FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Clustering Heuristics for the Hierarchical Ring Network Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Rainer Schuster
Matrikelnummer 0425205

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl
Mitwirkung: Univ.-Ass. Dipl.-Ing. Christian Schauer

Wien, 30.11.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Clustering Heuristics for the Hierarchical Ring Network Problem

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Rainer Schuster

Registration Number 0425205

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl
Assistance: Univ.-Ass. Dipl.-Ing. Christian Schauer

Vienna, 30.11.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Rainer Schuster
Zubergasse 147, 2020 Sonnberg

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I want to thank my advisors ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl and Univ.-Ass. Dipl.-Ing. Christian Schauer. Their constructive feedback and their experience was a big help for writing and improving this thesis.

Special thanks goes to my family and my friends. Studying can be quite time-consuming and stressful sometimes and their support is invaluable.

Abstract

In this thesis the application of clustering algorithms for solving the Hierarchical Ring Network Problem (HRNP) is investigated.

When the network is represented as a graph, an informal problem definition for this NP-complete problem is: Given a set of network sites (nodes) assigned to one of three layers and the costs for establishing connections between sites (i.e., edge costs) the objective is to find a minimum cost connected network under certain constraints that are explained in detail in the thesis. The most important constraint is that the nodes have to be assigned to rings of bounded size that connect the layers hierarchically.

The ring structure is a good compromise between the robustness of a network and the cost for establishing it. It is guaranteed, that the network can continue to provide its service if one network node per ring fails.

The basic idea in this thesis for solving this network design problem was to cluster the sites with hierarchical clustering heuristics and to use the resulting hierarchy as support for the ring-finding heuristics. Previous apporaches for related network design problems did not use the inherent network structure in such a way. Usual approaches are based on greedy heuristics.

Three clustering heuristics were implemented: Girvan-Newman, K-means and Kernighan-Lin. Especially the first algorithm is interesting, because it was successfully applied analyzing large network structures, also in the context of internet communities.

For finding rings three heuristics were implemented too. Strategic variation of the maximum allowed ring size helps the first heuristic to find rings using the cluster hierarchy. The second heuristic finds rings by searching for paths that are connected to previously found rings. Third a repair heuristic was implemented that tries to add remaining nodes to existing rings.

Local search heuristics are applied last to improve the solution quality.

To check how the clustering approach performs for solving the problem of this thesis two test instance generators were implemented. One generates instances randomly and the second generates instances based on the popular TSPLIB archive.

The evaluation of the random test instances has shown, that all three clustering heuristics were able to solve those test instances, while Girvan-Newman and Kernighan-Lin found valid solutions in each test run this was not possible for K-means. When Kernighan-Lin was used as clustering algorithm solutions could be found faster on average, but the resulting costs where slightly higher. For the TSPLIB based instances the clustering algorithms had more problems to find valid solutions, but for each test instance at least one type of clustering was successful.

Kurzfassung

In dieser Diplomarbeit wird die Anwendung von Clusteringalgorithmen untersucht, um das Hierarchical Ring Network Problem (HRNP) zu lösen.

Wenn das Netzwerk als Graph repräsentiert ist, ist dieses NP-vollständige Problem wie folgt definiert: Gegeben ist Menge von Knoten welche jeweils einer von drei Schichten zugewiesen sind, und eine Kostenfunktion, welche die Verbindungskosten zwischen zwei Knoten (d.h. Kantenkosten) zuweist. Gesucht ist ein zusammenhängendes Netzwerk mit minimalen Gesamtkosten, wobei dieses bestimmte Struktureigenschaften zu erfüllen hat, welche im Detail in der Diplomarbeit beschrieben werden. Die wichtigste dieser Eigenschaften ist, dass Knoten gemäß einer hierarchischen Struktur zu größenbeschränkten Ringen verbunden werden.

Ringstrukturen sind ein guter Kompromiss zwischen der Verfügbarkeit von Netzwerken und deren Herstellungskosten. Die Verfügbarkeit ist gewährleistet, solange maximal ein Knoten pro Ring ausfällt.

Die grundlegende Idee dieser Diplomarbeit um dieses Netzwerkdesign-Problem zu lösen, ist die Knoten mit Hilfe von hierarchischen Clusteringalgorithmen anzuordnen und die resultierende Hierarchie für nachfolgende Heuristiken zu verwenden, welche die Ringe finden. Vorhergehende Ansätze für vergleichbare Netzwerkdesign-Probleme haben die inhärente Netzwerkstruktur nicht auf solche Weise genützt und eher Greedy-Heuristiken eingesetzt.

