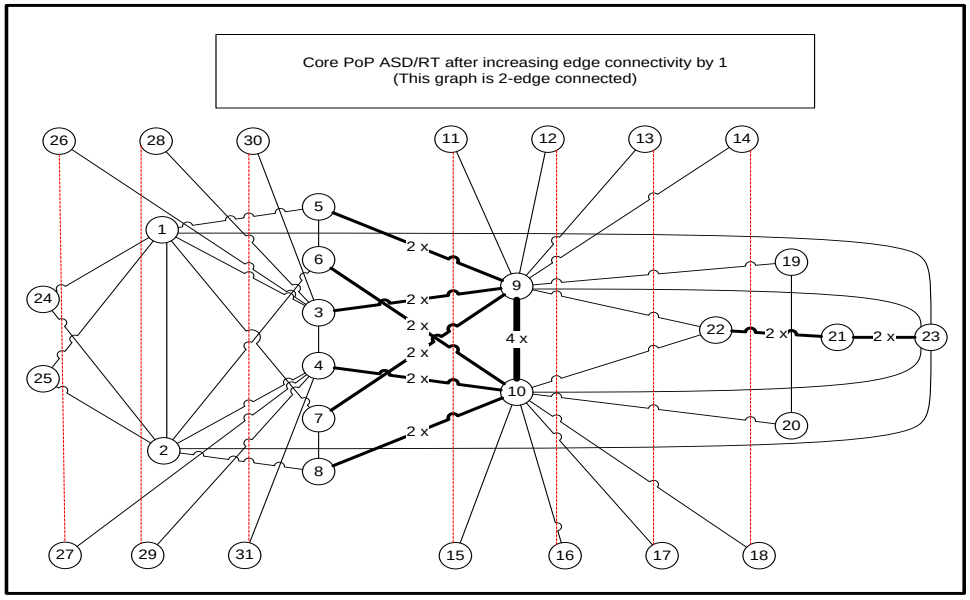

Figure 3.34: The 2-edge connected core PoP ASD/RT.

The next interesting step is to determine whether an improvement from the above graph to a 3-edge-connected graph can be achieved using $16 - 7 = 9$ additional edges, with respect to $G^1_{ASD/RT}$. The cactus representation of this graph is shown in figure 3.35. It seems that this is indeed possible by augmenting $G^1_{ASD/RT}$ with the 9 edges indicated in the next table[21].

[21] Note that there are more possibilities to achieve this target.

Labeled based edge names	Vertex-number based edge names	Labeled based edge names	Vertex-number based edge names
{U1,U10}	{24,25}	{U6,U15}	{13,14}
{U2,U11}	{11,12}	{U7,U16}	{17,18}
{U3,U12}	{15,16}	{U8,U17}	{30,19}
{U4,U13}	{26,28}	{U9,U18}	{31,20}
{U5,U14}	{27,29}		

Table 3.2: A set of edges for 2-augmenting core PoP ASD/RT.

But even though an optimal solution is found using the 2 stage approach, there is still a problem, because not all of these edges are compatible with the subnet's functionality. A realistic approach is to replace the edges {30,19} and {31,20} by the following 4 edges {9,20}, {10,19}, {30,3} and {31,4}.

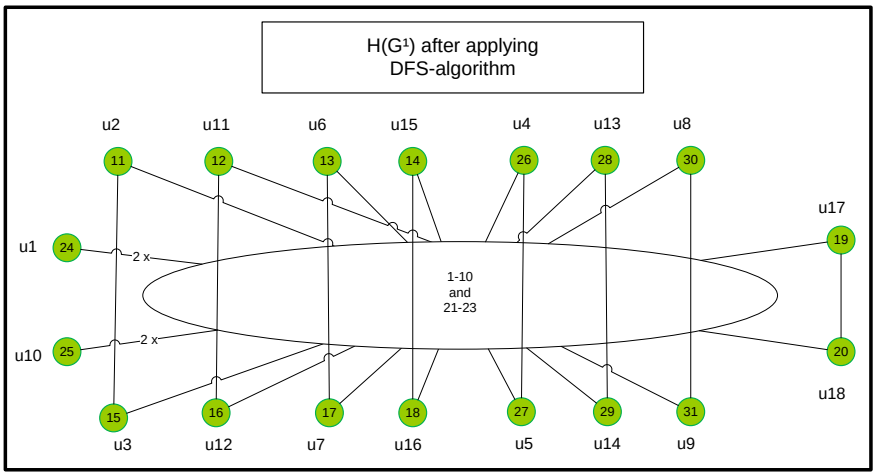

Figure 3.35: Cactus representation of the 1-augmented core PoP ASD/RT.

Increasing edge-connectivity of the entire graph

For KPN it is interesting to see how the augmentation process can be done, such that the costs are minimal. It is this constraint that makes only the increase to 2-connectivity interesting for KPN. In this subsection the edge-augmentation is considered where $\delta = 1$. Therefore the cactus of the entire graph, $H(G_{entire})$, is constructed. The red line in figure 3.36 distinguishes the green 1-edge-connected vertices form the stronger connected ones. This means that all vertices, not green, can be condensed into a single node[22].

Figure 3.37 shows the cactus of the entire graph along with the labels obtained after a "smartly" chosen modified DFS algorithm. This "smart" choice refers to the fact that the corresponding solution is both possible to be physically implemented and also cost efficient. It conforms in such a way with the analysis of the subnets, that exactly the same edges are added here as were added to the core PoP subnets[23]. Thus, with the addition of the minimum amount of 22 edges, the edge-connectivity-augmentation problem is solved for the special case of $\delta = 1$ (for the PS mobile core network).

[22] Note that only the first 4 steps of algorithm 3.12 need to be applied, as $\delta = 1$.

[23] See the KPN deliverable for the analysis of the subnets other than core PoP ASD/RT.

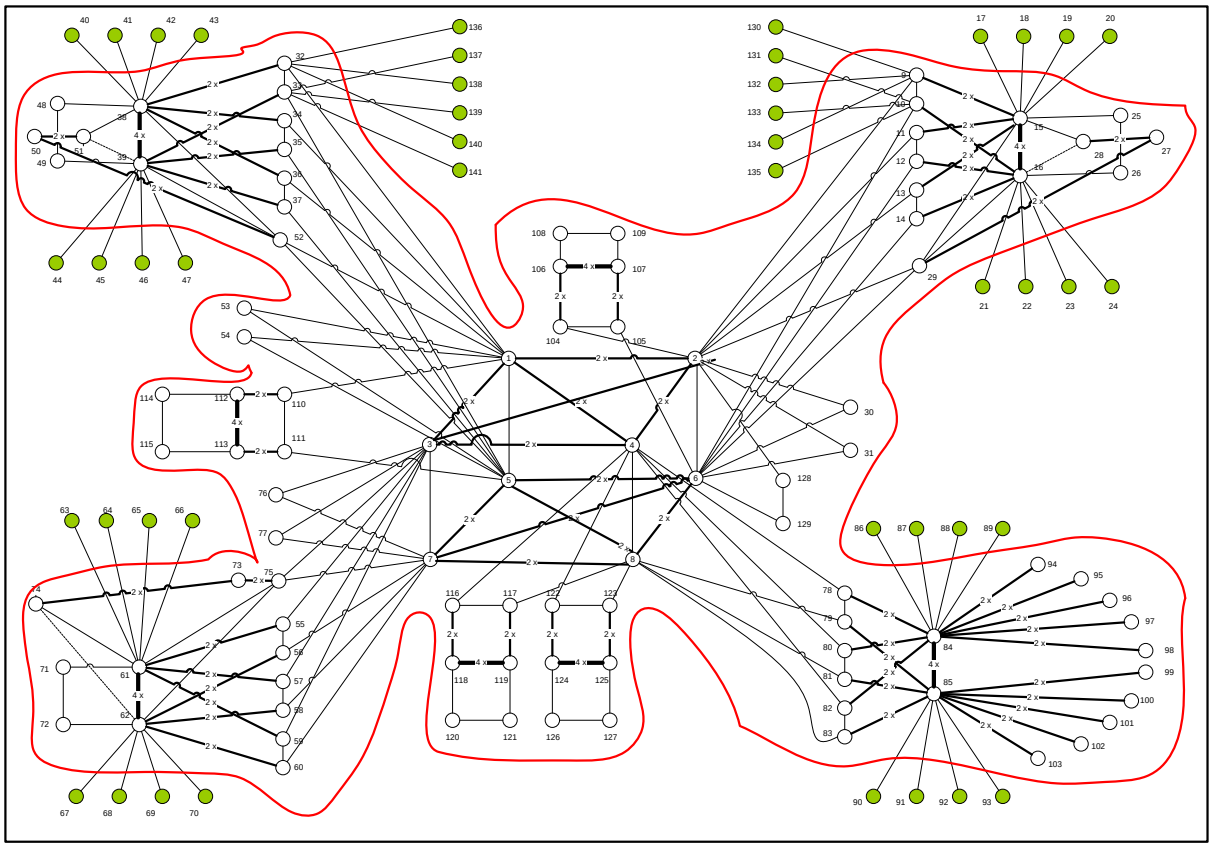

Figure 3.36: Constructing the cactus representation of the entire graph.

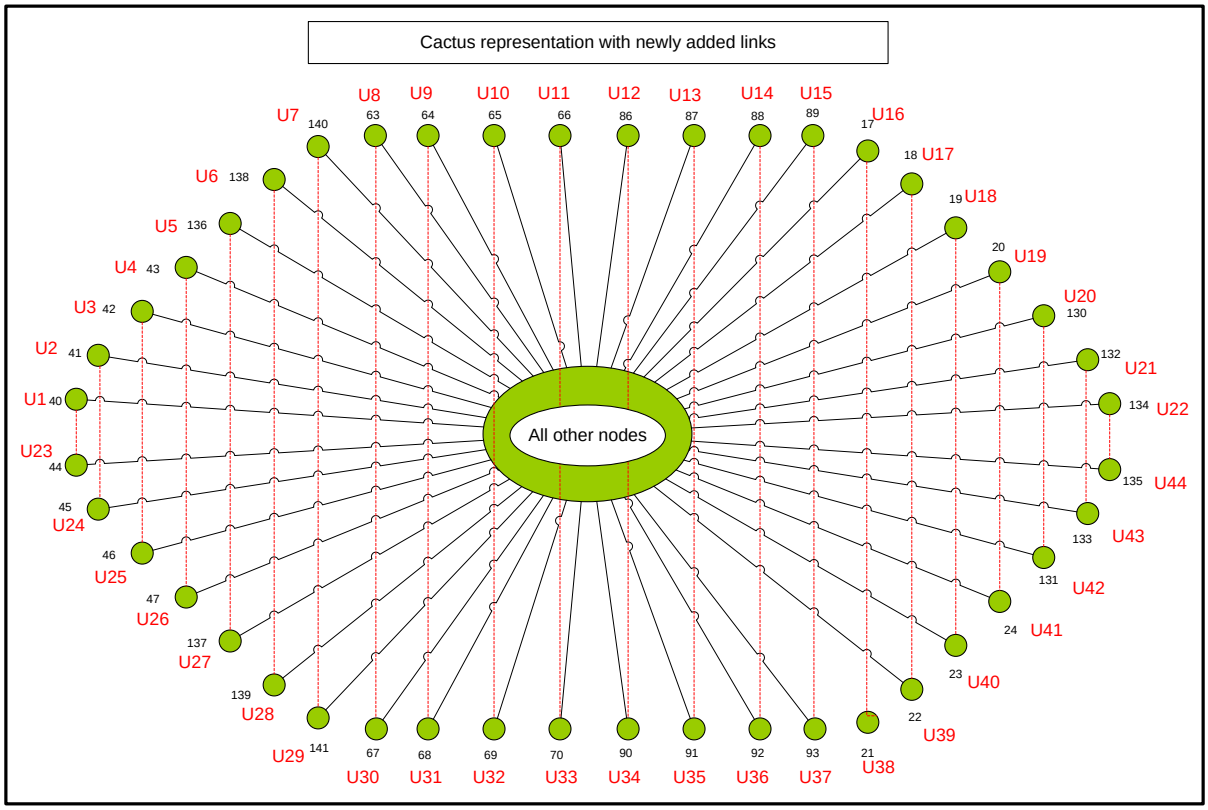

Figure 3.37: Cactus representation of the entire graph.

3.5 Vertex-connectivity augmentation

The reliability of a network will increase even more when the vertex–connectivity (connectivity in short) increases. This may be beneficial if the traffic load increases with time, in terms of improved performance. According to the results of both edge and vertex-connectivity-augmentation, KPN can make a more appropriate decision whether or not to implement new edges in the core network. The vertex-connectivity of the PS mobile core network is 1 (figure B3). This means that there is at least 1 vertex whose removal disconnects the graph. If this vertex-connectivity were to be increased to 2, any kind of single failure would never lead to the network getting disconnected, which is why a target of 2-vertex-connectivity is worth examining. Achieving a vertex-connectivity of 3 would already be too expensive.

The vertex-connectivity-augmentation problem is defined as to find an edge set of minimum size to upgrade a k-connected graph to a $(k+\eta)$-connected graph. Section 3.1 explains which algorithms are available to solve this problem. Due to the fact that the augmentation, where $\eta > 1$, would be too expensive, an algorithm is used that gives an optimal solution for obtaining a $(k+1)$-connected graph. Algorithm 3.13 is used for the purpose of increasing the connectivity by 1 and is based on Jordan's algorithm [10, 46], which finds an optimal solution for achieving 1 and 2 connected networks [10]. Algorithm 3.13 is presented next. Because an attempt is made to increase the vertex-connectivity form 1 to 2, the situation is somewhat simpler and the following steps are sufficient[24]:

1. Find the maximum number of pair-wise disjoint tight sets, $t(G)$, where a tight set of a *k*-connected graph $G(V,E)$ is a vertex set Q, such that the number of neighbors of Q is equal to *k* (denoted as $|N(Q)| = k$) and that $|V - Q| \geq (k+1)$. In other words $t(G)$ is the maximum integer $l \geq 0$, such that $D_1, D_2, \dots, D_l$ are all the tight sets in G and $D_i \cap D_j = \emptyset, (1 \leq i < j \leq l)$.
2. Find all the *k*-separators of the *k*-connected graph *G* and then find the maximum number of clusters of $(G - S)$, denoted as $b(G)$, where S is the most critical *k*-separator[25]. The most critical k-separator is that subset S, consisting of *k* vertices, which maximizes the number of clusters if it is removed from G. A separator S of a connected graph *G* is defined as an (inclusion wise) minimal subset $S \subset V$, such that $G - S$ consists of at least 2 clusters. In mathematical form: $b(G) = \max\{2, (b^*(G)|G - S, S \subset V, |S| = k)\}$, where $b(G) = 2$ if *G* has no *k*-separators.
3. Find the lower bound of newly to be added edges to graph *G* for increasing the connectivity by 1, using the following equation: $\max\{b(G) - 1, \lceil t(G)/2 \rceil\}$.
4. Construct a $(k+1)$-connected graph $\tilde{G}(V \cup s, \tilde{E})$ from the original k-connected graph, by adding a new vertex s and 1 edge between s and each $v \in V$.
5. For each $v \in V$ remove each edge {s,v} from $\tilde{G}$ if the $(k+1)$-connectivity criterion is not jeopardized. The edges between vertex *s* and the tight sets will remain after applying this step.

[24] It is advised to refer to [10] for the complete algorithm.

[25] $G - S$ is defined as graph G without a subset of vertices S and all edges incident to the vertices of S.

6. For the remaining edges apply the splitting-off theorem[26] in such a way that no more than $|E'| = \max\{b(G) - 1, \lceil t(G)/2 \rceil\}$ new edges need to be added with respect to *G*, such that a 2 connected graph $G^1(V, E \cup E')$ can be constructed.

Figure 3.38 illustrates how the minimum number of edges for the augmentation is obtained. The example graph is clearly 1-connected and there are 4 tight sets, according to step 1 of the algorithm. Step 2 is to find $b(G)$ and it can be seen that there are two 1-seperators in G, namely vertex 1 and 3. Removal of either one results in three clusters, which means that $b(G) = 3$. According to step 3, the minimum number of edges to be added is: $\max\{b(G) - 1, \lceil t(G)/2 \rceil = \max\{3 - 1, \lceil 4/2 \rceil\} = 2$.

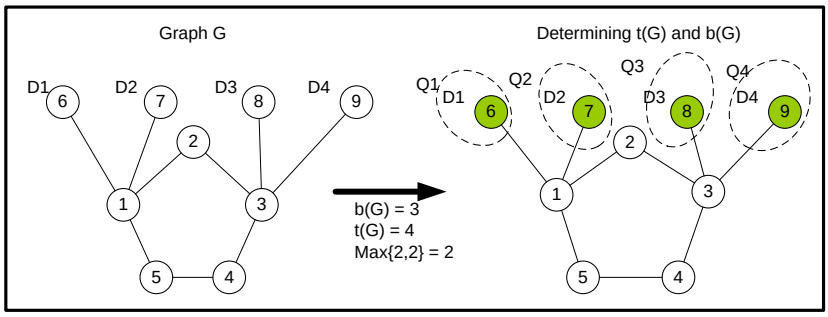

Figure 3.38: Calculating b(G), t(G) and the lower bound.

Step 4 results in the addition of the dashed edges as shown in the left graph of figure 3.39. The center graph shows the remaining edges after step 5 is applied. In this example it is clear that only the edges between vertex s and the vertices representing the tight sets remain. Finally the removal of *s* and the addition of 2 edges according to the splitting-off theorem results in a 2-connected graph G^1.

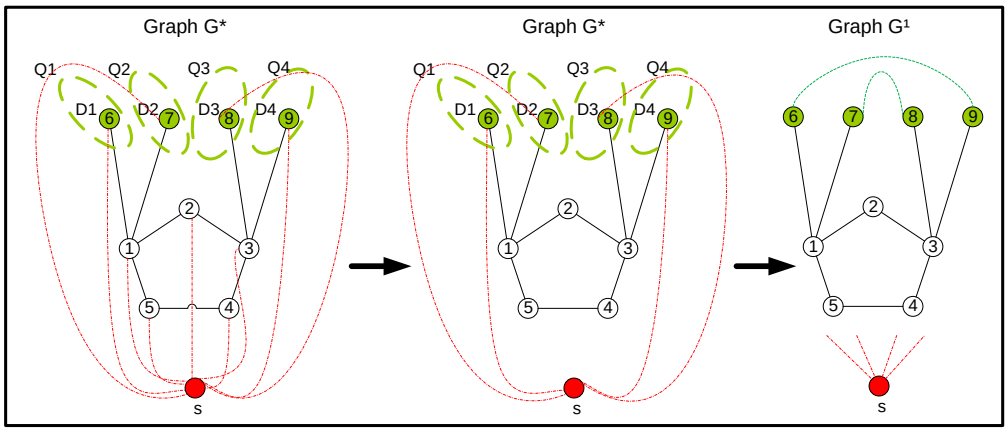

Figure 3.39: From a k-connected to a (k+1)-connected graph.

Algorithm 3.13 is applied to $G_{ASD/RT}$ (to represent the result on subnet level) and the entire graph[27]. It is assumed that the algorithm is explained well enough and therefore the results are directly given. An important factor to cope with is that the new edges should be possible to be implemented in

[26] The splitting-off theorem for vertex-connectivity says that splitting-off the edges (s, v) and (s, w) incident to node s means to remove these edges and add a new edge between node v and node w.

[27] The analysis of the other subnets can be found in the KPN deliverable.

reality. With these constraints in mind an attempt is made to increase the connectivity of the PS mobile core network.

Vertex-connectivity augmentation of core PoP ASD/RT

Figure 3.40 almost gives the answer of the edges to be added to achieve 2-connectivity. According to the algorithm a minimum of 7 new edges does the trick and the newly to be added edges are chosen conform figure 3.34. In this particular case figure 3.34 also applies to the result obtained here.

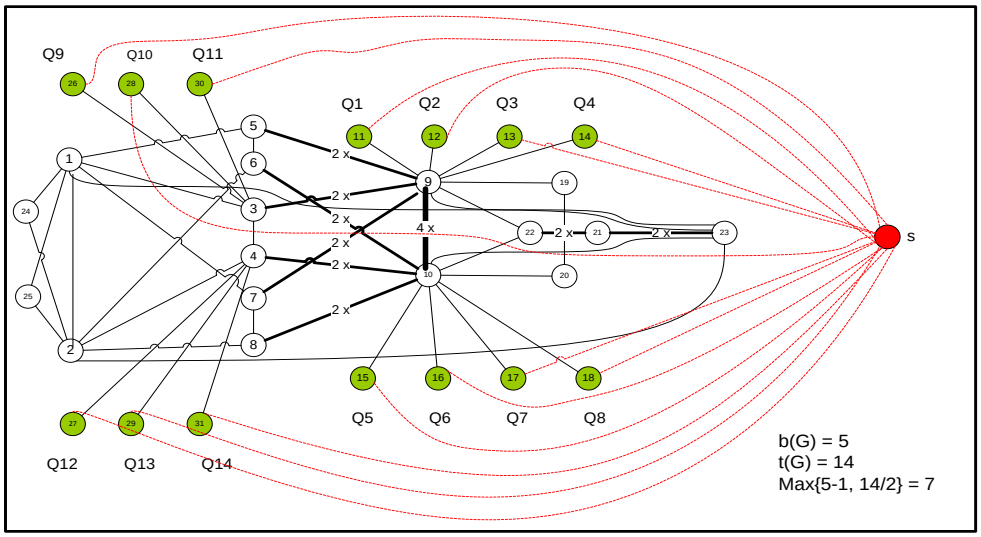

Figure 3.40: Core PoP ASD and RT edge augmentation procedure.

Vertex-connectivity augmentation of entire graph

Now that it is clear how the subnets should be augmented, algorithm 3.13 is also applied on the entire graph. In general it is not true that augmenting the subnets of a larger network ultimately leads to higher connected larger network. However, in the case of the PS mobile core network, with a target connectivity of 2, this phenomenon does occur and the result is shown in the next table.

Edge #	New edge	Subnet	Edge #	New edge	Subnet	Edge #	New edge	Subnet
1	(40,44)	RT	10	(65,69)	AH	19	(97,100)	GV
2	(41,45)	RT	11	(66,70)	AH	20	(98,99)	GV
3	(42,46)	RT	12	(86,90)	GV	21	(17,21)	ASD
4	(43,47)	RT	13	(87,91)	GV	22	(18,22)	ASD
5	(136,137)	RT	14	(88,92)	GV	23	(19,23)	ASD
6	(138,139)	RT	15	(89,93)	GV	24	(20,24)	ASD
7	(140,141)	RT	16	(94,103)	GV	25	(130,131)	ASD
8	(63,67)	AH	17	(95,102)	GV	26	(132,133)	ASD
9	(64,68)	AH	18	(96,101)	GV	27	(134,135)	ASD

Table 3.3: The new edges for augmenting the entire graph.

Figures B8 and B9 (in appendix B) show the results regarding the analysis of the entire graph. The edges to be added according to table 3.3 are the same as the edges found according to the analysis on subnet level (shown in the KPN deliverable).

3.6 Relationship between augmentation and algebraic-connectivity

This section discusses how $a(G)$ evolves when algorithm 3.12 and algorithm 3.13 are applied to a graph and discusses a problem discovered in the process. To illustrate this $G_{ASD/RT}$ is chosen, but the analysis of other subnets and G_{entire} is similar. The goal is to verify how the output of both algorithms relate to $a(G)$. In this thesis this kind of analysis is referred to as connectivity relations. Remember that the connectivity target was defined to be 2 and that anything beyond would be financially infeasible for KPN. Therefore the comparison between $\kappa_v(G)$ and $a(G)$ is done for 1 and 2-connectivity. On the other hand the comparison between $\kappa_e(G)$ and $a(G)$ is done for 1, 2 and 3-edge-connectivity.

Connectivity relations for core PoP ASD and RT

The following 2 tables give an overview of $\kappa_e(G_{ASD/RT})$ versus $a(G_{ASD/RT})$ and $\kappa_v(G_{ASD/RT})$ versus $a(G_{ASD/RT})$ respectively. The 1st column of the respective tables show the number of new edges required to increase $\kappa_e(G)$ and $\kappa_v(G)$ by 1, for each step. The reference graph for k-edge-connectivity (k-connectivity) is the graph, which is $(k-1)$-edge-connected ($(k-1)$-connected), corresponding to the previous row in each table. A comparison is made with $a(G)$, according to the chosen combinations (in section 3.4 and 3.5), but also with the maximum and minimum $a(G)$ that could have been achieved by respectively choosing a maximizing and minimizing combination of newly to be added edges. To find the maximizing and minimizing combination, each possibility to add the minimum edge set should be tried out. We have used a brute force method to compute $a(G)$ for all combinations (see Matlab code in CD). In order to simplify the problem for the vertex-connectivity of $G_{ASD/RT}$, subsets 1 and 3 as well as subsets 2 and 4 (shown in figure 3.41) are merged together into 2 sets, each consisting of 7 nodes. In the case when edges need to be added between 2 subsets there are $|E'|!$ ways of adding $|E'|$ new edges. The case of 2 subsets was used to compute the maximum and minimum $a(G)$ for all subnets (see KPN deliverable).

The 7 previously chosen edges for 2-edge-connectivity/2-connectivity are shown as dashed lines in figure 3.34. The 9 previously chosen edges, whose addition results in 3-edge-connectivity, are presented in table 3.2. There are more combinations resulting in the minimum (maximum) $a(G)$ shown in the tables below.

k-edge-connectivity		Algebraic-connectivity		
# of new edges	$\kappa_e(G)$	$a(G)$: chosen	$a(G)$: maximized	$a(G)$: minimized
0	1	0,4087	Not applicable	Not applicable
7	2	0,4087	0,6482	0,4087
9 (w.r.t previous)	3	0,5201	0,9709	0,5101

Table 3.4: Edge-connectivity compared with algebraic-connectivity of core PoP ASD/RT.

k-connectivity		Algebraic-connectivity		
# of new edges	$\kappa_v(G)$	$a(G)$: chosen	$a(G)$: maximized	$a(G)$: minimized
0	1	0,4087	Not applicable	Not applicable
7	2	0,4087	0,6482	0,4087

Table 3.5: Vertex-connectivity compared with algebraic-connectivity ASD/RT.

New problem: finding all combinations between 3 or more subsets

From figure 3.40 in the previous section it can be seen that there are exactly 4 separators, namely the singletons 3, 4, 9 and 10. Each separator has a subset of nodes, consisting of nodes of degree 1. Figure 3.41 shows 3 examples of connecting the subsets, such that 2-connectivity is achieved optimally and 1 example (d) of doing this with more than 7 edges. It is quite a task to sort out all the possibilities of adding 7 edges[28] between 4 subsets and upgrading the vertex-connectivity by 1.

In the author's opinion the general combinatorial problem of finding all possibilities of adding m' new edges across $h > 2$ subsets, each consisting of an arbitrary amount of vertices, is an interesting open problem. A solution for this particular problem has the advantage that one is able to solve the edge-augmentation (vertex-augmentation) problem in such a way that a minimum set of edges is chosen to increase the edge-connectivity (vertex-connectivity), while a particular combination can be used that produces the highest $a(G)$ (of all possible combinations) at the same time.

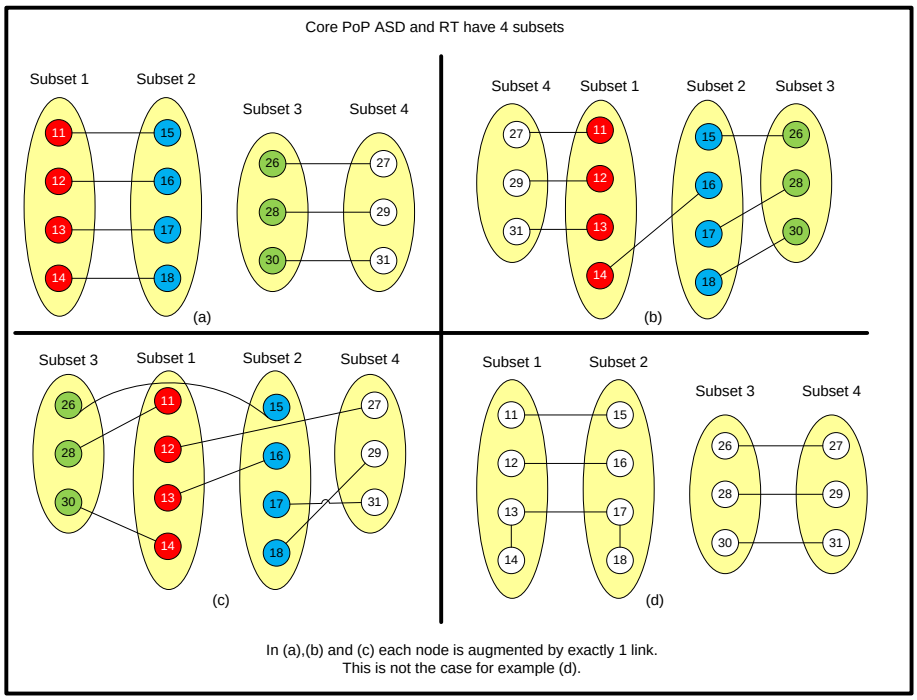

Figure 3.41: Vertex-augmentation possibilities of core PoP ASD/RT.

[28] Remember that new edges may only be added across different subsets and only one new link may be additionally incident to each vertex.

4 Capacity Management in the PS domain

Capacity management is a huge topic and just like robustness it has received a lot of attention in the academic world. Another commonality between the two is that both have impact on the performance of a network. Despite the importance, capacity management is still in a "baby"-phase (regarding the PS mobile core) and issues like bottlenecks and congestion are treated reactively. But solving these problems afterwards sometimes leads to serious performance degradation. Currently, capacity management has been given a top priority at KPN [57] and the company's target is to achieve an automated and proactive capacity management system, which includes forecasting capabilities for preventing performance degradation due to capacity shortages.

The academic world has done much research regarding efficient use of resources [63, 3, 16] and in general it seems that improving the routing mechanism, such that underutilized elements and edges are used more efficiently, reduces stress on the "popular" paths. These "popular" paths may suffer from heavy loads when less efficient algorithms (based only on shortest path routing) are used.

Table E1 (in appendix E) indicates the most important parameters, with respect to capacity management, that KPN should take into consideration. The contribution of the author's work to KPN's capacity management plan [57] is based on perhaps the most crucial capacity-parameter, namely the bandwidth. The main reason to choose only this parameter is the fact that the core network is not yet operational, making it impossible to do measurements on it. Nevertheless, a technique is found to do some bandwidth management. On the other hand, it seems difficult (and perhaps impossible) to do analysis on other parameters due to this constraint. Secondly, dealing with just one of these parameters already requires a lot of time, which means that surely it is not possible to cover all of them in the available time. The most important results are the following:

1. A program that calculates the relative amount of bandwidth usage of each network connection. This program is the called the CTA-edge-betweenness program.
2. A program that calculates the vertex-betweenness centrality for a weighted graph, while taking the effect of the edge vertices into account. The output generated by these programs is then used (along with 2 other criteria) to estimate how critical each vertex in the network is.

4.1 Bandwidth management of edges

Using only topology information in the form of a bandwidth matrix, say BW_{uv} , and traffic information in the form of a traffic matrix, say TM_{uv} , it is possible to estimate the percentage of usage of each connection in a network. The CTA-edge-betweenness algorithm, computes the amount of bandwidth usage for all network edges [80]. Figure 4.1, gives a high level overview of how the technique works. In the box located at the right lower corner, the colour scheme presents the colours used to indicate how crucial the edge usage is. If the edge usage exceeds the 60% threshold, then caution is required, because nowadays data traffic increases at an exponential rate (see the

forecast in figure E1 in appendix E). In this case the solution is to upgrade the edge(s) (or add (a) parallel edge(s)), whose usage exceeds the threshold.

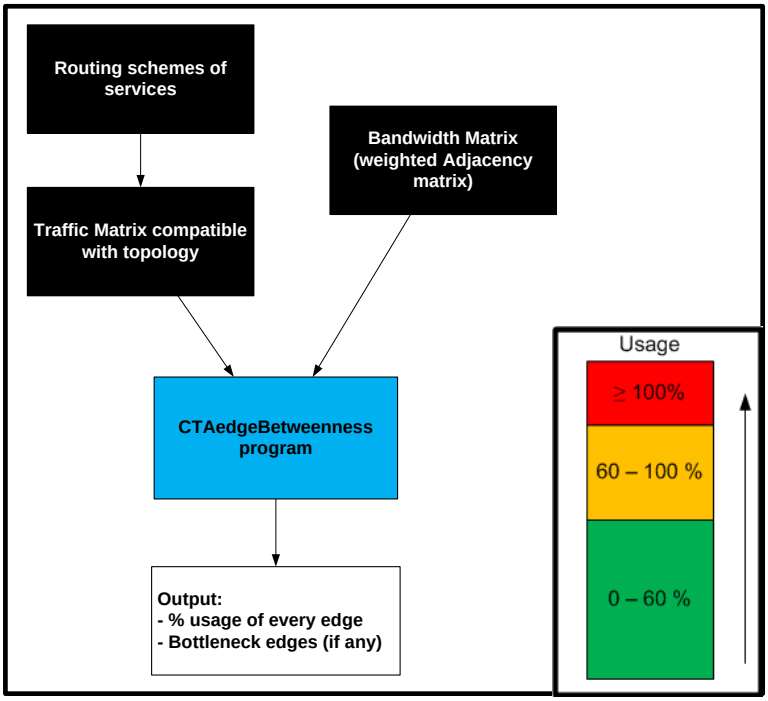

Figure 4.1: Usage of the CTA-edge-betweenness program.

The algorithm used for the CTA-edge-betweenness program is based on a modified version of calculating the edge-betweenness centrality in the network. Betweenness centrality is a graph theoretical concept that measures the degree to which a vertex or edge acts as an intermediary in the communication between every source-destination pair in a graph. The following equation is used to calculate the edge-betweenness centrality:

$$C_b(e) = \sum_{s,t \in V} \frac{\tau(s,t|e)}{\tau(s,t)} \qquad (4.1)$$

$\tau(s,t)$ denotes the number of shortest paths between source s and destination t, while $\tau(s,t|e)$ indicates the number of shortest paths between s and t passing through a given edge e. In the original conception of betweenness centrality, "shortest path" is defined in terms of the number of hops. This permits a regular Breadth First Search (BFS) algorithm [6] to identify the shortest paths. However, taking a bandwidth-weighted graph makes it interesting to consider routing traffic over less congested paths that also have the property of being relatively short in terms of the number of hops. The CTA-edge-betweenness algorithm balances the traffic distribution using both the edge capacities and the hopcount information. It is shown to be more accurate (with respect to the real situation) than simple algorithms based on the basic definition of betweenness centrality [80]. Of course there are other more complicated algorithms that can produce perhaps even better results (e.g. SAMCRA [82]), but here it is preferred to keep things as simple as possible.

There are 2 important modifications [80] to be applied to the basic edge-betweenness centrality[29] to achieve the CTA version:

[29] The basic algorithm for both vertex and edge-betweenness centrality can be found in [6].

1. Should more paths of equal hopcount exist, the one having higher bandwidth is preferred. To achieve this, the Widest Shortest Path (WSP) [31] algorithm is chosen to replace the BFS algorithm.
2. Whenever a given edge e appears in a path between end-vertices s and t, an increment proportional to the contribution of the vertex pair (s,t) to the total traffic is to be considered in the calculation of the edge-betweenness. This is an improvement because the basic edge-betweenness centrality formula (4.1), uses an increment of 1 for every vertex pair (s,t) .

Algorithm 4.2 in figure 4.3 is considered to be a subroutine, which is invoked for every source s (actually every vertex $s \in G$) by algorithm 4.1, which is the main routine. For each $s \in G$ the subroutine computes a vector $\pi = [\pi_1, \pi_2, \dots, \pi_{|V|}]$ that contains the parent vertex of every vertex $v \in V$ towards s. In fact, this vector is used as a tree rooted at s, where a parent is closer to the root than its child vertices. For the construction of this vector, algorithm 4.2 uses a hopcount vector D, a largest minimum capacity vector M and a priority queue Q. Initially the queue is loaded with s, that has priority 0. A smaller priority value p corresponds to a higher priority to be removed out of Q as shown in figure 4.2.

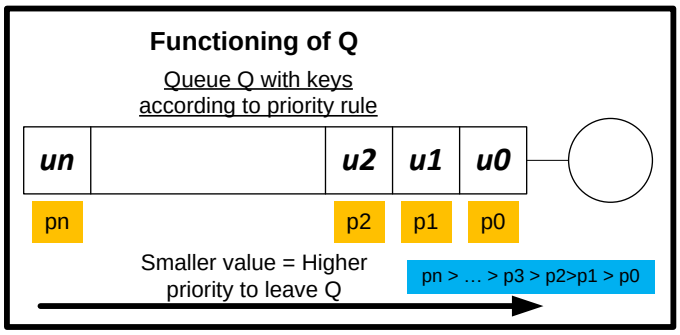

Figure 4.2: Functioning of the queue in algorithm 4.2.

In line 15 of algorithm 4.2 it is shown that the construction of π stops only when Q is empty. Line 16-17 are assumed to be clear and in line 18 it is shown how the variable γ, which is used to find the edge of largest min capacity, is updated. Line 19 is a condition which is satisfied if (1) a 1st parent vertex or (2) a parent vertex reachable through a higher capacity edge towards s is found. If this condition is indeed satisfied, then all 3 vectors (lines 20-21) are updated. Line 23 shows the rule according to which the priority queue is updated to be a function of the hopcount and the edge capacity. When the queue is empty, the vector π is returned to algorithm 4.3 (line 7).