Um gültige Ringe zu finden, wurden drei Heuristiken implementiert. Strategisches Variieren der erlaubten Ringgröße hilft der ersten Heuristik Ringe unter Benützung der Cluster-Hierarchie zu finden. Die zweite Heuristik baut auf den in der vorherigen Schicht gefundenen Ringen auf, indem sie nach gültigen Pfaden sucht, die an diese Ringe angeschlossen werden können. Drittens wird eine Reparaturheuristik angewendet, welche versucht verbleibende Knoten zu bestehenden Ringen zuzuweisen.

Zuletzt werden lokale Suchverfahren eingesetzt, um die Gesamtkosten zu verbessern.

Um zu überprüfen, wie gut dieser Lösungsansatz funktioniert, wurden zwei Testinstanz-Generatoren implementiert. Der Erste generiert Instanzen zufallsbasiert, der Zweite baut auf dem bekannten TSPLIB-Archiv auf.

Die Evaluierung der zufallsbasierten Testinstanzen hat gezeigt, dass alle drei Heuristiken sämtliche Instanzen lösen konnten, wobei Girvan-Newman und Kernighan-Lin in jedem Testlauf Lösungen gefunden haben, war dies bei K-means nicht der Fall. Mit Kernighan-Lin konnte im Durchschnitt schneller eine Lösung gefunden werden, aber die Gesamtkosten waren bei den beiden anderen Algorithmen etwas besser. Mit den TSPLIB-basierten Testinstanzen konnte nicht mit allen Clusteringalgorithmen eine Lösung erzielt werden, aber zumindest war für jede Testinstanz mindestens ein Clustering-Verfahren erfolgreich.

Contents

CHAPTER 1

Introduction

The design of networks (e.g., telecommunication, transportation etc.) is undoubtedly an important task in today's world. As networks grow larger a need for algorithms that can handle big instances emerges. Traditionally Integer Linear Programming (ILP) Techniques are used for solving network design problems because it is able to find optimal solutions. Unfortunately they are often incapable of solving big instances. Heuristics have shown that they can produce very good results in far less time. Therefore, various heuristic algorithms are investigated in this work.

A special focus lies on reliability. Reliable networks often have a ring structure in common that helps building so called self healing networks. If one hub node fails the traffic can be rerouted so that the other nodes are not affected. It is even possible to keep the network alive and continue providing services if one node per ring fails. Other network structures would ensure high availability and robustness features too, but ring structures have the important benefit that they can be built at relatively cheap costs.

Nevertheless, heuristics are often not applied for rinding ring structures in networks, because it is usually difficult to find appropriate methods for establishing those structures. In this thesis clustering heuristics are taken as an approach to tackle this challenge. Moreover the input graph is in general not complete, which is closer to natural input instances.

As an example telecommunication networks should be mentioned. Typically they are built on existing infrastructure, for example beside roads or railways. This implies restrictions on the design of networks. A natural approach for the design would be to use data about existing infrastructure as an input model on which the network is built. The challenge would be to select appropriate parts and to extend the model where necessary. This could mean choosing the roads where new network cables should be laid. In some situations connections are required where no appropriate infrastructure exists. Then additional connections have to be inserted in the model, usually with relatively high costs.

1.1 Problem Definition

This section explains the formal specification of the problem. A convention for the symbolic notation is introduced first. An example input instance is illustrated in Figure 1.1 and one possible solution is shown in Figure 1.2.

Notation

V	...	Set of nodes.
E	...	Set of undirected edges.
$G = (V, E)$	...	Graph G with vertex set V and edge set E. The notation $V(G)$ stands for the vertex set of graph G.
K	...	Number of layers.
k	...	The k^{th} layer, where $1 \leq k \leq K$.
V_k	...	Nodes in layer k.
E_k	...	Edges in layer k, i.e., $(i, j) \in E_k \Leftrightarrow (i \in V_k \wedge j \in V_k)$.
E'_k	...	Edges between layer k and layer $k-1$, i.e., $(i, j) \in E'_k \Leftrightarrow ((i \in V_k \wedge j \in V_{k-1}) \vee (i \in V_{k-1} \wedge j \in V_k))$.
$R_{k,i}$	...	The i^{th} ring in layer k. The notation $G[R_{k,i}]$ stands for the graph representing the ring $R_{k,i}$.
b^l_k	...	Lower bound of nodes for a ring in layer k, i.e., a ring in layer k can have at least l nodes.
b^u_k	...	Upper bound of nodes for a ring in layer k, i.e., a ring in layer k can have up to u nodes.
c_{ij}	...	Cost of edge $(i, j) \in E$.
x_{ij}	...	Variable which is 1 (true) exactly if the edge $(i, j) \in E$ is part of the solution, otherwise it is 0 (false).
$l : V \rightarrow \mathbb{N}$	...	Function l that assigns a layer (level) to each node.