In the main routine (algorithm 4.1), π is used as a map to find the shortest-widest path for each $v \in V$, towards the source s. For each source such a tree map is constructed. Lines 1-9 of the main routine are assumed to be clear, but line 10-14 may require some explanation to understand the algorithm. First an edge between a parent and child vertex is chosen (line 11). Then $\tau[e]$ (number of shortest-widest paths through e) is updated by using input information stored in the traffic matrix TM_{st} (line 12). After that the parent vertex is made the child vertex (line 13) for the next iteration of the while loop (lines 10-14). When the while loop is finished, the values found for τ are stored in the betweenness centrality vector C_b for each s. More information about the Matlab code for the CTA-edge-betweenness can be found in appendix C.

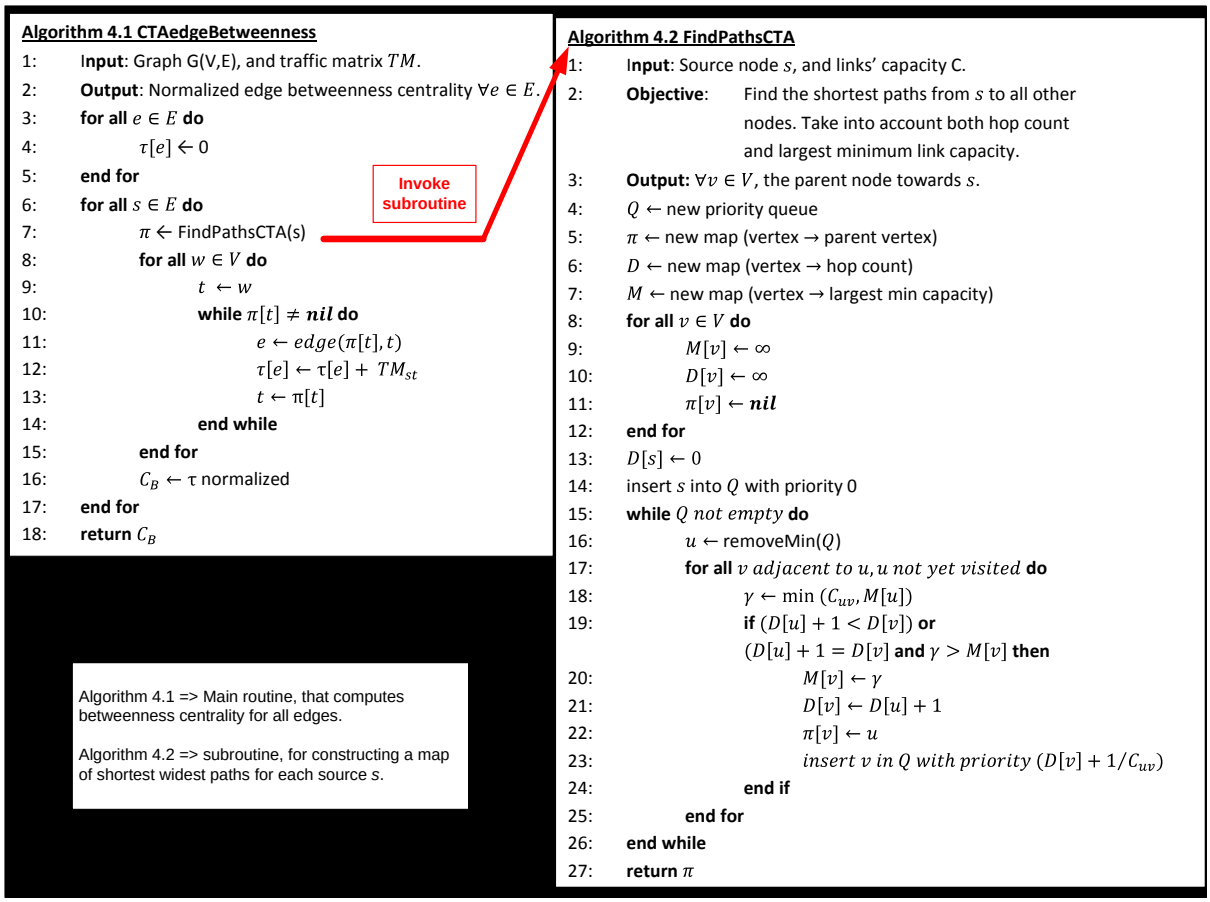

Figure 4.3: The algorithm for CTA-edge-betweenness [80].

On top of the explanation of algorithm 4.1 and 4.2, it seems that a minor modification in the latter gives an interesting result (e.g. for weighted ring graphs). In line 23 of algorithm 4.2, the priorities in constructing the widest-shortest paths are computed by the equation $D[v] + C_{uv}^{-1}$. The author has generalized this equation to $\{(\beta^{-1} \times D[v]) + (\beta \times C_{uv}^{-1})\}$, such that the variable β can be used as a tuning parameter to control the amount of influence of both the hopcount $D[v]$ and the capacity C_{uv} in the determination of the priorities (e.g. Increasing β, puts more weight on the capacity). The latter equation will be referred to as the linear rule in this report. A logarithmic rule $\{(\beta^{-1} \times D[v]) + log_{10}(\beta \times C_{uv}^{-1})\}$[30] can also be applied. Both rules will be tested with three different values for β, after the different types of traffic matrices are discussed.

As the PS mobile core is treated as the case study, the bandwidth matrix BW and traffic matrix TM of its complete graph are the input entities in the analysis[31]. However, the construction of TM is a problem, because there is incomplete information about the routing schemes of services and for constructing TM, it is necessary to have all source-destination pairs in the network. In January 2010 KPN made a start in documenting the schemes of the most important service types, but until August

[30] This type of log function is chosen in the rule to ease the computation, because most links in the PS mobile core network have a bandwidth of 10 Gbps, 1 Gbps or 100 Mbps. Of course it is possible to use other logarithmic functions as well (e.g. $\ln(x)$).

[31] The sparse form of BW_{uv} is shown in figure c14 of the KPN deliverable.

2010 a relatively small part of the complete picture was finished. Due to this disadvantage it is not possible to run a simulation based on data complying with reality. Therefore, 2 alternative scenarios, each using its own type of traffic matrix, are simulated to test the program. The following matrix types are used:

1. *TM* based on uniform packet size and transmission probability p.
2. *TM* based on the partial (available) data.

For all the traffic matrices it is assumed that routers and switches are routing vertices and are therefore not treated as sources or destinations of packets, but as intermediaries on source-destinations paths. A short sample of the output produced by the CTA-edge-betweenness program is placed as table E2 in appendix E[32]. Furthermore, histograms are generated that show how the percentage of remaining bandwidth of the edges is distributed. The relative amount of remaining bandwidth is quantified in bins, each (except 1) having a width of 10%. For example, the 90% bin represents edges with a remaining bandwidth of 85%-95%. The 100% bin is the only one, with a smaller interval, namely 95%-100%. It should be clear that there are negative bins, which are physically impossible, but should be interpreted as the amount of relative bandwidth shortage. The negative bins arise, because algorithm 4.1 gives negative values of remaining bandwidth (see table E2) for edges if the traffic load exceeds the available bandwidth. Histograms of the lists are used to analyze:

1. What happens when β is varied using a fixed packet size.
2. The output produced using different *TM* types discussed earlier.

The effect of varying β (on the 2 rules)

In figure 4.4, the comparison between the linear and logarithmic rule is done for $\beta \in \{1, 2, 10\}$, using a packet size of 100 Mb. The figure shows that the differences between the results of the 2 rules are small. They get even smaller (and even negligible) when β increases. This can be explained using the results in the table 4.1, which show that as β increases, the differences between the priorities get bigger between edges of different capacities, which are noticeable when comparing for example column 6 with column 4. The effect of increasing β on the priorities (to place the vertices in queue Q of algorithm 4.1) is such that there is less difference in the queue order when comparing both rules.

	$\beta = 1$		$\beta = 2$		$\beta = 10$	
C_{uv} (Gbps)	Linear priority	Logarithmic priority	Linear priority	Logarithmic priority	Linear priority	Logarithmic priority
0.1	$(D(v)/2) + 10$	$(D(v)/2) + 1$	$(D(v)/2) + 20$	$(D(v)/2) + 2$	$(D(v)/2) + 10\,0$	$(D(v)/2) + 10$
1	$(D(v)/2) + 1$	$(D(v)/2)$	$(D(v)/2) + 2$	$(D(v)/2)$	$(D(v)/2) + 10$	$(D(v)/2)$
10	$(D(v)/2) + 0.1$	$(D(v)/2) - 1$	$(D(v)/2) + 0.2$	$(D(v)/2) - 2$	$(D(v)/2) + 1$	$(D(v)/2) - 10$

Table 4.1: Effect of tuning parameter on linear and log rules.

Figure 4.4 shows that different values for β do not yield significant differences. However, $\beta = 2$ is a reasonable choice if one wants to put more weight on the capacity, rather than the hopcount, in the

[32] Due to the large size of both the input traffic matrices (also the sparse representations consisting of almost 7000 rows) and the generated output lists, they are not presented in this report. Instead they are stored on the CD.

priority determination. This choice also gives an opportunity to verify, which rule performs (a little bit) better.

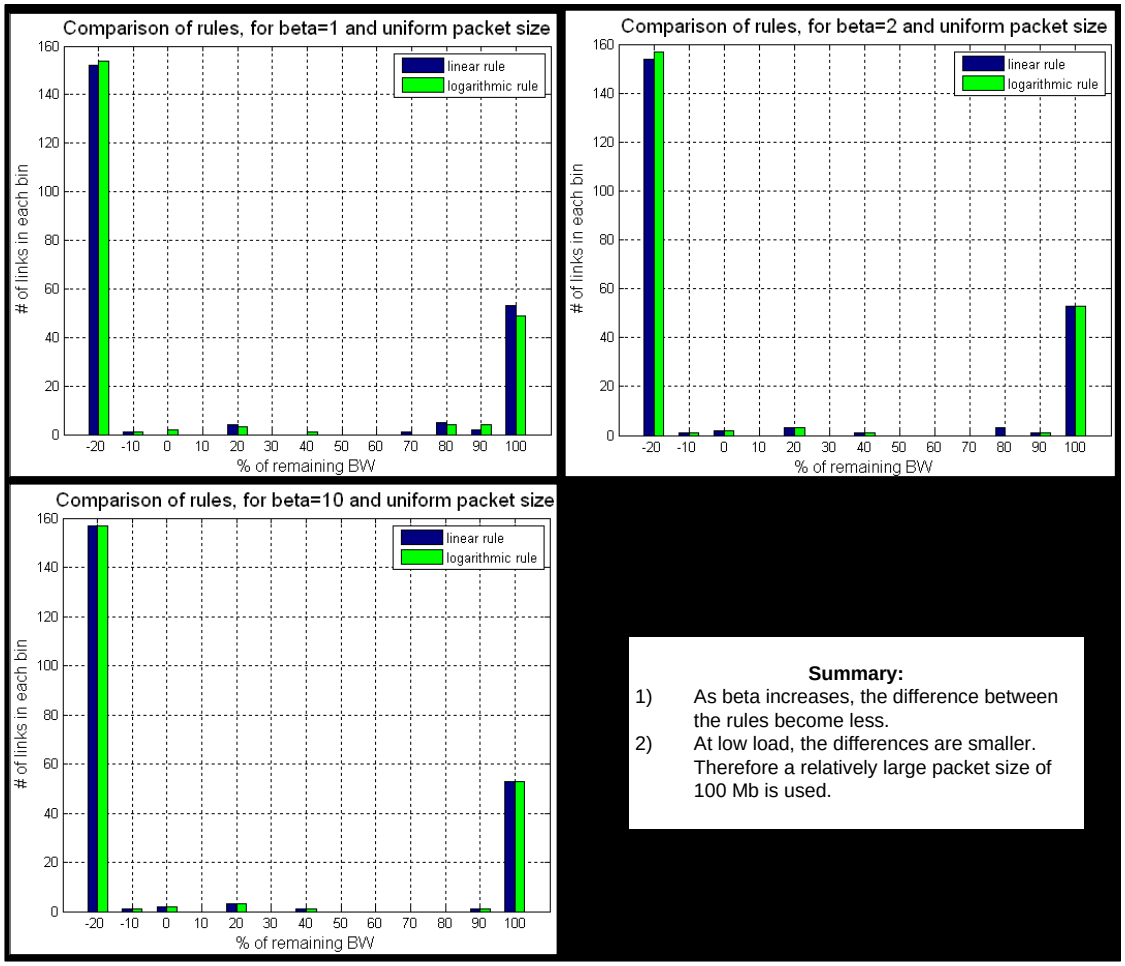

Figure 4.4: Difference of linear and logarithmic rule.

The result using different *TM's*

As the differences between the results of the 2 rules are not significant, the focus will be more on the results of the algorithm from now on (rather than on these differences). In the remainder of this subsection, $\beta = 2$ is the tuning parameter of choice and the linear rule is (arbitrarily) chosen for the next set of simulations[33].

For the 1st *TM* type it is assumed that each non-routing source vertex sends a packet of uniform size towards each possible destination (non-routing) vertex. The simulations are done for different packet sizes (0.1 Mb, 1 Mb, 10 Mb and 100 Mb). For each packet size a simulation is done for different transmission probabilities $p \in \{0.25; 0.5; 0.75; 1\}$, as shown in figure 4.5. p indicates the chance with which a packet is sent by a source vertex and this chance holds for all source vertices. The histograms

[33] From the KPN deliverable it can be verified that the logarithmic rule produces the same histograms for the case study network.

show that the remaining bandwidth of many edges become less, as the packet size increases, no matter the value of p. For a uniform packet size of 0.1 Mb, many edges are in the 100 % bin and are therefore (relatively) unused. As the packet size increases the bins to the left get higher values, meaning that the remaining bandwidth of the edges becomes less, indicating that the network load increases. At a packet size of 100 Mb, the -20% bin has the highest peak. This negative bin should be interpreted as the number of congested edges. A uniform packet size of 10 Mb will cause congestion even for a low transmission probability of $p = 0.25$. The histograms also show that the remaining capacity decreases with increasing p (e.g. compare the results of the 1st and the 4th histogram).

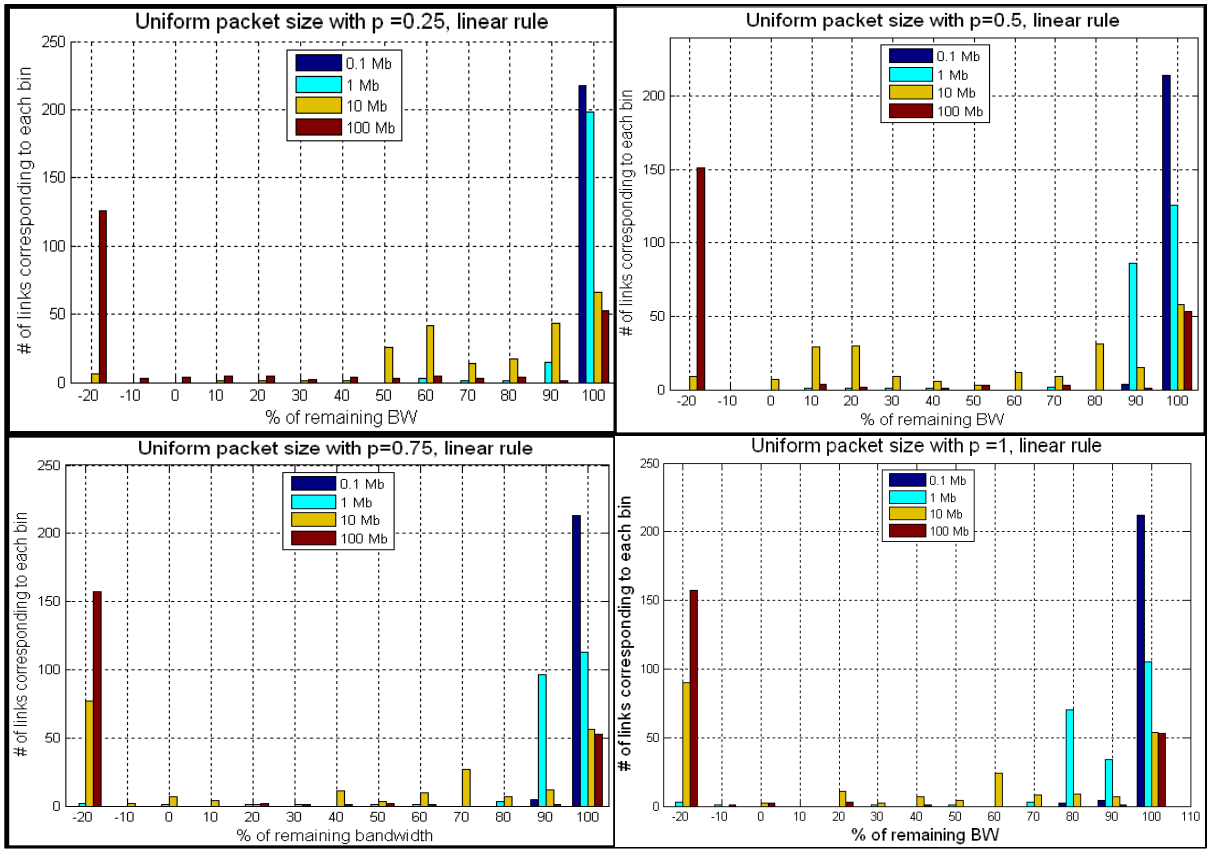

Figure 4.5: The output CTA-edge-betweenness using the 1st TM type.

For the relatively large packet size of 100 Mb, the results show that there are many edges that remain relatively unused. The amount is the same for different probabilities. This can be explained by the fact that (1) the cross edges of the network are not used under normal conditions (no failures) and (2) that the traffic matrix is not a representation of the real situation. Cross edges connect the A and B elements in figure B1 and should only be used for transmitting data under failure conditions.

The 2nd *TM* type is constructed using the (incomplete) routing information, which was available at KPN at the time of writing. For this simulation a non-uniform packet size is considered, because each service type has its own peak value (based on the busiest hour), as shown in the next table. Because the service routing schemes are strictly classified, they are not available in this report[34].

[34] For those who it may concern, this data is available at the capacity management group of KPN Care Customer. The constructed TM can be found on the CD.

	Service	Highest peak 2010		Service	Highest peak 2010
1	(Fast) Internet	*Classified*	5	APN Telfort	*Classified*
2	Portal mm	*Classified*	6	MOO (aggregated)	*Classified*
3	Blackberry	*Classified*	7	Roaming	*Classified*
4	Machine to machine	*Classified*	8	Billing	*Classified*

Table 4.2: Most important services and their peak values.