Definitions

A *chain* C_k is a node disjoint path, i.e., a sequence of disjoint edges. All nodes that are part of the chain must be on the same level: $\forall e \in C_k : e \in E_k$.

A *ring* R is a node disjoint cycle, i.e., a sequence of disjoint edges $e \in E$, that start and end in the same node. It can be seen as a closed path. Ring is a figurative expression that symbolizes the intended structure. A ring R_k consists of two chains, the *upper chain* and the *lower chain* and two connection-edges (uplinks) $u_1, u_2 \in E'_k$. The upper chain consists of the nodes that are part of the connected ring (a path between the hub nodes). The lower chain is a path between the hub nodes that essentially consists of the nodes to be connected to the upper layer. For a ring R_k the upper chain is in layer $k-1$ and the lower chain is in layer k.

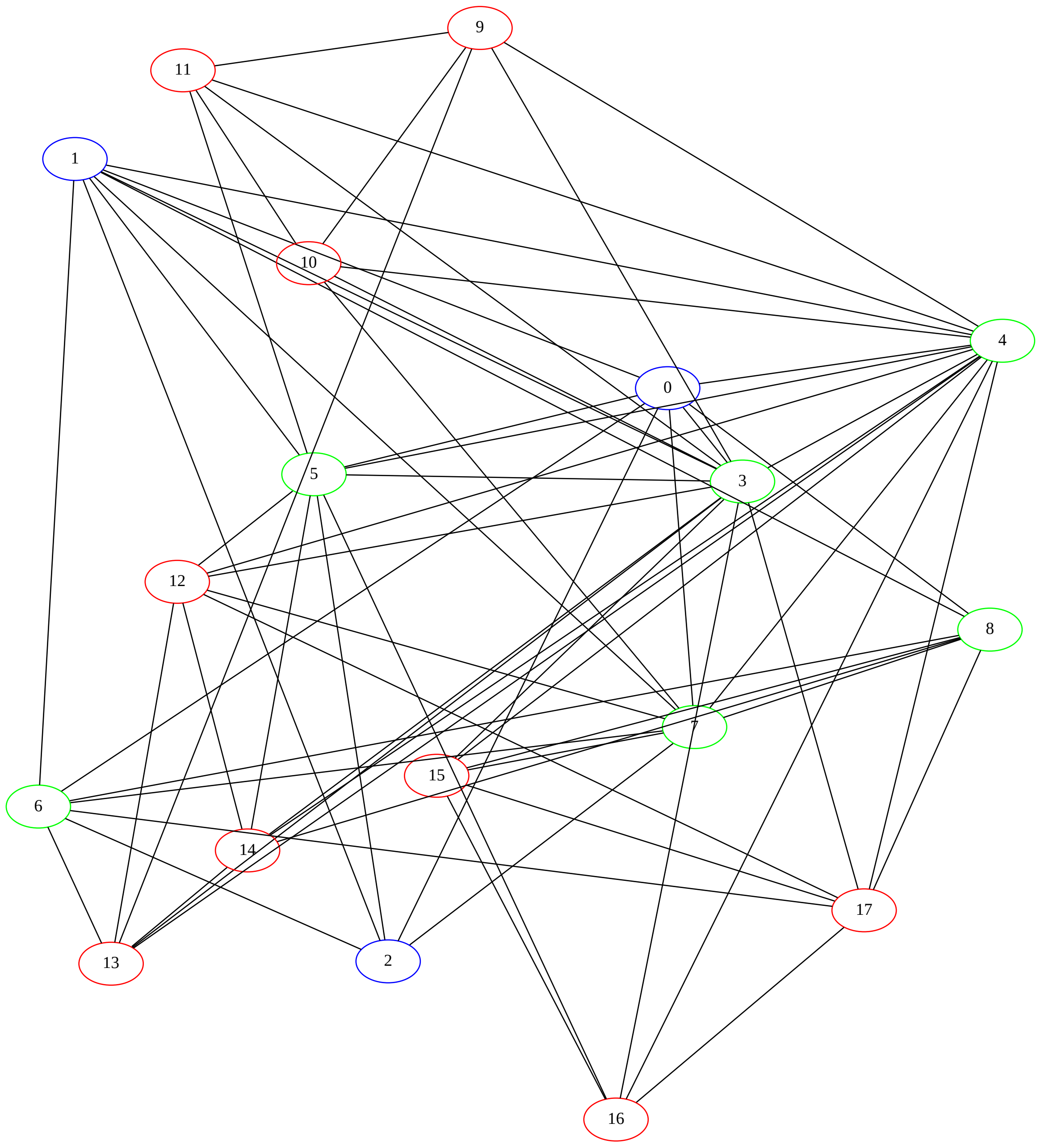


Figure 1.1: Sample input instance. Blue nodes (0 – 2) are assigned to layer 1, green nodes (3 – 8) to layer 2 and red nodes (9 – 17) to layer 3.

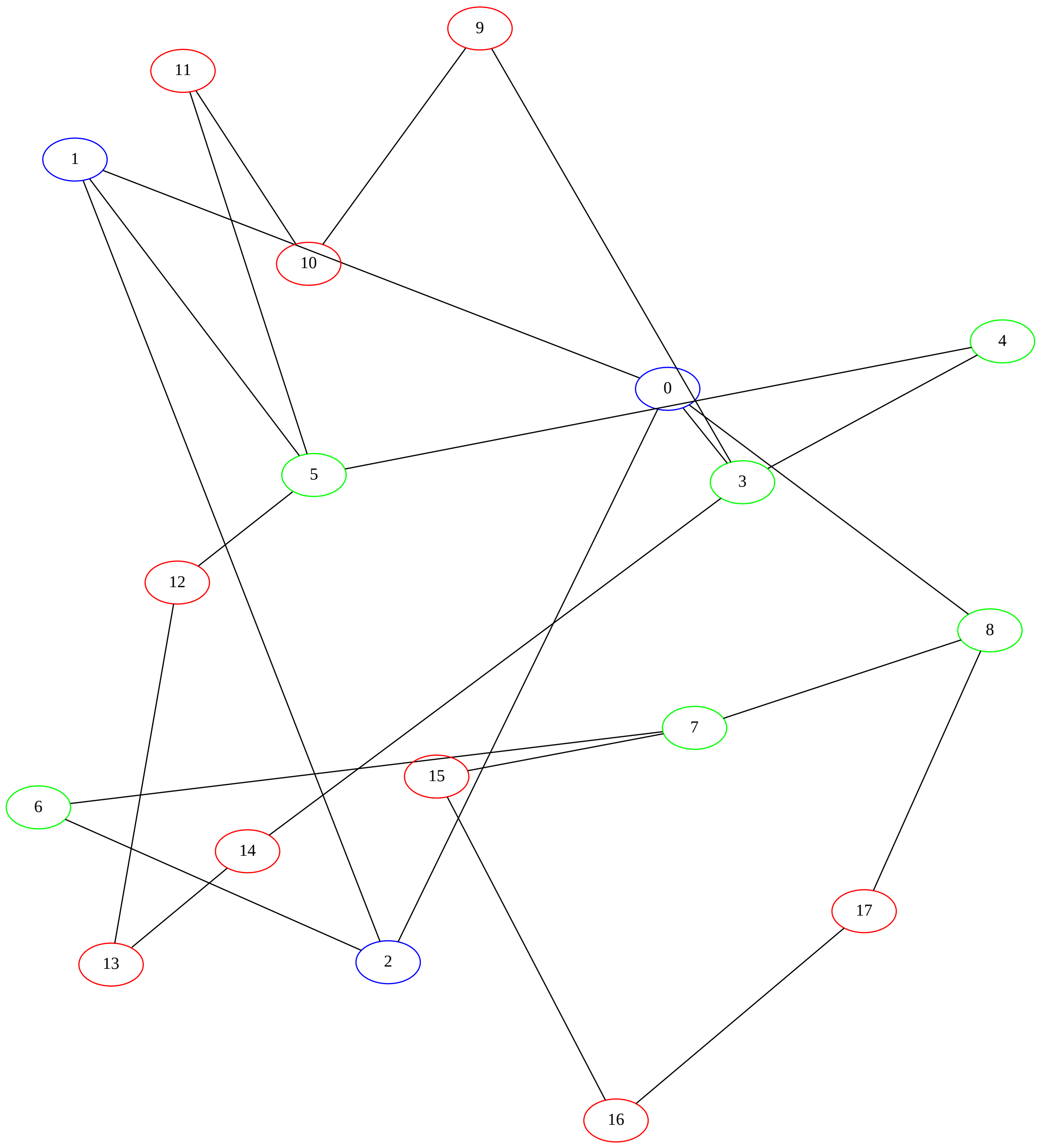


Figure 1.2: Solution for sample instance (see Figure 1.1).

Partition Types / Cluster Types

In this thesis the special case of hierarchical clustering with ring structures is used. The term cluster can have various intended meanings. The meanings used in this thesis are explained in the following:

- Type R: Ring
  A Type R cluster contains the complete ring R_k.
- Type A: Ring + Subrings
  A Type A cluster contains the nodes of the ring and all the nodes of the lower levels that are connected to the ring.
- Type Lc
  A Type Lc cluster contains only the lower chain.