The result is shown in figure 4.6 and it shows that the network does not have any bottleneck edges. There is no heavy load to be carried by the network and the histograms confirm this as many edges (in the 100% bin) are (relatively) unused. This can be explained by the fact that:

1. Cross edges are unused, under normal conditions.
2. Only 8 services are used to model the *TM* of the 2nd type, while in reality there are more.
3. Many edges having a capacity of 10 Gbps (figure B3) in the design, should also carry the load of the Radio Network Controllers in the future. Therefore a relatively small portion of these edges is used now.

On the other hand, some edges correspond with just 19% of remaining bandwidth (RBW). A closer examination (table 4.3 combined with figure B3) reveals that these are the edges that connect the IP BB to the GGSN. As these are very important connections for the functioning of the core network, it is advised that they should be monitored and upgraded if necessary.

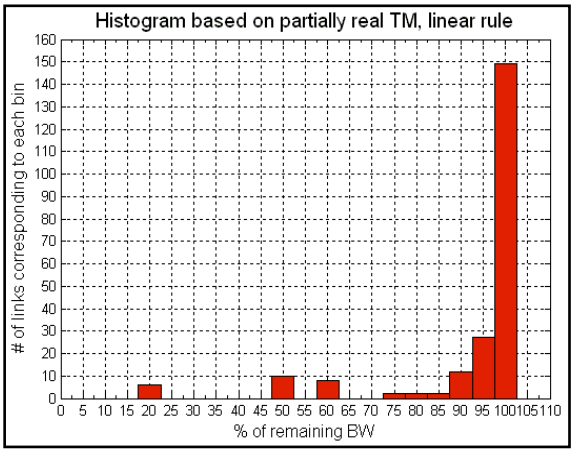

Figure 4.6: Output CTA edge-betweenness using the 2nd TM type.

Edge_#	Vertex_i	vertex_j	%RBW	Edge_#	Vertex_i	vertex_j	%RBW
47	2	29	19	87	5	52	19
48	6	29	19	125	3	75	19
86	1	52	19	126	7	75	19

Table 4.3: The edges which exceed the safety margin.

4.2 Vertex criticality

It is possible to estimate the relative importance of each vertex within its network. The vertex criticality ($VC(v)$), gives the network operator a "rough" idea how much the loss of each vertex affects the network as a whole, and therefore it indicates how important it is relative to its network. The higher the criticality of a vertex is, the more negative the impact is on the network if it were to be perturbed. The vertex criticality $VC(v)$ is defined by the author as follows:

$$VC(v) = C_b(v) + RI(v) + SC(v); for\ all\ v \in V\ in\ G(V,E) \qquad (4.3)$$

The equation states that the vertex criticality $VC(v)$ of vertex v is the sum of its vertex-betweenness centrality $C_b(v)$, its relative importance in the network $RI(v)$ and the size of an external network (cloud) $SC(v)$ connected to it, if any. All the three components of $VC(v)$ are explained next.

Vertex-betweenness centrality

In [80] it is shown how the CTA-edge-betweenness algorithm computes the edge criticality for every network connection, using a notion of edge-betweenness centrality. Analogous to this, vertex criticality is defined based on vertex-betweenness centrality [6] and two other factors. Vertex-betweenness centrality is usually computed using the following equation:

$$C_b(v) = \sum_{s,t \in V} \frac{\tau(s,t|v)}{\tau(s,t)} \qquad (4.2)$$

In this case $\tau(s,t|v)$ indicates the number of shortest paths between s and t passing through a given vertex v, while $\tau(s,t)$ is defined similarly as in equation (4.1). In [6] there are several variants for computing the vertex-betweenness centrality, namely:

1. The basic algorithm, based on the hopcount.
2. A variant which takes vertices at the periphery of the network into account.
3. A variant that takes the effect of edge weights into account.
4. A combination of 2 and 3 (our own contribution).

It is necessary to understand the efficient procedure used to compute $C_b(v)$ in the basic algorithm. Efficient computation is based on the fact that the cubic number of pair-wise dependencies $\zeta(s,t|v) = \tau(s,t|v)/\tau(s,t)$ can be aggregated, without computing all of them explicitly. If one sided dependencies[35] are defined as $\zeta(s|v) = \sum_{t \in V} \zeta(s,t|v)$ for all $s,t \in V$, the following can be exploited [5]:

$$\zeta(s|v) = \sum_{\substack{w:(v,w)in\ E\ and \\ dist(s,v)=dist(s,v)+1}} \frac{\tau(s,v)}{\tau(s,w)} \times (1 + \zeta(s|w)) \qquad (4.4)$$

This relation is recursive and asserts that the dependency of a vertex s on some vertex v can be derived from dependencies on vertices one hop further away. The basic algorithm uses this, since $C_b(v) = \sum_{s \in V} \zeta(s|v)$, by iterating over all vertices $s \in V$, each time computing $\zeta(s|v)$ for all $v \in V$ in two steps. The 1st step is a breadth first search used to find distances and shortest-path counts

[35] One sided dependency refers to the fact that the ratio $\zeta(s,t|v) = \tau(s,t|v)/\tau(s,t)$ becomes only dependent on the source side if a summation as $\zeta(s|v) = \sum_{t \in V} \zeta(s,t|v)$ is done over all possible destination vertices $t \neq s$.

relative to s. In the 2nd step all vertices are visited in reverse order of their discovery (so those farthest from s first). The 2nd variant for computing $C_b(v)$ has one difference with the former, in that $\zeta(s,t|v) = 1$ in the case that $v \in \{s,t\}$. This has the effect that vertices at the edge of the network also contribute to the betweenness score of each $v \in V$. The 3rd variant works with a weight matrix such that the shortest-path is actually a minimum weight path, instead of a minimum hop path. This is certainly a feature that is also implemented in the routing strategy of the intelligent edge core network.

The last variant is the one used for calculating $C_b(v)$ and is obtained by combining the algorithms for the 2nd and the 3rd variant, where the latter two are explained in [6]. Including the contribution of vertices at the edge (periphery) is not really a necessity, but it does ensure that $C_b(v) \neq 0\ (\forall\ v \in V)$. This in turn ensures that $VC(v) \neq 0\ (\forall v \in V)$ if for some vertex $RI(v) + SC(v) = 0$ would hold. On the other hand, including the weighted scenario conforming to variant 3 is required as it is closer to the real situation.

Figure 4.7 shows both the algorithm and the functioning of variant 4. Notice that a Weight (or Cost) matrix is the input of this algorithm[36]. The priority keys of $v \in V$, used for Q are dependent on the distance (or hopcount) relative to source s. The most intriguing part is the while loop (that runs as long Q is not empty), in line 9-22. The loop runs for each vertex $s \in V$, because each vertex becomes the source exactly once (line 4). In line 10, v is extracted from Q and pushed into the stack S as shown in figure 4.8. Each time v is chosen, such that it has the smallest distance towards s. For each neighbour of v, the path discovery and path counting procedures are done. The distances are updated by addition of the edge weights $\xi(v,w)$, which are taken from the weight matrix. After $dist[w]$, Q, $\tau[w]$ and the predecessor list $Pred[w]$ are processed properly in the Path discovery procedure, the path counting procedure is invoked to construct the predecessor list. Note that a vertex can have multiple predecessors/parents. This means that the algorithm considers all possible shortest paths and not just one in the case that there are more options. The predecessor list and the stack are used in the accumulation procedure to compute the betweenness centrality. The effect of vertices at the edge of the network is counted in the accumulation as shown in line 23 and 28. Line 23 counts the number of times that s is a source (end-vertex) for every other vertex in the graph, while line 28 counts each destination end-vertex $(w \in V|w \neq s)$ once for every source s. Equation 4.4 is used in line 27 to update the ratio $\zeta(s,t|v) = \tau(s,t|v)/\tau(s,t)$. For an elaborate explanation of this it is advised to read [5]. Finally line 28 shows how the vertex-betweenness centrality is updated, using the result of the previous line.

[36] The weight matrix used for the Intelligent Edge core can be found in the KPN deliverable.

Algorithm 4.3 Betweenness in valued networks

1: **Input**: directed graph $G(v, E)$ with edge weights $\xi: E \to R_{>0}$.
2: **Data**: priority queue Q with keys $dist[.]$and stack S (both initially empty) for all $v \in V$.
$dist[v]$: distance from source.
$Pred[v]$: list of predecessors on shortest paths from source.
$\tau[v]$: number of shortest paths from source to $v \in V$.
$\zeta[v]$: dependency of source on $v \in V$.
3: **Output**: betweenness $C_B[v]$ for all $v \in V$ (initialized to 0).
4: **for** s $\in V$ **do**
Single-source shortest path problem
Initialization
5: **for** $w \in V$ **do** $Pred[w] \leftarrow empty\ list$
6: **for** $t \in V$ **do** $dist[t] \leftarrow \infty;\ \tau[t] \leftarrow 0$
7: $dist[s] \leftarrow 0;\ \tau[s] \leftarrow 1$
8: enqueue $s \to Q$
9: **while** $Q\ not\ empty$ **do**
10: extract $v \leftarrow Q$ with minimum $dist[v]$; push $v \to S$
11: **foreach** $vertex\ w\ such\ that\ (v, w) \in E$ **do**
Path discovery (% shortest path to w)
12: **if** $dist[w] > dist[v] + \xi(v, w)$ **then**
13: $dist[w] \leftarrow dist[v] + \xi(v, w)$
14: insert/update $w \to Q$ with new key; $\tau[w] \leftarrow 0$;
15: $Pred[w] \leftarrow empty\ list$
16: **end**
Path counting
17: **if** $dist => dist[v] + \xi(v, w)$ **then**
18: $\tau[w] \leftarrow \tau[w] + \tau[v]$
19: append $v \to Pred[w]$
20: **end**
21: **end**
22: **end**
Accumulation
23: $C_B[s] \leftarrow C_B[s]$ + (|S|-1) *(% number of times s is a source)*
24: **for** $v \in V$ **do** $\zeta[v] \leftarrow 0$ **end**
25: **while** $S\ not\ empty$ **do**
26: $pop\ w \leftarrow S$
27: **for** $v \in Pred[w]$ **do** $\zeta[v] \leftarrow \zeta[v] + (\tau[v]/\tau[w]).(1 + \zeta[w])$ **end**
28: **if** $w \neq s$ **then** $C_B[w] \leftarrow C_B[w] + \zeta[w] + 1$ **end** *(% w is target of s once)*
29: **end**
30: **end**

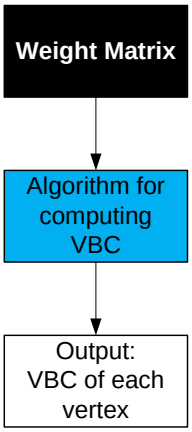

Figure 4.7: Algorithm and functioning of the vertex-betweenness-centrality program.

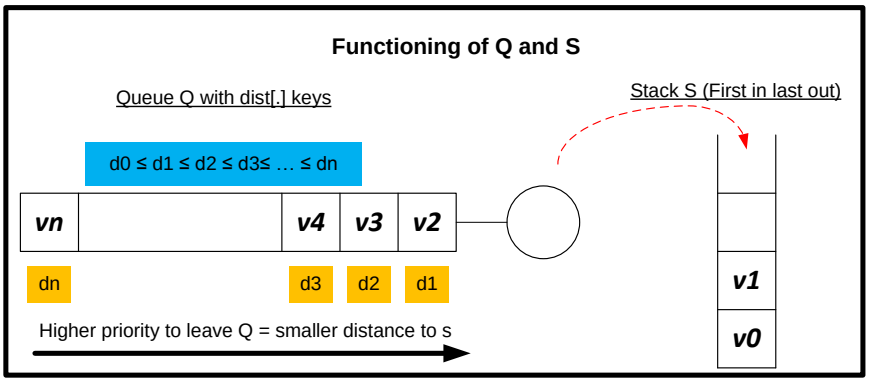

Figure 4.8: Functioning of the queue and stack of algorithm 4.3.

The relative importance function $RI(v)$

In equation (4.3) $RI(v)$ stands for relative importance of vertex v and as the name already says, it is defined as a function that indicates the relative importance of each network element. This definition gives the network operator some freedom to use it as a tuning value.

For the mobile core network, consisting of different types of both routing and functional elements, this function does make sense. In accordance with some members of the capacity management group, a list is produced (table 4.4) with relative importance factor $\Omega(v)$ for each vertex type, which is based on logical reasoning and the work experience of this group[37]. To make this factor have any significant meaning in the vertex criticality estimation the relative importance is computed as follows:

$$RI(v) = \Omega(v) \times \max\{C_b(u) \,|(for\ all\ u \in V)\} \qquad (4.5)$$

Element type	$\Omega(v)$	Element type	$\Omega(v)$	Element type	$\Omega(v)$
SGSN	1	MDX	0.5	ZR2	0.5
GGSN	1.5	ZR1	0.5	ASW	0
TR 4	0.5	VRFLR	0.5	SRLB	0
MOO SW	0.5	CSW	0.5	SRIPS	0
ITR	0.5	ZR3	0.5	FR	0
CR	0.5	BOG	0.5	STAR	0

Table 4.4: The relative importance factor list.

The size of the connected cloud function

$SC(v)$ is a function of the size and type of one or more external networks (clouds), if any, connected to vertex v. It is obvious that vertices (such as a border gateway, an Internet access router, etc.) that connect the own network to external networks are very important. If these elements were to fail the own network gets isolated, which can be catastrophic for an operator. The next table contains a size of cloud factor, $\omega(v)$, for all types of network elements that serve as a gateway to an external network, from the perspective of the PS mobile core network. The clouds can be seen in figure B2 in

[37] The meaning of the abbreviations used in this table (and table 4.5) can be found in appendix A.

appendix B. Similarly as with the previous component, the function of the 3rd component of the vertex criticality is derived as:

$$SC(v) = \omega(v) \times \max\{C_b(u) \,|(for\ all\ u \in V)\} \qquad (4.6)$$

Name	$\omega(v)$	Name	$\omega(v)$
SGSN	1	BOG	0.5
TR 4	1	MDX	0.5
MOO SW	1	ASW	0.5
ITR	1		

Table 4.5: The size of cloud factor list.

The result for vertex criticality is summarized in tables E3 and E4 in appendix E, from which it can be seen that $\max\{C_b(u) \,|(for\ all\ u \in V)\} = 9730$. The vertex-criticality results are now being used as a (rough) guideline in prioritizing the network elements and it therefore helps the capacity management group (at KPN) to start setting up the proactive capacity management system. The next figure[38] shows the histogram corresponding to this table and it is clear that no vertex has a betweenness centrality of 0, while $SC(v)$ and $RI(v)$ is zero for many elements. The former corresponds to the fact that the effect of edge vertices was taken into account when computing the vertex-betweenness-centrality. Most vertices have a betweenness centrality of 280 (corresponding to the 500 bin). Those with high value are mostly the core routers in the IP BB and the core switches in the PoP locations. The highest peak for $SC(v)$ (corresponding to 0) is logic, because not many network elements are connected to a cloud as shown in figure B2. On the other hand, most elements have $RI(v) = 0.5$, which explains (in combination with equation 4.5) the peak of this component at the bin of 2500. Figure 4.9 also shows that many vertices have the minimum vertex criticality (mostly the access switches), while few have a high value. It is obvious that the elements in the rightmost bins are crucial to be monitored.

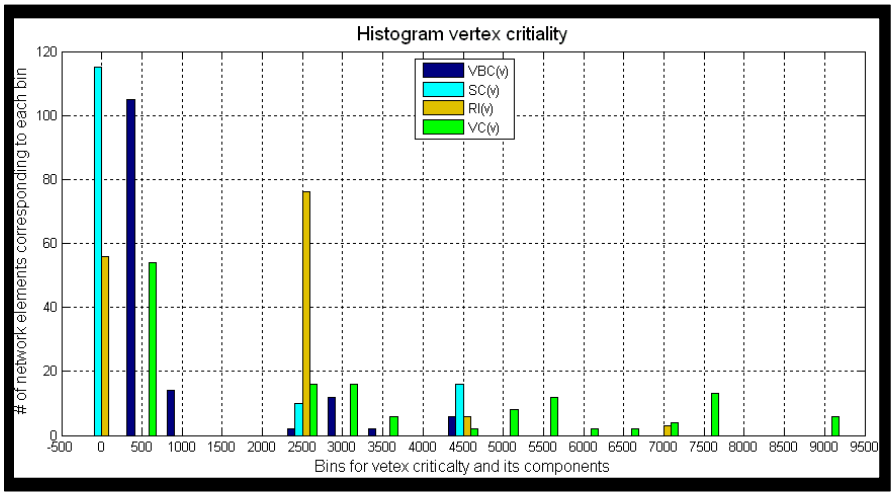

Figure 4.9: Histogram for vertex criticality.

[38] Each bin has a width of 500 (e.g. bin 500 goes from 250 to 750).

5 Conclusions

1. The network drawings (as a result of this thesis) of the PS mobile core network and its subnets are based on the description of several documents [2, 56, 58, 59, 37, 33] and explanations given by several architects and designers. Because these drawings are considered to be designs, there is a small chance that certain changes may be applied to the network and that the results regarding robustness and capacity management may not be applicable anymore. Therefore the methods and tools used are generic and can be used for any network.

2. Three sub-optimal strategies have been used to increase the algebraic-connectivity of a graph G. In the case of the PS mobile core network (and its subnets), the most effective one is the Fiedler vector strategy[39], which uses the Fiedler vector for finding the new edges to be added to the network in an iterative process.

3. For designed networks it is better (in general) to increase robustness by applying edge-connectivity or vertex-connectivity augmentation, instead of increasing the algebraic-connectivity. The reason is that the former 2 methods produce optimal solutions, while increasing $a(G)$ optimally is NP-complete. The former 2 directly focus on the weak spots, by optimally increasing the number of edge or vertex disjoint paths and this is a more efficient way to increase the network's resilience to edge and vertex failures.

4. An algorithm that can be used to write a program that generates a cactus $R = H(G)$ out of a graph $G(V, E)$ is a major step forward in the analysis of increasing the edge-connectivity and therefore increasing the network's robustness. With respect to this a quite abstract algorithm has been extended, that can be used for this purpose. By integrating 5 new sub-algorithms into the former, a less abstract algorithm is achieved, that can be used for writing a tool that automatically generates a cactus representation.

5. As there are no means of performing measurements on the PS mobile core network (because it is not in place yet), the accuracy of the CTA-edge-betweenness tool cannot be verified. However, the tool should be used to estimate where potential bottlenecks/congestion may occur, when no measurements are available. The vertex-criticality tool should be used to prioritize the network elements. This tool is useful in the beginning phase of setting up a capacity management environment, as it indicates which elements to start with.

[39] In this thesis this strategy is also referred to as strategy 3.

Bibliography

[1] N.M.M. de Abreu, *Old and new results on algebraic-connectivity of graphs*, Linear Algebra and its Applications 423: 53-73, 2007.

[2] S. Al, R. Kuiters and J. van Huessen, *Intelligent Edge Design Guide*, KPN intern, 2008.

[3] J. Allspaw, *The Art of Capacity Planning*, O'Reilly Media, first edition, 2008.