Input

- Connected (undirected) simple graph $G = (V, E)$
- Each edge $e = (i, j)$ has an assigned weight: c_{ij}
- Each node v has an assigned layer (level): $l(v)$
- The number of layers K equals 3
- The graph is preprocessed so that it contains no edge connecting nodes in V_1 and V_3 is contained

Objective Function

$$\min \sum_{i,j \in V} c_{ij} \cdot x_{ij} \tag{1.1}$$

Minimize the sum of edge weights of the solution.

Instance Constraints

The formal problem description is split into two cases: The definition for the backbone ring V_1, which can be treated as a separate problem, and the definition for the other layers.

Formulation for $k = 1$:

$$\sum_{(i,j)\in E_1} x_{ij} = 2, \quad i \in V_1 \tag{1.2}$$

$$\sum_{(i,j)\in E_1 | i\in S, j\notin S} x_{ij} \geq 2, \quad \forall S \subset V_1, S \neq \emptyset, V_1 \tag{1.3}$$

$$x_{ij} \in \{0, 1\}, \quad \forall (i, j) \in V_1 \tag{1.4}$$

The constraints for the case $k = 1$ ensure that the nodes in the first layer build a Hamiltonian cycle. It is a classical TSP formulation [3].

Formulation for $1 < k \leq K$:

For each layer k a set of rings $\{R_{k,1}, \ldots R_{k,m_k}\}$ has to be found with the following constraints:

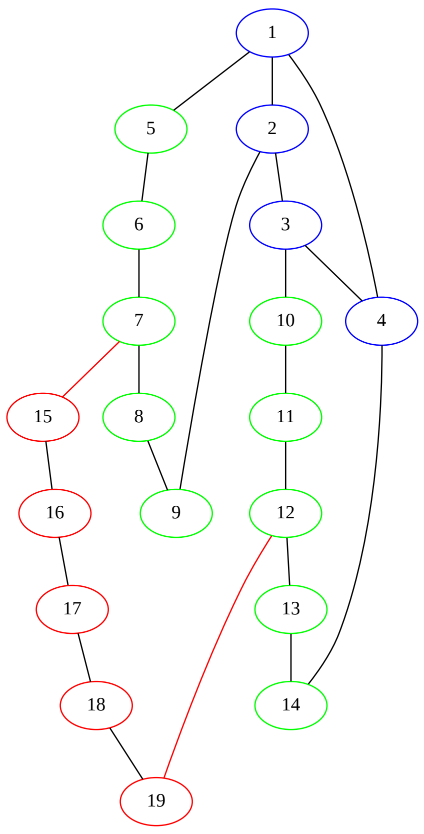
$$R_{k,i} \subset E, \forall i = 1, \ldots, m_k \tag{1.5}$$
$$|V(G[R_{k,i}]) \cap V_k| \geq b_k^l, \forall i = 1, \ldots, m_k \tag{1.6}$$
$$|V(G[R_{k,i}]) \cap V_k| \leq b_k^u, \forall i = 1, \ldots, m_k \tag{1.7}$$
$$|V(G[R_{k,i}]) \cap V_{k-1}| \geq 2, \forall i = 1, \ldots, m_k \tag{1.8}$$
$$(V(G[R_{k,i}]) \cap V_k) \cap (V(G[R_{k,j}]) \cap V_k) = \emptyset, \forall i, j \in 1, \ldots, m_k \wedge i \neq j \tag{1.9}$$
$$V(G[\bigcup_{i=1}^{m_k} R_{k,i}]) \cap V_k = V_k \tag{1.10}$$
$$m_k \geq 1 \tag{1.11}$$
$$x_{ij} = 1, \quad \forall (i,j) \in \{R_{k,1}, \ldots R_{k,m_k}\} \tag{1.12}$$

Constraints 1.6 and 1.7 restrict the chain size to the lower and upper bound. Each chain has to be homed to two hubs in the upper chain 1.8. The lower chains (of the same level) must be node disjoint 1.9. Each node has to be contained in one ring 1.10. All levels must consist of at least one node 1.11 e.g., it is not possible that there is no ring in level 2. Constraint 1.12 connects these constraints with the objective function 1.1 which means that for all edges contained in rings the objective variable x has to be set to 1 (true).

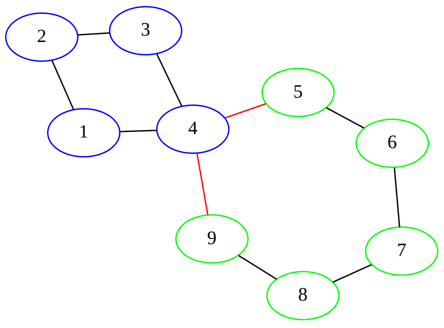
Informal constraints for the heuristics:

In the following constraints are rewritten informally for a better understanding.

- The level difference of the endpoints of each edge $e = (v_1, v_2)$ in a ring solution must be ≤ 1. For example an edge between a level 1 node and a level 3 node must not exist.
- Each ring with $k > 1$ has to be connected to 2 hub nodes (2-connectivity). This is also called dual-homing. For a violation see Figure 1.4b and Figure 1.4d.
- Exactly three layers are used, i.e., $K = 3$.
- The hubs a ring is connected to must be on the same ring. For a violation see Figure 1.3a.
- The hubs a ring is connected to must be distinct (see Figure 1.3b).
- All nodes must be connected to form a single component.
- Level of hub = level of connected node + 1 (see Figure 1.4a).



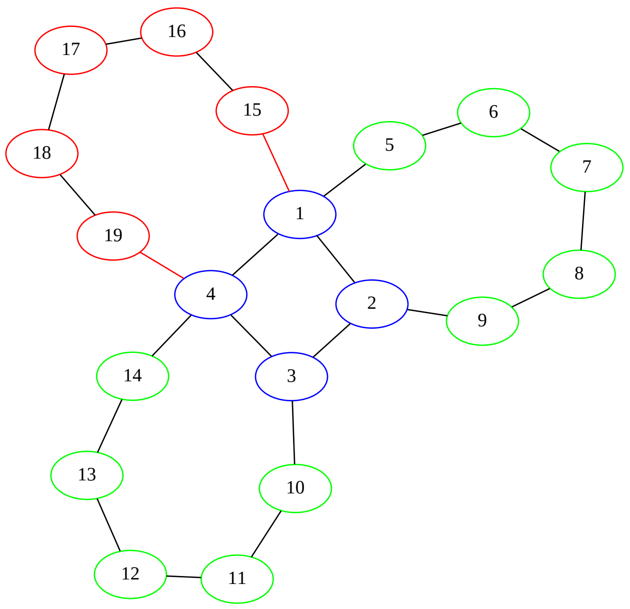
(a) Violation 1: The layer 3 ring is homed to two different layer 2 rings.



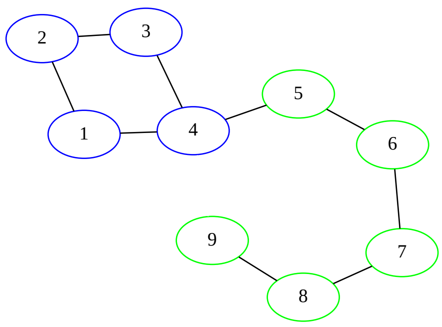
(b) Violation 2: The layer 2 ring is homed to the same layer 1 node twice.

Figure 1.3: Some violations of the instance constraints. An upper bound of $u = 5$ is assumed.

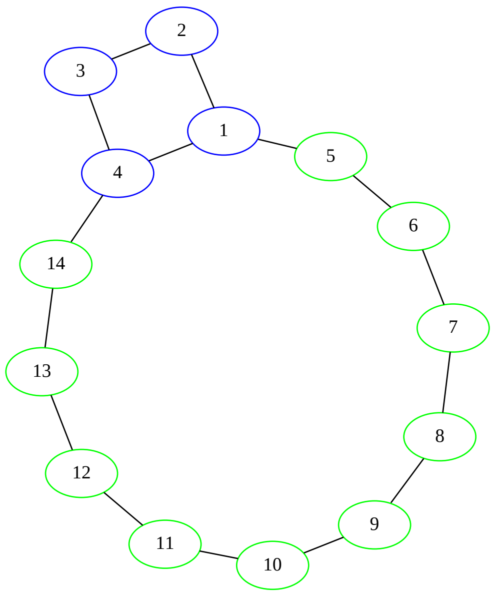
- The ringsize is bounded by b_k^u (see Figure 1.4c). This avoids degenerate solutions where too long rings are built, similar to the TSP.
- If $b_k^l = 1$ it is allowed that a lower chain consists of only one node. This helps finding solutions in layer k, but avoids homing subrings of layer $k+1$ to this ring because it prohibits dual-homing.



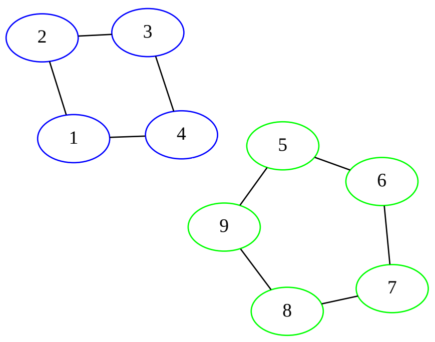
(a) Violation 3: The layer three ring is connected to a layer 1 ring (but it must be connected to a layer 2 ring).



(b) Violation 4: The layer 2 ring is only connected once (no dual homing).