[4] A. Benczur, *Augmenting undirected connectivity in RNC and in* $\tilde{O}(|V|^3)$ *randomized time*, Proc. of the twenty-sixth annual ACM symposium on theory of computing**:** 658-667, 1994.

[5] U. Brandes, *A faster algorithm for betweenness centrality*, Journal of Mathematical Sociology 25 (2): 163-177, 2001.

[6] U. Brandes, *On variants of shortest path betweenness centrality and their generic computation*, Elsevier, Social Networks Vol. 30 issue 2: 136-145, 2008.

[7] R. H. Byrne, J.T. Feddema and C.T. Abdullah, *Algebraic-connectivity and Graph Robustness*, Unlimited Release SAND2009-4494, 2009.

[8] G. Cai and Y. Sun, *The minimum augmentation of any graph to a K-Edge-Connected graph*, NETWORKS, Vol. 19, pp. 151-172, 1989.

[9] E. Cheng and T. Jordan, *Successive edge-connectivity augmentation problems*, Math Program 84: 577 – 593, Springer, 1999.

[10] J. Cheriyan and R. Thurimella, *Fast Algorithms for k-Shredders and k-Node Connectivity Augmentation*, Journal of Algorithms 33: 15-50, Academic Press, 1998.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C.Stein, *Introduction to Algorithms*, 2nd edition, The MIT Press, 2001.

[12] K.P. Eswaran and R.E. Tarjan, *Augmentation problems*, SIAM J. Comput., Volume 5 Issue 4: 653-665, 1976.

[13] S. Fallat and S. Kirkland, *Extremizing Algebraic-connectivity Subject to Graph Theoretic Constraints*, The Electronic Journal of Linear Algebra, A publication of the International Linear Algebra Society, Volume 3: 48-74, 1998.

[14] M. Fiedler, *Absolute Algebraic-connectivity of Trees*, Linear and Multilinear Algebra, Vol. 26: 85–106, 1990.

[15] M. Fiedler, Algebraic *connectivity of graph,* Czechoslovak Math. J., 23:298-305, 1973.

[16] S. Floyd and V. Jacobson, *Link-Sharing and Resource Management Models for Packet Networks*, IEEE/ACM Transactions on Networking, Vol. 3 No. 4, 1995.

[17] A. Frank, *Augmenting graphs to meet edge-connectivity requirements*, SIAM Journal on Discrete Mathematics**,** Volume 5, Issue 1, pages 25-53, Society for Industrial and Applied Mathematics, 1990.

[18] A. Frank, *Connectivity augmentation problems in network design*, State of the art 1994, (J.R. Bridge and K.G. Murty, Eds.), pages 34-36, 1994.

[19] H. Frank and Chou, Connectivity considerations in the design of survivable networks, IEEE Transactions on Circuit Theory, Volume 17 Issue 4: 486–490, IEEE explore, 1970.

[20] A. Frank and T. Jordan, *Minimal edge-coverings of pairs of sets*, J. Comb. Theory Ser. B 65(1):73-110, 1995.

[21] A. Frank and L. Vegh, *An algorithm to increase the node-connectivity of a digraph by one*, Discrete Optimization, Vol. 5 Issue 4: 677-684, ScienceDirect, 2008.

[22] G. Frederickson and J. Já'Já', *Approximation algorithms for several graph augmentation problems*, SIAM J. Comput. 10: 270–283, 1981.

[23] G. Frederickson and J. Já'Já', *On the relationship between the biconnectivity augmentation and traveling salesman problems*, Theoretical Computer Science, Volume 19, Issue 2: 189-201,1982.

[24] H. Gabow, *A matroid approach to finding edge-connectivity and packing arborescences*, Journal of Computer and System Sciences, Volume 50 Issue 2: 259-273, 1995.

[25] H. Gabow, *Applications of a Poset Representation to Edge-connectivity and Graph Rigidity*, 32nd Annual Symposium of Foundations of Computer Science: 812-821, 1991.

[26] A. Galluccio and G. Prioetti, *Polynomial Time Algorithms for 2-Edge-Connectivity Augmentation Problems*, Algorithmica (2003) 36: 361 – 374, New York, 2003.

[27] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum flow problem,* Journal of the ACM 35(4): 921-940, 1988.

[28] R. Grone and R. Merris, *Algebraic-connectivity of trees*, Czechoslovak Math. J. Vol. 37 No. 4: 660 -670, 1987.

[29] R. Grone and M. Morris, *Ordering trees by algebraic-connectivity*, Graphs and Combinatories Vol. 6 No. 3: 229 – 237, SpringerLink, 1990.

[30] M. Grötschel, C. Monma and M. Stoer, *Design of survivable networks*, Handbooks in Operations Research and Management Science, Volume 7: 617-672, Elsevier Science B.V., 1995.

[31] R. Guerin, A. Orda and D. Williams, *QoS routing mechanisms and OSPF extensions*, IEEE/ACM Transactions on Networking, Vol.7 Issue 3: 350 – 364, 1997.

[32] A. Gunnar, M. Johansson and T. Telkamp, *Traffic Matrix Estimation on a large IP Backbone: A Comparison on Real Data*, Proc. of the 4th ACM SIGCOMM conference on Internet measurement**:** 149 - 160, ACM, 2004.

[33] W. Hesen, R. Nagtegaal, P. Kuyper, *"Detail Ontwerp MIPnet"*, version 0.14, KPN intern, 2009.

[34] T. Hsu, *On Four-Connecting a Triconnected Graph,* 33rd Annual Symposium on Foundations of Computer Science: 70-79, 1992.

[35] T. Hsu and V. Ramachandran, *Smallest triconnectivity augmentation Part 1: General graphs,* manuscript, 1994.

[36] T. Hsu and V. Ramachandran, *Smallest triconnectivity augmentation Part 2: Biconnected graphs*, manuscript, 1994.

[37] J. Huessen, *Vision and Roadmap 2009 IP networking*, version 0.1, KPN intern, 2006.

[38] T. Hun and V. Ramachandran, *On finding a Smallest Augmentation to Biconnect a graph*, SIAM J. on Computing, 1993.

[39] B. Jackson and T. Jordan, *A near optimal algorithm for vertex-connectivity augmentation*. In ISAAC '00: Proceedings of the 11th International Conference on Algorithms and Computation: 312-325, 2000.

[40] B. Jackson and T. Jordan, *Independence free graphs and vertex-connectivity augmentation*, J. Comb. Theory Ser. B 94(1): 31-77, 2005.

[41] A. Jamakovic and S. Uhlig, *On the relationship between the algebraic-connectivity and graph's robustness to node and link failures*, 3rd Euro NGI Conference on Next Generation Internet Networks: 96-102, 2007.

[42] A. Jamakovic and P. Van Mieghem, *On the Robustness of Complex Networks by using the Algebraic-connectivity***,** Proc. of the 7th international IFIP-TC6 networking conference on AdHoc and sensor networks, wireless networks, next generation internet, pages 183-194**,** Springer-Verslag, 2008.

[43] J. Jensen, A. Frank and B. Jackson, *Preserving and increasing local edge-connectivity in mixed graphs*, SIAM J. Discrete Math, Vol. 8 No. 2: 155-178, 1995.

[44] J. Jensen and T. Jordan, *Edge-Connectivity Augmentation Preserving Simplicity*, Proc. of the ninth annual ACM-SIAM symposium on Discrete algorithms: 306-315, Society for Industrial and Applied Mathematics, 1997.

[45] T. Jordan, *A note on the vertex-connectivity augmentation problem*, J. Comb. Theory, Ser. B 71(2): 294–301, Academic Press, 1997.

[46] T. Jordán, *On the optimal vertex-connectivity augmentation*, J. Comb. Theory Ser. B 63: 8-20, 1995.

[47] Y. Kajitani and S. Ueno, *The Minimum Augmentation of a Directed Tree to a k-Edge-Connected Directed Graph*, Networks, Volume 16 Issue 2, pages 181-197, 1986.

[48] S. Khuller, *Approximation Algorithms for finding Highly Connected Subgraphs*, University of Maryland, 1995.

[49] S. Khuller and B. Raghavachari, *Improved approximation algorithms for uniform connectivity problems*, Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, pages 1-10, 1995.

[50] S. Khuller and R. Thurimella, *Approximation algorithms for graph augmentation*, Lecture Notes in Computer Science, Volume 623, pages 330-341, 1992.

[51] Y. Kim and M. Mesbahi, *On maximizing the second smallest eigenvalue of a state-dependent graph Laplacian*, Proceedings of the 2005 American Control Conference, pages 99-205, 2005.

[52] S. Kirkland, *A bound on the algebraic connectivity of a graph in terms of the number of cutpoints*, Linear and Multilinear Algebra, Volume 47 Issue 1, pages 93-103, 2000.

[53] S. Kirkland and M. Neumann, *Algebraic connectivity of weighted trees under perturbation*, Linear and Multilinear Algebra, Volume 42 Issue 3, pages 187-203, 1997.

[54] S. Kirkland, M. Neumann and B. Shader, *Characteristic vertices of weighted trees via perron values*, Linear and Multilinear Algebra, Volume 40 Issue 4, pages 311-325, 1996.

[55] G. Kortsarz and Z. Nutov, *Approximating minimum cost connectivity problems*, Handbook on Approximation Algorithms and Metaheuristics, chapter 58, Chapman & Hall/CRC, 2007.

[56] B. Kranendonk, *"Voorraad strategie Mobile IP Backbone"*, version 1, KPN intern 2009.

[57] C.J. Kromjong, *"Care Customer Capacity Management plan"*, version c0.2, KPN intern, 2009.

[58] N. Kruijt, "*Globale Analyse Mobile 10G IP Backbone"*, version 1.1, KPN intern, 2009.

[59] N. Kruijt, *Technical Service Description: Mobile IP Backbone*, version 1.0, KPN intern, 2007.

[60] G. Liberman and Z. Nutov, *On shredders and vertex connectivity augmentation,* Journal of Discrete Algorithms, Volume 5 Issue 1, pages 91-101, 2007.

[61] W. Liu, H. Sirisena, K. Pawlikowski and A. McInnes, *Utility of Algebraic Connectivity Metric in TopologyDesign of Survivable Networks*, seventh International Workshop on the Design of Reliable Communication Networ*ks*, pages 131-138, 2009.

[62] T. Masuzawa, K. Hagihara and N. Tokura, *An optimal time algorithm for the k-vertex-connectivity unweighted augmentation problem for rooted directed trees,* Discrete Applied Mathematics, Volume 17 Issues 1&2, pages 67-105, 1987.

[63] A. Medina, N.Taft, K. Salatian, S. Bhattacharya and C. Diot, *Traffic matrix estimation: existing techniques and new directions***,** Proceedings of the 2002 SIGCOMM conference, pages 161–174, 2002.

[64] R. Merris, *A survey of graph laplacians, Linear and Multilinear Algebra*, Volume 39 Issues 1&2, pages 19-31, 1995

[65] R. Merris, *Characteristic vertices of trees, Linear and Multilinear Algebra*, Volume 22 Issue 2, pages 115-131, 1987.

[66] R. Merris, *Laplacian matrices of graphs: a survey,* Linear Algebra and its Applications, Volumes 197& 198, pages 143-176, 1994.

[67] B. Mohar, *Eigenvalues, diameter and mean distance in graphs*, Graphs and Combinatorics 7, pages 53-64, 1991.

[68] B. Mohar, *Laplace eigenvalues of graphs – a survey*, Discrete Mathematics, Volume 109 Issues 1-3, pages 171-183, 1992.

[69] D. Mosk-Aoyama, *Maximum algebraic connectivity augmentation is NP-hard,* Operations Research Letter**s**, Volume 36 Issue 6, pages 677-679, 2008.

[70] H. Nagamochi, *Graph algorithms for network connectivity problems*, Journal of the Operations Research Society of Japan, Volume 4 Issue 4, pages: 199-223, 2004.

[71] H. Nagamochi and T. Ibaraki, *Algorithmic aspects of graph connectivity*, Cambridge University Press, USA, 2008.

[72] H. Nagamochi and T. Ibaraki, *Graph connectivity and its augmentation: applications of MA orderings,* Discrete Applied Mathematics, Volume 123 Issues 1-3, pages 447-472, 2002.

[73] D. Naor, D. Gusfield and C. Martel, *A fast algorithm for optimally increasing the edge-connectivity*, Siam J. Comput., Vol. 26 No. 4, pages 1139-1165, August 1997.

[74] Z. Nutov, *Approximating connectivity augmentation problems***,** ACM Transactions on Algorithms, Volume 6 Issue 1, article no. 5, 2009.

[75] R. Olfati-Saber, *Ultrafast consensus in small-world networks*, Proceedings of the 2005 American Control Conference, pages 2371-2378, 2005.

[76] M. Penn and H. Krupnik, *Improved Approximation Algorithms for Weighted 2- and 3-Vertex Connectivity Augmentation Problems*, Journal of Algorithms, Volume 22 Issue 1, pages 187-196, 1997.

[77] D. Poole, *Linear algebra: a modern introduction*, Thomas Brooks/Cole, second edition, Canada 2006.

[78] R. Ravi and D.P. Williamson, *An approximation algorithm for minimum-cost vertex-connectivity problems*, Algorithmica, Volume 18 Issue 1, pages 21-43, 1997.

[79] A. Rosenthal and A. Goldner, *Smallest augmentation to biconnect a graph*, SIAM Journal on computing, Volume 6, pages 55-66, 1977.

[80] J. Segovia, E.Calle and P. Vilà, *An Improved Method for Discovering Link Criticality in Transport Networks*, Sixth international conference on Broadband Communications, Networks, and Systems, pages 1-8, IEEE explore digital library, 2009.

[81] S. Taoka and T. Watanabe, *Maximum weight matching-based algorithms for k-edge-connectivity augmentation of a graph*, IEEE International Symposium on Circuits and systems, pages 2231 – 2234, 2005.

[82] P. Van Mieghem, *Data Communications Networking,* Amsterdam: Techne Press, Amsterdam, 2006.

[83] L. Végh, *Augmenting undirected node-connectivity by one*, Proceedings of the 42nd ACM symposium on Theory of computing, pages 563-572, 2010.

[84] H. Wang, *Robustness of Networks*, PhD. thesis, Faculty of Electrical engineering, Mathematics and Computer Science, Delft University of Technology, 2009.

[85] T. Watanabe and A. Nakamura, *3-connectivity augmentation problems,* IEEE International Symposium on Circuits and systems, pages 1847-1850, 1988.

[86] T. Watanabe and A. Nakamura, *A minimum 3-connectivity augmentation of a graph,* Journal of Computer and System Sciences, Volume 46 Issue 1, pages 91-128, 1993.

[87] T. Watanabe, *An efficient augmentation to k-edge connected graph,* Tech. report C-23, Dept. of Applied Math., Hiroshima University, 1988.

[88] T. Watanabe and A. Nakamura, *Edge-connectivity augmentation problems, Journal of Computer and System Sciences*, Volume 35 Issue 1, pages 96-144, 1987.

[89] T. Watanabe and A. Nakamura, *On a smallest augmentation to k-edge connect a graph*, *Technical Report C-20*, Department of Applied Math., Faculty of Engineering, Hiroshima University, Japan, 1984.

[90] C. Wu, *Algebraic connectivity of directed graphs*, Linear and Multilinear Algebra, Volume 53 Issue 3, pages 203-223, 2005.

Websites:

[w1] D. Gleich, http://www.mathworks.com/matlabcentral/fileexchange/10922, 30 April 2006. (Downloaded in March 2010)

Appendix A: List of abbreviations and Symbols

Abbreviations	
ASW	Access Switch
BB	Backbone
BOG	Border Gateway
CNC	Cycle-type Normal Cactus
CR	Core Router
CS	Circuit Switched
CSW	Core Switch
CTA	Capacity and Traffic Aware
DFS	Depth First Search
ES	Extreme Sets partition
EST	Extreme Sets Tree
FR	Function Router
GGSN	Gateway GPRS Support Node
GRX	GPRS Roaming Exchange
GSM	Global System for Mobile Communications
IE	Intelligent Edge
IP	Internet Protocol
ITR	Internet Traffic Router
MDX	MMS Domestic Exchange
MIPnet	Mobile IP Network
MOO	Mobile Office Online
OSPF	Open Shortest Path First
PoP	Point of Presence
PS	Packet Switched
SGSN	Serving GPRS Support Node
SLA	Service Level Agreement
SMS	Short Message Service
SRIPS	Service Router Intrusion Prevention System
SRLB	Service Router Load Balancer
STAR	STAR Router
TR	Traffic Router
UMTS	Universal Mobile Telecommunications System
VRF	Virtual Routing Function
VRFLR	VRF Lite Router
ZR	Zone Router

Symbols	
G	A graph (of a network)
$V, V(G)$	Vertex set of a graph
$E, E(G)$	Edge set of a graph
$\|V\|, n$	Number of vertices
$\|E\|, m$	Number of edges
μ	Laplacian eigenvalue
$\|W\|$	Number of nodes in cactus
$\|F\|$	Number of links in cactus
$R, H(G)$	Cactus graph
$W, W(G)$	Vertex set of a cactus
$F, F(G)$	Edge set of a cactus
φ	Mapping of vertices to nodes
k	(edge) connectivity
λ	Min cut value
δ	Edge augmentation value
η	Vertex augmentation value
$a(G)$	Algebraic-connectivity
$\kappa_e(G)$	Edge-connectivity
$\kappa_v(G)$	Vertex-connectivity
$\boldsymbol{b}$	Fiedler vector
$d(U)$	Degree of set U (U can be a singleton)
$\Gamma(v)$	Neighbor set of v
CMC_{Ak}	Circular minimum cut partition for segment A_k
MC_{Bj}	Minimum cut partition for segment B_j
$\Phi(x)$	Edge demand function
$t(G)$	Number of tight sets
$b(G)$	Number of clusters after removing separator S
C_B	Betweenness Centrality
τ	Number of shortest paths
π	Parent list towards source
D	Hopcount towards source
M	Largest minimum capacity
Q	Priority queue
S	Stack
β	Tuning parameter
p	Transmission probability
$VC(v)$	Vertex criticality of vertex v
$\zeta(s,t)$	Cubic number of pairwise dependencies
$\xi(u,v)$	Weight of edge{u,v}
$dist[v]$	Distance vector of v
$Pred[v]$	Predecessor list of v towards source
$RI(v)$	Relative importance of v
$\Omega(v)$	Relative importance factor of v
$SC(v)$	Size of cloud connected to v
$\omega(v)$	Size of cloud factor of v

Appendix B: Network drawings and additional information

B1 Drawings of the PS mobile core network

Figure B1 shows what the PS mobile network looks like according to the near future design, consisting of the Intelligent Edge network and the Mobile 10G IP BB. Each element is labeled with its own network name, according to the terminology by KPN personnel in their documentation. The core PoPs of ASD and RT are the largest subnets, containing most of the functionality and access to external networks. Core PoP AH is not in place yet, but is being implemented at the time of writing, while core PoP GV is most likely going to be phased out in the future. The network can be divided into the following subnets:

1. The mobile 10G IP BB
2. 4 core PoP locations
3. 4 larger VRF lite locations
4. 1 smaller VRF lite location, which is used to establish an interconnection with Telfort.