(c) Violation 5: The layer 2 ring is too long (it exceeds the bound $u = 5$).



(d) Violation 6: The solution rings are not connected.

Figure 1.4: Some violations of the instance constraints. An upper bound of $u = 5$ is assumed.

1.2 About the Complexity

To get a better understanding how difficult it is to solve a combinatorial problem is, it can be investigated by means of complexity analysis, which is a major topic in computer science. One of the most interesting questions is, if the considered problem belongs to the set of $\mathcal{NP}$-complete problems, which means that no algorithm with polynomial time complexity can exist to solve this problem, as long as we can assume that $\mathcal{P} \neq \mathcal{NP}$. If one could find an algorithm of polynomial time complexity that can solve at least one of the $\mathcal{NP}$-complete problems, this would mean that all problems in this complexity class could be solved with a polynomial time algorithm. This would also have the impact that the assumption that $\mathcal{P} \neq \mathcal{NP}$ holds could be rejected.

In the case of the HRNP the first layer can be treated independently as a Traveling Salesman Problem (TSP), see [14], which is $\mathcal{NP}$-complete. The optimal ring connection of all nodes of V_1 resembles an optimal TSP tour for these nodes.

The problem to find appropriate rings for the other layers can be reduced from the Traveling Salesman Problem with Precedence Constraints (TSPPC), see [13], which was shown to be $\mathcal{NP}$-hard.

For layer 2 (layer 3) this means that if exactly two uplinks are contained that determine the precedence and $b_2^u = |V_2|$ ($b_3^u = |V_3|$), the single ring connecting all nodes of V_2 (V_3) resembles a TSPPC tour.

To know that there is no polynomial algorithm that solves the Hierarchical Ring Network Problem motivates the use of heuristics to tackle larger instances, as it was done in this thesis.

CHAPTER 2

Related Work

Hierarchical Network Design Problem

Current introduces the basic Hierarchical Network Design Problem (HNDP) in [5]. It consists of primary nodes building a primary path that from a start- to an end-node. The other nodes are the secondary nodes that have to be connected to the primary nodes via secondary paths. The total cost is the cost of the primary path plus the sum of costs of all secondary paths. The objective is to minimize the total cost. The paper contains an ILP formulation and a heuristic approach. The heuristic calculates the M shortest paths between the start- and the end-node and then calculates an MST for each of them and takes the minimum of the M solutions.

Multi-level Network Design Problem

In [1] the Multi-level Network Design (MLND) problem is introduced. It is a generalization of the Hierarchical Network Design (HNDP) problem, where K levels are used to describe the importance of nodes. If two levels are used then this is called the Two-level Network Design (TLND) problem. The generalization of the HNDP is that the primary layer can consist of more than exactly two designated nodes. An ILP formulation based on steiner trees and one based on multicommodity flow are provided.

Aspects of Network Design

Klincewicz [12] investigates various aspects of network design, like the cost, capacity, reliability, performance and demand pattern of networks. The cost can be important for hubs (nodes) and links either for creating or for using them. Also for both of them capacity constraints can apply. Reliability can be improved by multi-homing or by the more general approach that multiple paths between nodes are available for rerouting. The performance measures if enough networks resources are available for a given demand, e.g., if enough capacity is available along some path.

Demand patterns describe the needed communication between the nodes, for example many-to-many or many-to-one node relations. The paper also gives a good survey about combinations of network structures between backbone (primary) and tributionary (secondary) topologies.

Survivable Networks with Bounded Rings

Fortz uses in this phd thesis [8] bounded rings for the reliability of networks. The resulting network has to be connected. A branch-and-cut approach and various heuristics are used to find the rings.

Some of the heuristics are shortly explained below (the names of the heuristics are from the original thesis):

Ear-inserting method

This method creates one ring after the other. At first a minimum length cycle is created. Then the node is chosen to be inserted into the ring that has the least insertion cost (i.e., replacing an existing edge by the two new ones that connect the node to the ring) until no node can be inserted without violating ring constraints. Then a new cycle is created and the procedure is repeated until all nodes are inserted.

Cutting cycles into two equal parts

A solution should be found where the bound constraint is relaxed. This may lead to a Hamiltonian cycle. Later a cycle, where the bound is violated, is split into two parts and edges are added at the splitting points to get two cycles. This procedure is repeated recursively, until the bound constraints are fulfilled.

Path following method

At first a Hamiltonian cycle is created. Next the algorithm follows this tour (starting at some arbitrary point in some arbitrary direction) as long as the bound constraint is satisfied. Then an edge back to the starting point is added to close the cycle and the next point (following the tour) becomes the new starting point. This procedure is repeated until the first starting point is reached again.