Figure B2 displays the network as a graph, where the circular vertices are routers and switches, while the rectangular vertices are elements performing specialized functions. Figure B3 is the graph derived out of figure B2, where the clouds and the specialized vertices are relaxed. It shows the capacity of the edges in terms of Gbps. This so called relaxed graph is used as the case study and therefore every vertex is labeled with a number. The mapping can be verified by comparing figure B3 with figure B1.

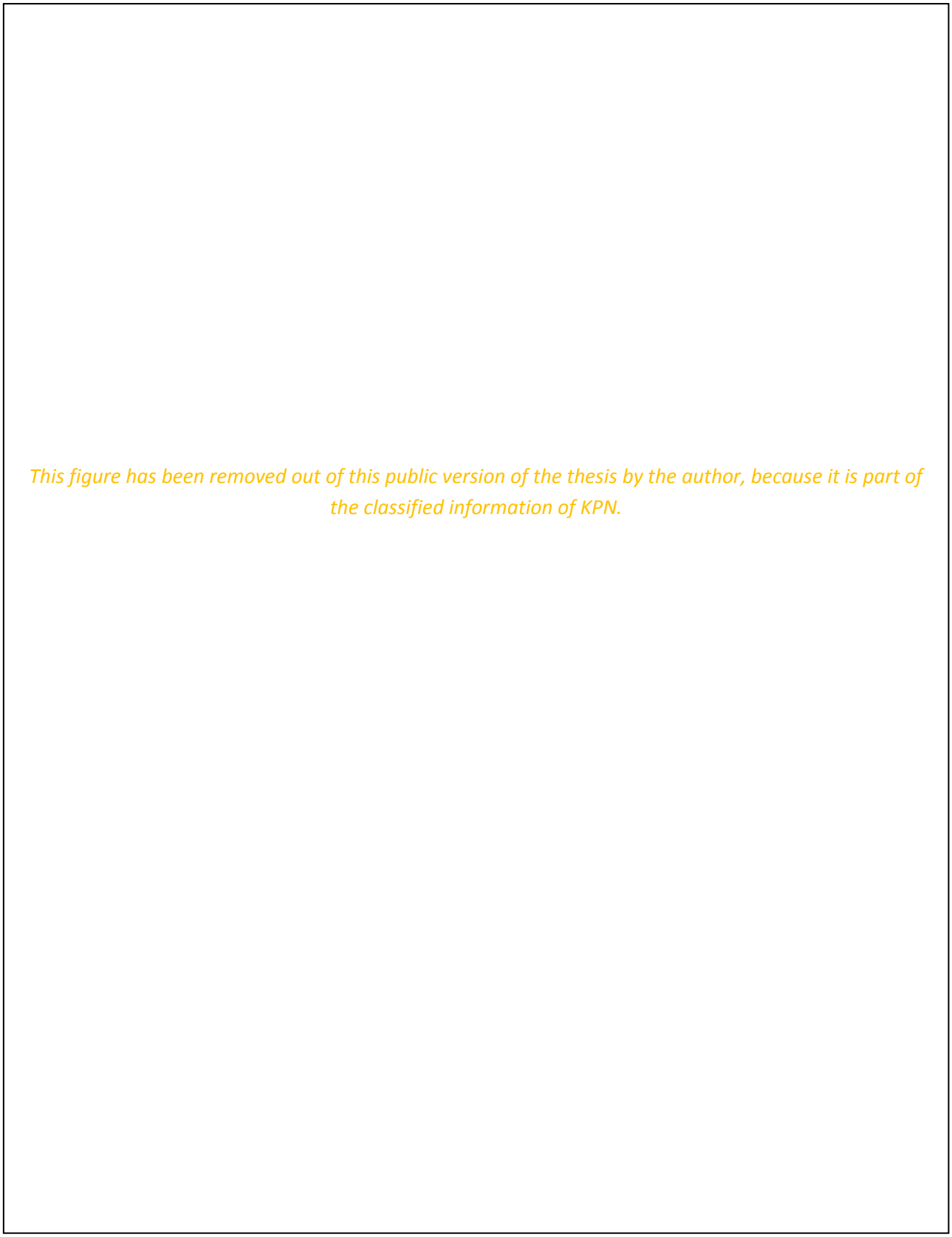

Figure B1: The PS mobile core network of KPN

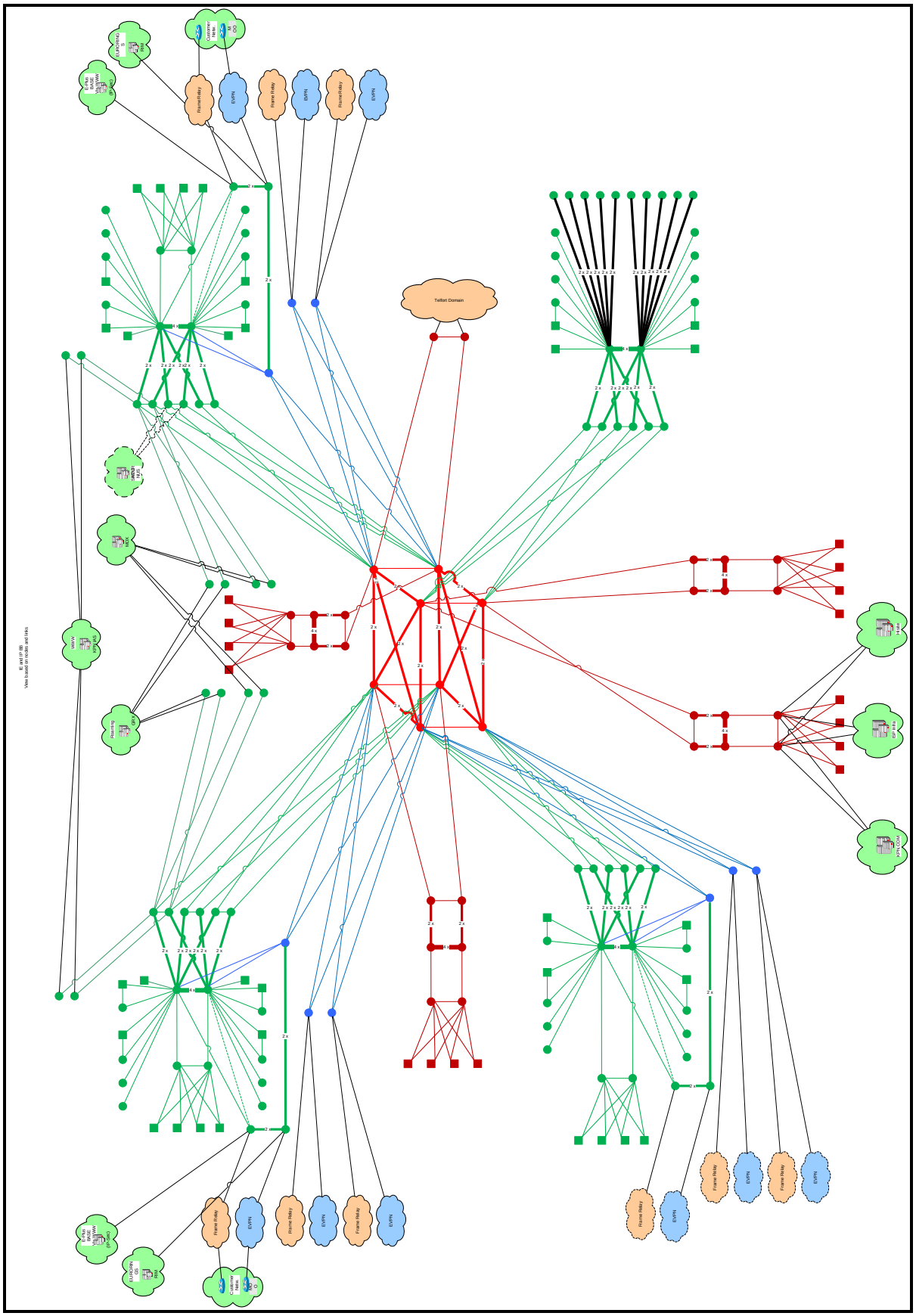

Figure B2: Graph of the PS mobile core network of KPN.

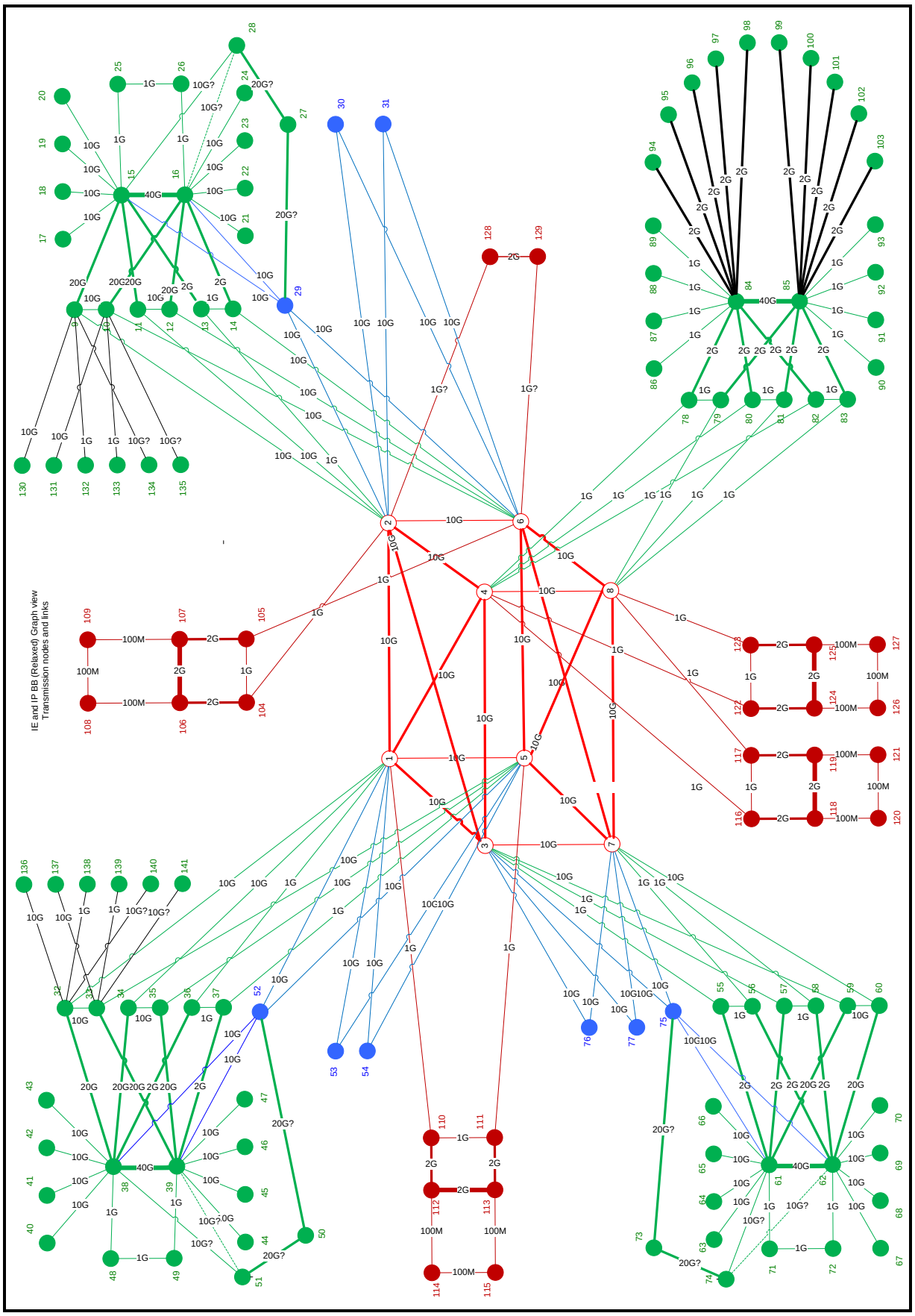

Figure B3: Relaxed graph of the PS mobile core network of KPN with edge capacities.

B2 Results regarding the increase of the algebraic-connectivity

After applying any of the 3 suboptimal strategies explained in section 3.2, additional edges will be added to the original graph of core PoP ASD/RT to increase the algebraic-connectivity and therefore the robustness. It is interesting to see how many edges are required for each strategy in increasing $a(G)$ such that the graph is at least 1-connected. Figure B4 till figure B6 show these results

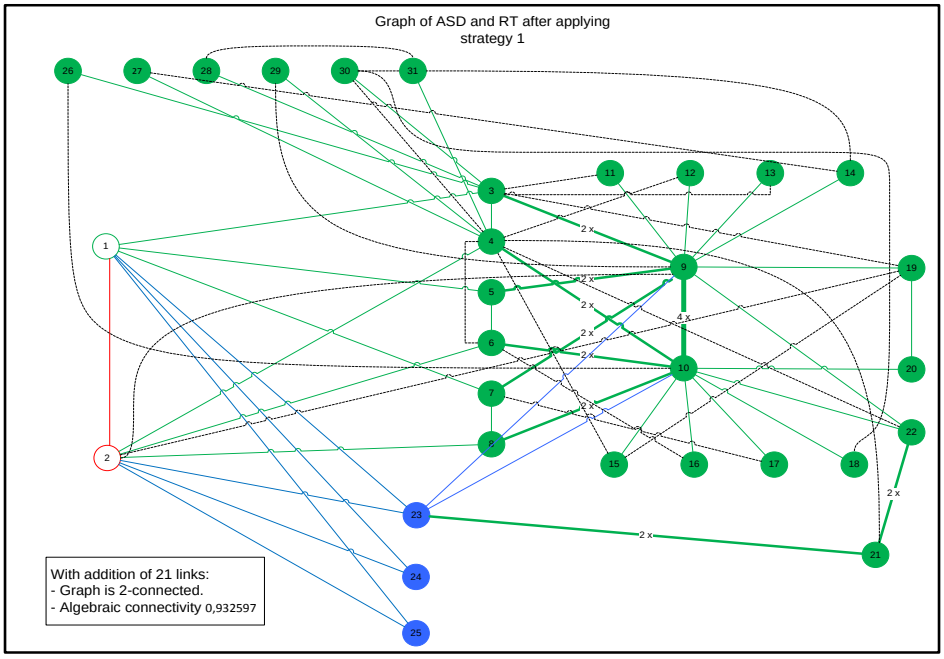

Figure B4: Graph of ASD/RT after applying strategy 1.

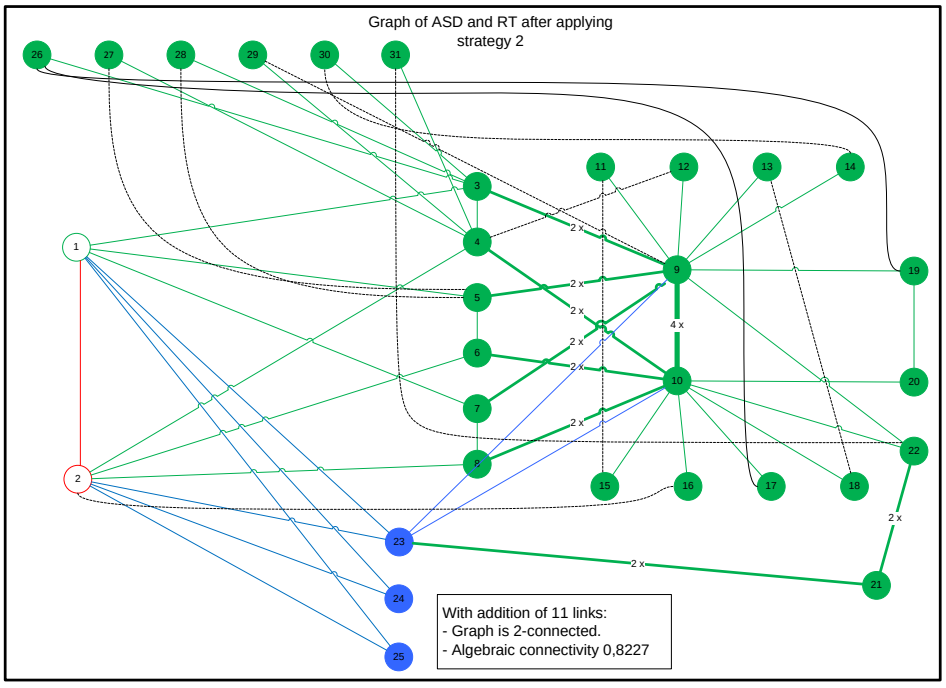

Figure B5: Graph of ASD/RT after applying strategy 2.

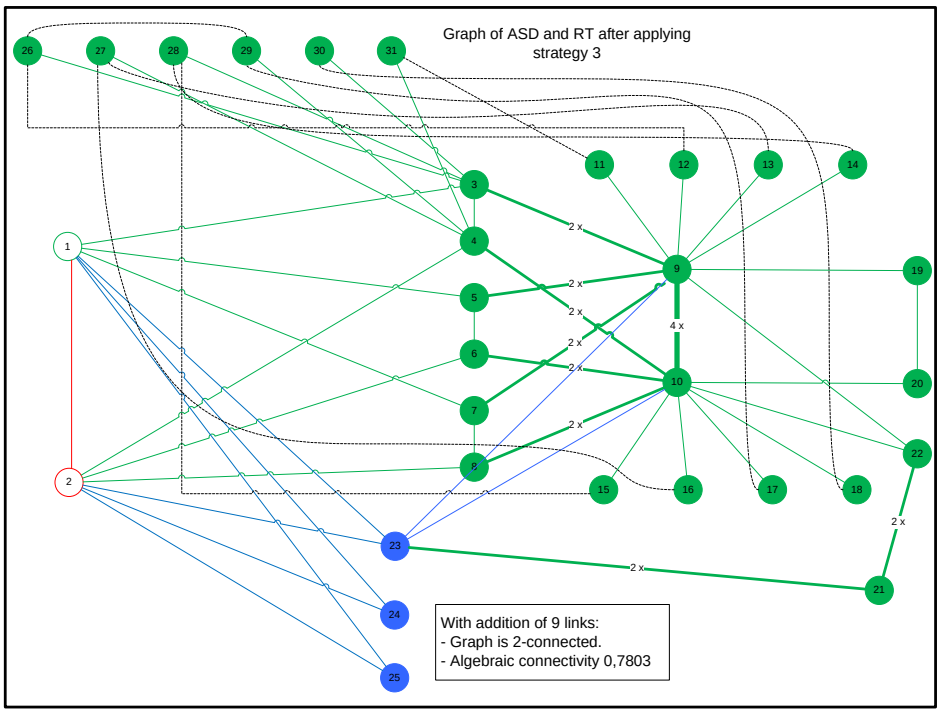

Figure B6: Graph of ASD/RT after applying strategy 3.

B3 Drawing results regarding edge and vertex-augmentation analysis

This section contains figure B7, which is the resulting 2-edge-connected complete graph after algorithm 3.12 (see section 3.3) is applied to the original complete graph. It also contains figures B8 and B9, which are the process of applying and the result of algorithm 3.13 respectively (see section 3.4). The analysis is based on the graph of the mobile core network shown in figure B3.

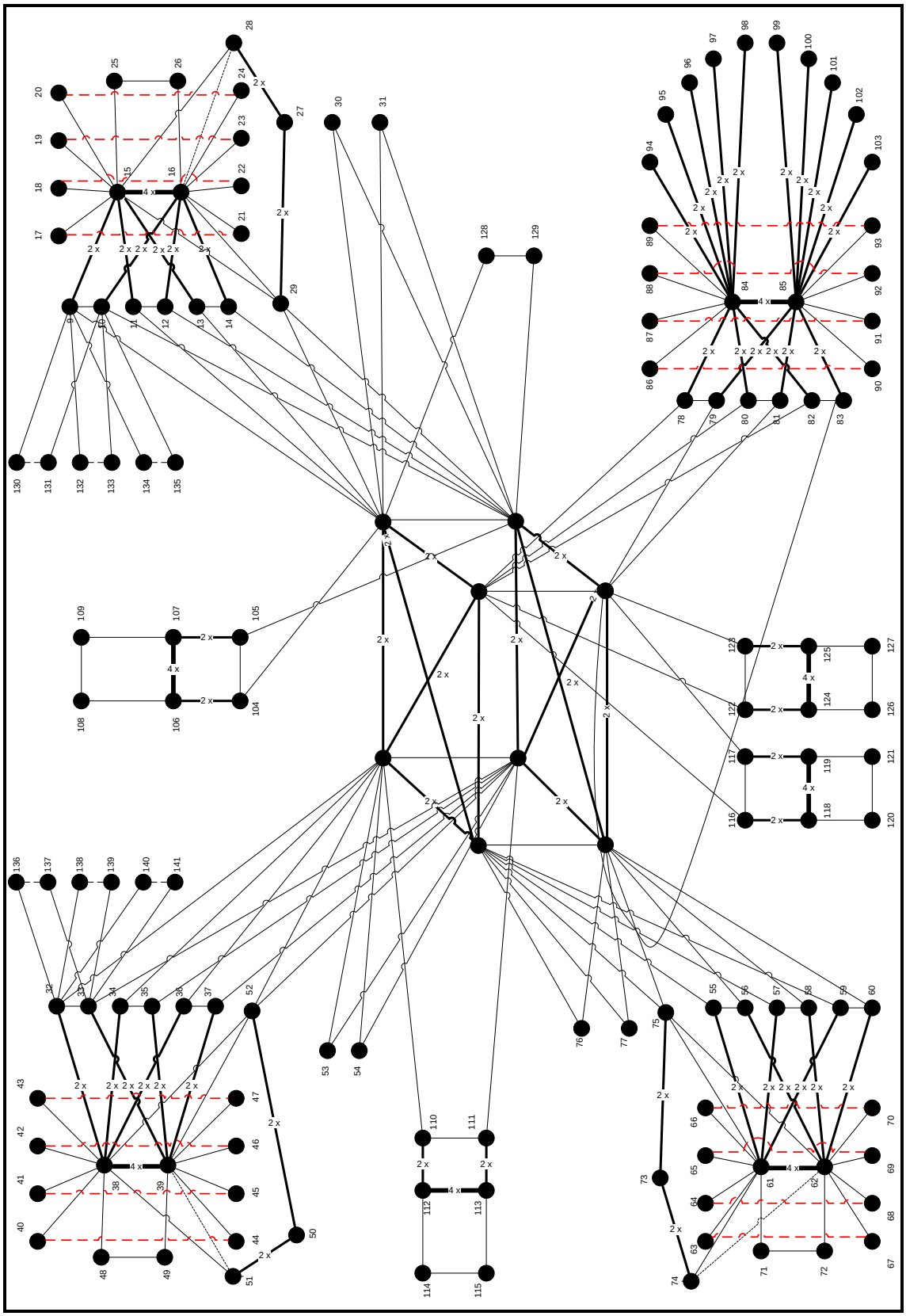

Figure B7: The complete graph after applying Algorithm 3.12.

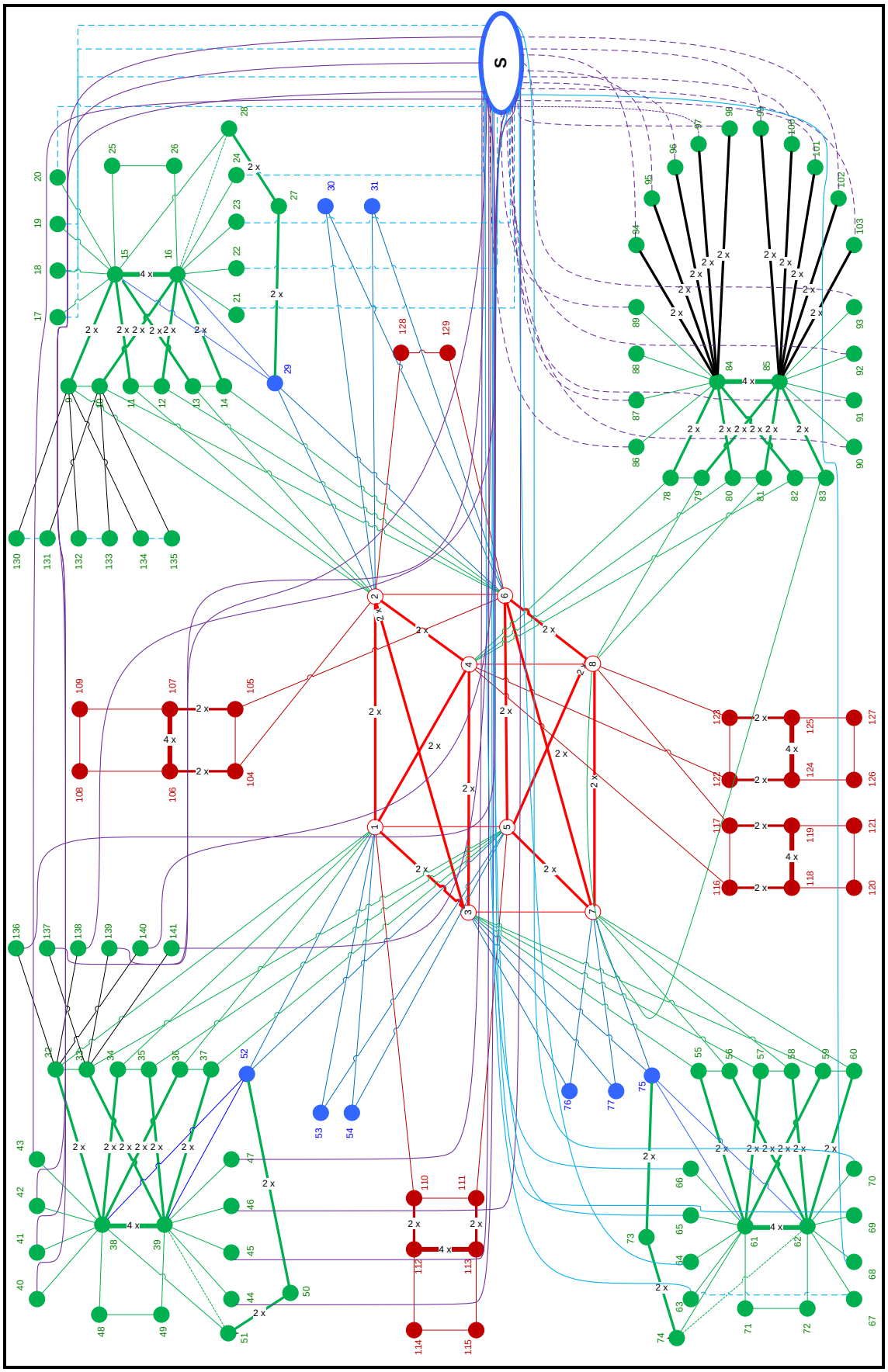

Figure B8: Applying Algorithm 3.13 for augmenting the vertex-connectivity.

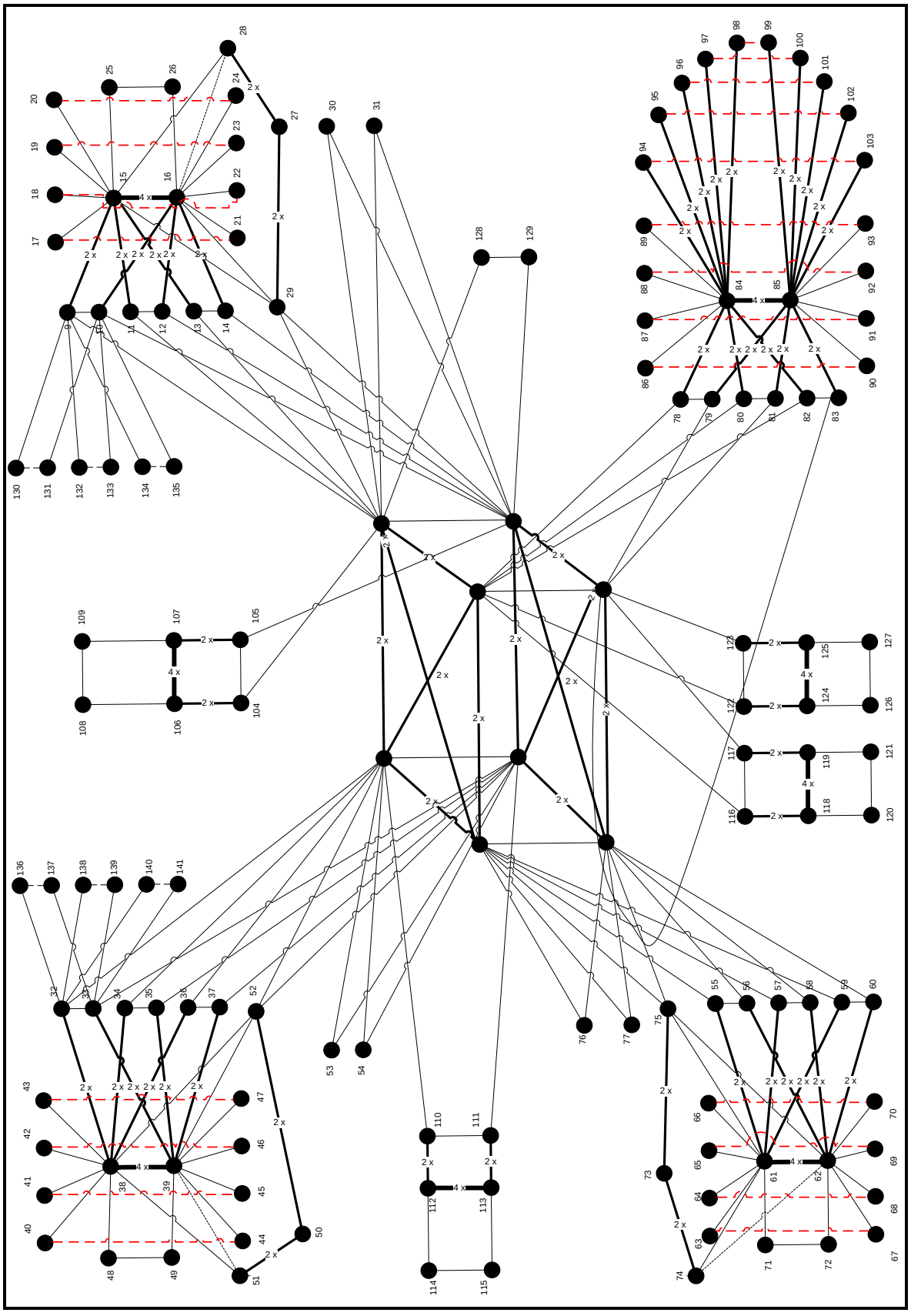

Figure B9: The complete graph after applying Algorithm 3.13.

Appendix C: The structure of MATLAB programs

All the programming work can be found in the KPN deliverable. Due to space limitations, only the structure of the programs consisting of multiple m files is treated in this appendix.

C1 Structure of code for strategy 1 – 3 for increasing a(G)

Figure C1 indicates how the source code for strategy 1 for calculating $a(G)$ works. A main routine is used to call 4 subroutines for a predefined number of iterations. The main routine also has the part of the code used for storage of the results. The 1st subroutine calculates L and its 2nd smallest eigenvalue. After that the same is done for each added edge and these results are stored in an eigenvalue matrix (The E matrix). This matrix stores $a(G)$ for all possible edges to be added. The 2nd subroutine (optional) displays and stores the E matrix in a figure. The 3rd one is used to find the new edge that increases $a(G)$ the most. Finally, the 4th is used to modify A and D, accordingly.

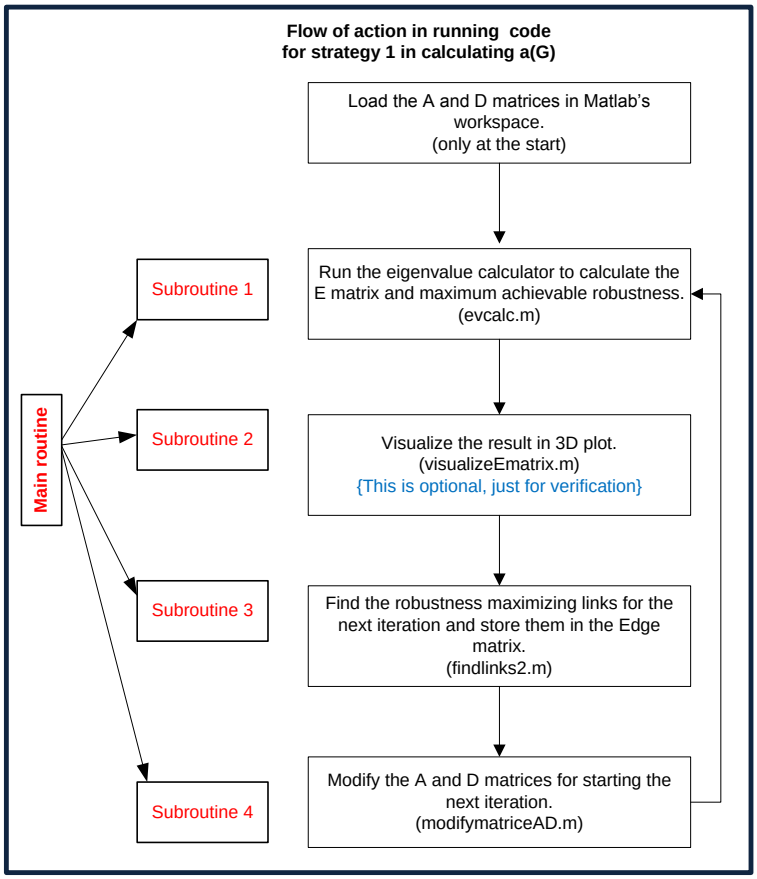

Figure C1: Flow of actions when running code of strategy 1 for calculating a(G).

Figure C2 shows the code structure of strategy 2. The main routine calls the three subroutines according to a predefined number of iterations and stores the results for each iteration. The 1st subroutine sequentially adds a new edge between a vertex of minimum degree and every other vertex, not having an edge already. All possibilities are tried out and the edge resulting in maximum

robustness gain is stored by the 2nd subroutine. If a tie occurs the first edge is chosen. The 3rd subroutine modifies A and D, according to this new edge, before the next iteration starts.

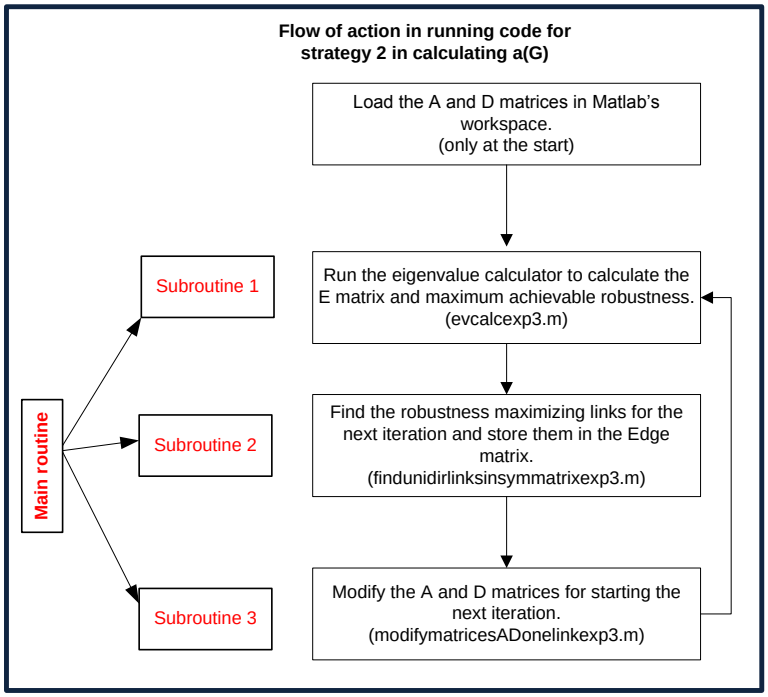

Figure C2: Flow of actions when running code of strategy 2 for calculating a(G).

The flowchart of strategy 3 is presented in figure C3. This strategy uses the Fiedler vector to find vertices i and j, between which the new edge should be added. At the end of each iteration L is modified so that the next iteration can start based on the new network, consisting of the original network plus all the edges added in the previous iterations.

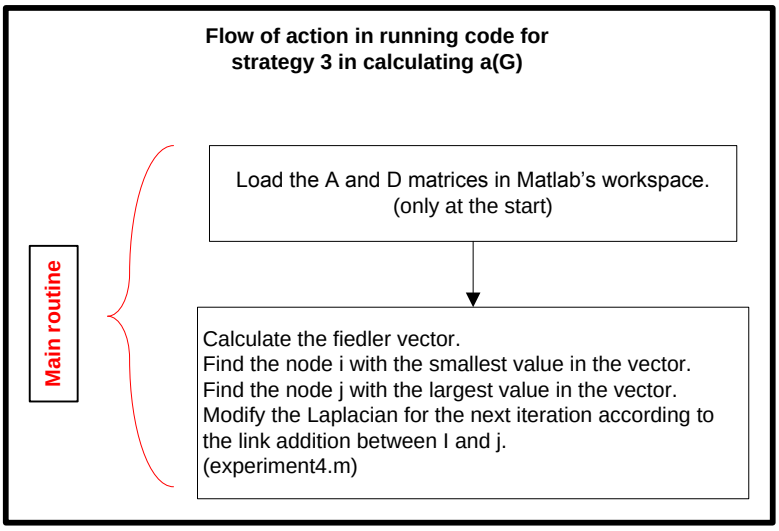

Figure C3: Flow of actions when running code of strategy 3 for calculating a(G).

C2 Code structure for CTA-edge-betweenness program

The code for the CTA-edge-betweenness program consists of a main and a subroutine as shown in figure C4. The subroutine calculates a predecessor list from each source vertex s to every other vertex. The main routine calculates the edge-betweenness centrality $\forall e \in E$.

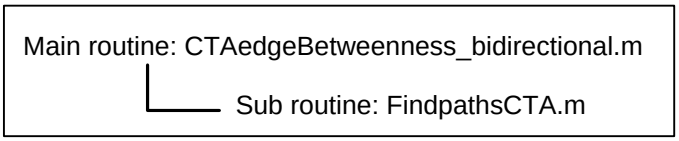

Figure C4: Code structure for CTA-edge-Betweenness program.

C3 Matlab code structure for cactus construction sub algorithms

The structures of the"*st*-MC-partition", "Update *st*-MC-partition" and "Construct st-cactus-representation" algorithms are shown in figures C5, C6 and C7 respectively. The m files of each algorithm can be found on the CD handed in along with this thesis. On this disk the same hierarchy as in the figure is used for storing the code.

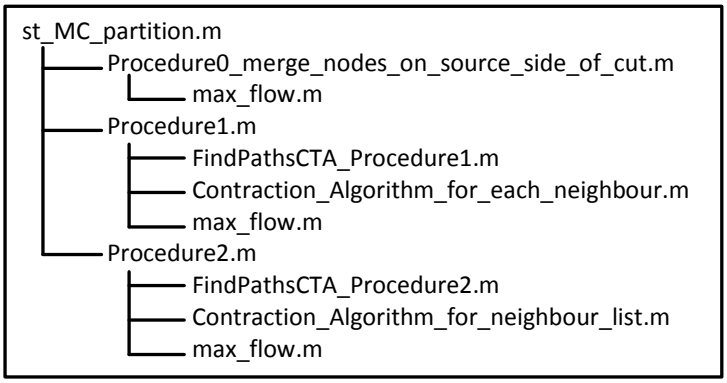

Figure C5: Hierarchy of the programs for computing st-MC-partition.

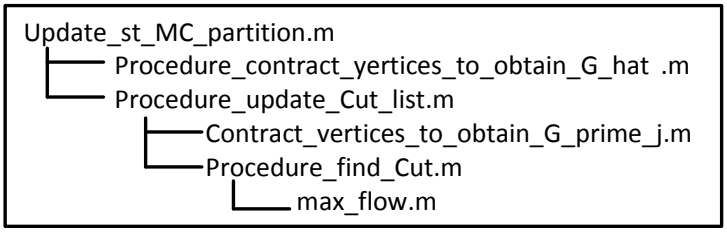

Figure C6: Hierarchy of the programs for updating st-MC-partition.

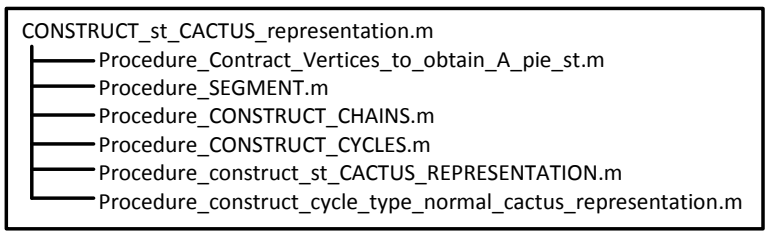

Figure C7: Hierarchy of the programs for constructing st-cactus-representation.

Appendix D: Algebraic-connectivity results in table format

The edge column of each strategy (in table D1) represents the edge to be added, before the corresponding $a(G)$ is achieved. The labels used for the edges correspond to the vertex labels of figure B3. Adding the edges of the table to this figure (for each strategy) would provide a graphical view of the results.

	Strategy 1		Strategy 2		Strategy 3	
# of edges	a(G)	Edge	a(G)	Edges	a(G)	Edge
0	0.4087468	none	0.408747	none	0.4087468	none
1	0.4524422	{3,19}	0.448401	{26,19}	0.4368959	{31,11}
2	0.5096773	{4,12}	0.534336	{31,22}	0.5201419	{30,18}
3	0.5284789	{10,26}	0.550922	{28,5}	0.5512031	{14,28}
4	0.6115688	{9,29}	0.587513	{12,4}	0.6180828	{16,27}
5	0.6359319	{7,27}	0.622456	{16,2}	0.6478366	{26,29}
6	0.6399762	{4,21}	0.631358	{27,5}	0.6691836	{29,17}
7	0.6477021	{2,19}	0.701756	{17,26}	0.7460621	{27,13}
8	0.6508744	{4,6}	0.70871	{11,15}	0.7616127	{26,12}
9	0.6524729	{6,16}	0.73472	{14,30}	0.7803458	{28,15}
10	0.6644186	{3,11}	0.779633	{29,9}	0.8070166	{31,25}
11	0.7087881	{28,31}	0.822694	{13,18}	0.8978325	{24,30}
12	0.7164941	{4,15}	0.83054	{24,9}	0.9120071	{20,25}
13	0.7549945	{3,13}	0.886547	{14,15}	0.9135502	{28,27}
14	0.7735963	{4,30}	0.935896	{25,19}	0.9537357	{21,26}
15	0.8011487	{14,31}	0.936076	{18,26}	1.0127242	{24,13}
16	0.8570022	{18,30}	0.999269	{30,22}	1.0299458	{11,15}
17	0.8587648	{15,19}	1.03886	{12,26}	1.0794431	{19,17}
18	0.8662657	{30,31}	1.079436	{11,1}	1.1190373	{12,18}
19	0.8877258	{14,27}	1.164342	{31,18}	1.2233203	{21,14}
20	0.9189137	(2,9}	1.223385	{13,28}	1.2339375	{20,18}
21	0.9325968	(4,12)	1.243819	{21,7}	1.2665997	{17,16}
22	1.0835473	{1,31}	1.247766	{16,27}	1.3407325	{29,7}
23	1.2246554	{7,14}	1.324291	{20,15}	1.3942492	{15,30}
24	1.2943092	{25,28}	1.338533	{17,2}	1.4648112	{31,14}
25	1.3736695	{1,16}	1.39737	{29,14}	1.4888425	{28,6}
26	1.3783159	{5,14}	1.427592	{13,10}	1.5649382	{11,17}
27	1.4048595	{16,24}	1.516332	{16,14}	1.6490803	{12,13}
28	1.4116257	{3,15}	1.582306	{28,1}	1.7355115	{16,5}
29	1.4205709	{15,23}	1.593227	{20,16}	1.8038378	{21,19}
30	1.4259212	{14,24}	1.62774	{30,7}	1.8215658	{22,24}

Table D1: Increase algebraic-connectivity of core PoP ASD/RT.

Figure D1 shows how the algebraic connectivity increases, when adding edges to the entire graph, for the 3 strategies.

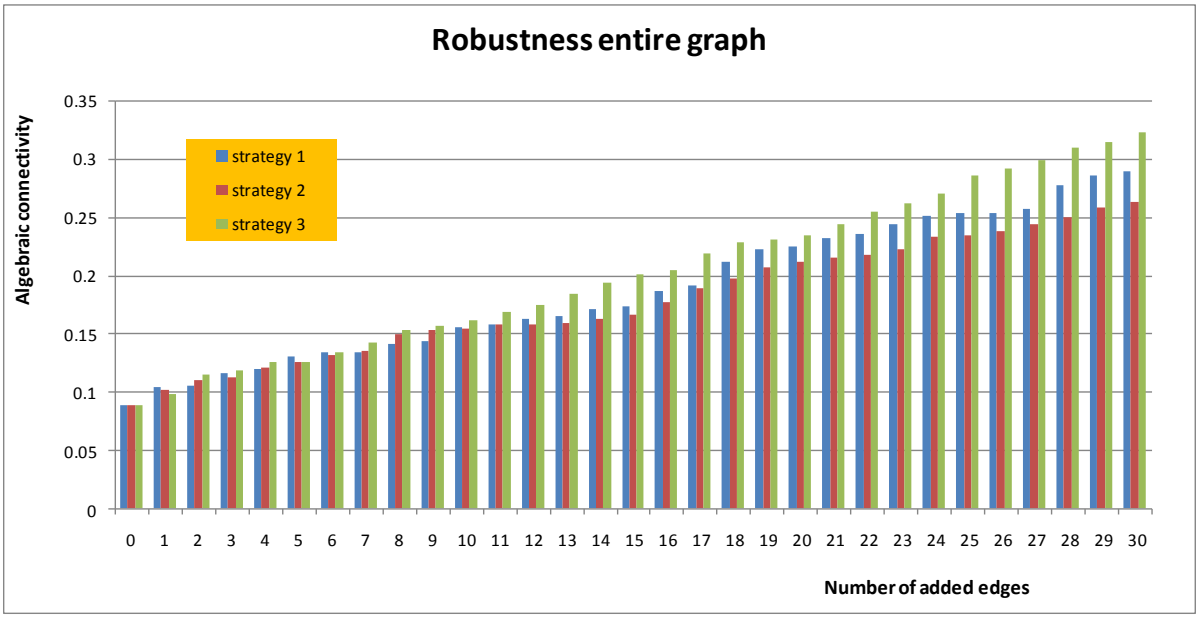

Figure D1: Increasing the Algebraic-connectivity of the entire graph.

Appendix E: Detailed results capacity management

Table E1 shows the most important parameters to be management within KPN's capacity management project.

Capacity parameters for elements (vertices)		Capacity parameters for connections (edges)	
1	Processing power (CPU) => processing delay	1	Available bandwidth (Installed capacity per edge)
2	Memory usage (Ram)	2	Delay (propagation and transmission delay)
3	Buffering space => buffering delay	3	Jitter
4	Disk space		
5	Simultaneously attached users (SAUs)		

Table E1: Parameters for capacity management.

Capacity management is a hot issue, because mobile data traffic is growing at an exponential rate, as shown in figure E1. This exponential growth is expected to continue further as more customers make use of services like blackberry and machine to machine communication.

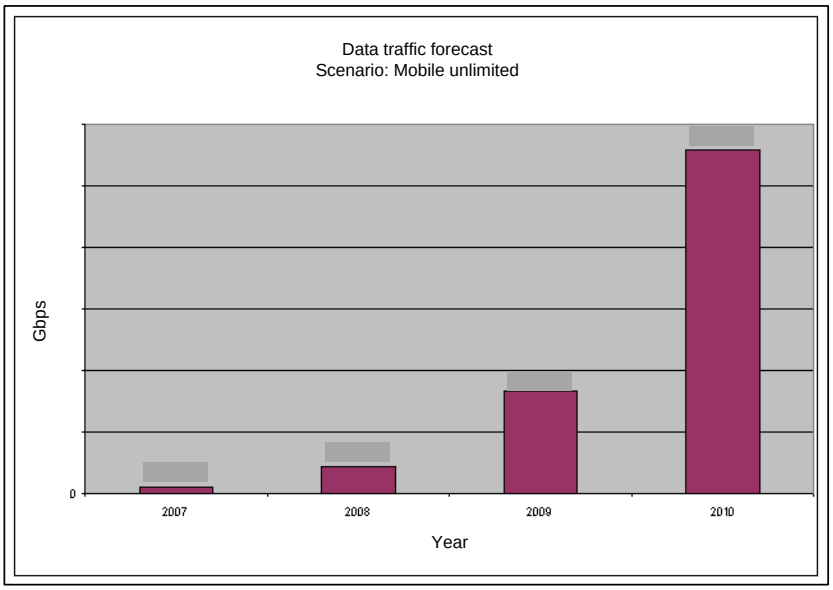

Figure E1: Capacity forecast until 2010.

				0.1 Mb		1 Mb		10 Mb		100 Mb	
#	u	v	ABW	Rel_RBW	Bottleneck	Rel_RBW	Bottleneck	Rel_RBW	Bottleneck	Rel_RBW	Bottleneck
1	1	2	10	0.99557	1	0.9557	1	0.557	1	-3.43	3
2	1	3	10	0.9975	1	0.975	1	0.75	1	-1.5	3
3	2	3	10	0.99725	1	0.9725	1	0.725	1	-1.75	3
4	1	4	10	0.99549	1	0.9549	1	0.549	1	-3.51	3
5	2	4	10	0.99505	1	0.9505	1	0.505	1	-3.95	3
6	3	4	10	0.99725	1	0.9725	1	0.725	1	-1.75	3
7	1	5	10	1	1	1	1	1	1	1	1
8	2	6	10	0.9994	1	0.994	1	0.94	1	0.4	2
9	5	6	10	0.99317	1	0.9317	1	0.317	2	-5.83	3
10	3	7	10	1	1	1	1	1	1	1	1
11	5	7	10	0.9956	1	0.956	1	0.56	1	-3.4	3
12	6	7	10	0.99525	1	0.9525	1	0.525	1	-3.75	3
13	4	8	10	1	1	1	1	1	1	1	1
14	5	8	10	0.99439	1	0.9439	1	0.439	1	-4.61	3
15	6	8	10	0.99395	1	0.9395	1	0.395	2	-5.05	3
16	7	8	10	0.99615	1	0.9615	1	0.615	1	-2.85	3
17	2	9	10	0.99025	1	0.9025	1	0.025	2	-8.75	3
18	6	10	10	0.98821	1	0.8821	1	-0.179	3	-10.79	3
19	9	10	10	0.99773	1	0.9773	1	0.773	1	-1.27	3
20	2	11	1	1	1	1	1	1	1	1	1
21	6	12	1	1	1	1	1	1	1	1	1
22	11	12	1	1	1	1	1	1	1	1	1
23	2	13	1	1	1	1	1	1	1	1	1
24	6	14	1	1	1	1	1	1	1	1	1
25	13	14	1	1	1	1	1	1	1	1	1
26	9	15	20	0.99652	1	0.9652	1	0.652	1	-2.48	3
27	11	15	2	1	1	1	1	1	1	1	1
28	13	15	2	1	1	1	1	1	1	1	1
29	10	16	20	0.99577	1	0.9577	1	0.577	1	-3.23	3
30	12	16	2	1	1	1	1	1	1	1	1

Table E2: Output CTAedgeBetweenness according to uniform distributed TM.

This table presents an incomplete output of the CTA edge-betweenness program, according to the linear rule, where the input (in this case) is a traffic matrix, constructed using uniform packet sizes. The 2nd and 3rd column show the vertices connected by a bidirectional edge. Rel_RBW stands for relative remaining bandwidth.

Label	Name	Location	BC	ω(v)	Ω(v)	VC(v)	Label	Name	Location	BC	ω(v)	Ω(v)	VC(v)
30			280	1	1	9736	133			280	0.5	0.5	5008
31			280	1	1	9736	134			280	0.5	0.5	5008
53			280	1	1	9736	135			280	0.5	0.5	5008
54			280	1	1	9736	138			280	0.5	0.5	5008
76			280	1	1	9736	139			280	0.5	0.5	5008
77			280	1	1	9736	140			280	0.5	0.5	5008
75			538	0	1.5	7630	141			280	0.5	0.5	5008
29			532	0	1.5	7624	59			2320	0	0.5	4684
52			532	0	1.5	7624	60			2320	0	0.5	4684
28			304	1	0.5	7396	78			1047	0	0.5	3411
51			304	1	0.5	7396	79			1047	0	0.5	3411
74			298	1	0.5	7390	80			1047	0	0.5	3411
27			282	1	0.5	7374	81			1047	0	0.5	3411
50			282	1	0.5	7374	82			1047	0	0.5	3411
73			282	1	0.5	7374	83			1047	0	0.5	3411
130			280	1	0.5	7372	104			820	0	0.5	3184
131			280	1	0.5	7372	105			820	0	0.5	3184
136			280	1	0.5	7372	110			820	0	0.5	3184
137			280	1	0.5	7372	111			820	0	0.5	3184
4			4728	0	0.5	7092	116			820	0	0.5	3184
8			4728	0	0.5	7092	117			820	0	0.5	3184
2			4587	0	0.5	6951	122			820	0	0.5	3184
6			4587	0	0.5	6951	123			820	0	0.5	3184
1			4355	0	0.5	6719	106			560	0	0.5	2924
5			4355	0	0.5	6719	107			560	0	0.5	2924
84			3623	0	0.5	5987	112			560	0	0.5	2924
85			3623	0	0.5	5987	113			560	0	0.5	2924
3			3047	0	0.5	5411	118			560	0	0.5	2924
7			3047	0	0.5	5411	119			560	0	0.5	2924
9			3046	0	0.5	5410	124			560	0	0.5	2924
10			3046	0	0.5	5410	125			560	0	0.5	2924
32			3046	0	0.5	5410	11			280	0	0.5	2644
33			3046	0	0.5	5410	12			280	0	0.5	2644
15			2963	0	0.5	5327	13			280	0	0.5	2644
16			2963	0	0.5	5327	14			280	0	0.5	2644
38			2963	0	0.5	5327	34			280	0	0.5	2644
39			2963	0	0.5	5327	35			280	0	0.5	2644
61			2951	0	0.5	5315	36			280	0	0.5	2644
62			2951	0	0.5	5315	37			280	0	0.5	2644
132			280	0.5	0.5	5008	55			280	0	0.5	2644

Table E3: Sample result for vertex criticality (part 1).

Label	Name	Location	BC	ω(v)	Ω(v)	VC(v)	Label	Name	Location	BC	ω(v)	Ω(v)	VC(v)
56			280	0	0.5	2644	67			280	0	0	280
57			280	0	0.5	2644	68			280	0	0	280
58			280	0	0.5	2644	69			280	0	0	280
120			280	0.5	0	2644	70			280	0	0	280
121			280	0.5	0	2644	71			280	0	0	280
128			280	0	0.5	2644	72			280	0	0	280
129			280	0	0.5	2644	86			280	0	0	280
17			280	0	0	280	87			280	0	0	280
18			280	0	0	280	88			280	0	0	280
19			280	0	0	280	89			280	0	0	280
20			280	0	0	280	90			280	0	0	280
21			280	0	0	280	91			280	0	0	280
22			280	0	0	280	92			280	0	0	280
23			280	0	0	280	93			280	0	0	280
24			280	0	0	280	94			280	0	0	280
25			280	0	0	280	95			280	0	0	280
26			280	0	0	280	96			280	0	0	280
40			280	0	0	280	97			280	0	0	280
41			280	0	0	280	98			280	0	0	280
42			280	0	0	280	99			280	0	0	280
43			280	0	0	280	100			280	0	0	280
44			280	0	0	280	101			280	0	0	280
45			280	0	0	280	102			280	0	0	280
46			280	0	0	280	103			280	0	0	280
47			280	0	0	280	108			280	0	0	280
48			280	0	0	280	109			280	0	0	280
49			280	0	0	280	114			280	0	0	280
63			280	0	0	280	115			280	0	0	280
64			280	0	0	280	126			280	0	0	280
65			280	0	0	280	127			280	0	0	280
66			280	0	0	280							

Table E4: Sample result for vertex criticality (part 2).