Stringy method

The noticeable feature of this method is that it starts with all edges and removes edges systematically. The crucial criterion for removing an edge is that the graph stays 2-connected after the removal.

Hierarchical Network Topologies

Thomadsen's phd thesis [18] focuses on hierarchical network topologies. Many properties are described as ILP formulations. A chapter about ring structures is provided. The thesis contains a formulation of the Fixed Charge Network Design (FCND) problem. It involves demand between nodes, edge costs and the cost to use edges (satisfy demand).

Ring-chain Dual Homing

Lee [15] describes Self Healing Rings (SHR) in the context of the Ring-chain Dual Homing (RCDH) problem. An SHR is a cycle of network nodes that can reroute the traffic in case of a node failure. A chain is a path of nodes that is linked (homed) to distinct hub nodes on the SHR at each chain end.

CHAPTER 3

Methodology

In this chapter the theoretical background is explained. It mainly contains the data structures and algorithms that are used for the implementation.

3.1 Datastructures

Different ways are used to model application data. Especially the hierarchical aspect and the ring structure are important.

Dendrogram

A dendrogram is a tree that can be used for the hierarchical representation of data. Nodes that are closer to the root (i.e., the path length from the node to the root is shorter) are hierachically on a higher level. The root stands for all values, and the leafs represent the values that are hierachically structured. E.g., if the values are nodes in a graph, each subtree in the dendrogram can be seen as a cluster. For an example see Figure 3.1.

Cluster-Tree

A cluster-tree is the basic abstraction for the (partial) solution graph. In the case of this topic each node in the tree represents a cluster and therefore, it stands for many nodes in the network. To be more precise it is intended to build a ring from the nodes of the cluster.

The edges of the cluster-tree stand for the connections between the rings. This means if two nodes from the cluster-tree are connected via an edge their rings are connected ("homed") in the graph. In the case of dual-homing the edge stands for the two connections from the subring to the hubs of the upper ring. Since uplinks are only allowed to the same ring, one edge is sufficient for representing this connection. It is important to note that the nodes and edges building the cluster-tree are abstract representations of the elements in the original graph, but they are not contained there.

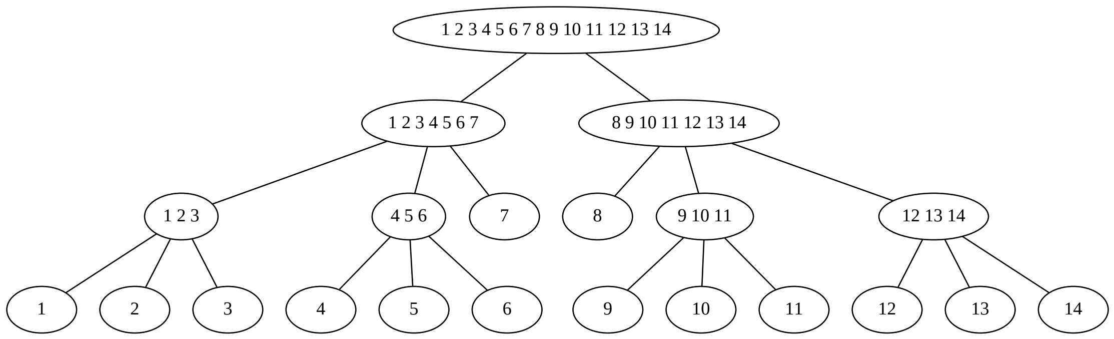


Figure 3.1: Dendrogram sample. Nodes 1, 2 and 3, for example are hierachically on a lower level than node 7.

The cluster-tree is similar to the dendrogram structure of clustered graphs, with two main differences:

1. Each node in the cluster-tree should represent a ring, whereas nodes in the dendrogram stand for regions and general clusters, which need not have the same granularity.
2. A dendrogram node stands for a cluster *including* all subclusters (i.e. Type A). In the cluster-tree a node *only* stands for the nodes in the respective ring (i.e. Type R). One complete subtree (i.e., the union of its nodes) of the cluster-tree would correspond to one dendrogram node.

3.2 Algorithms

The focus in this section lies on clustering algorithms. Moreover, improvement and repair heuristics are explained here too. The last part is about other algorithms that are important for the implementation.

Cluster Analysis

Clustering is a process where a set of data is divided into subsets so that the elements in each subset are similar according to some distance function. The subsets are called clusters and can be seen as groups of similar items. The terms cluster analysis and clustering are used synomymously. Cluster analysis is commonly used in statistics, where the data are usually statistical observations. For a simple illustration of a clustering example, see Figure 3.2.

The elements are typically represented as vectors in n-dimensional space, in which each dimension defines an element's property.

A distance metric is a function that takes two vectors as input and assigns a non-negative number based on the relative positions of the vectors to each other as output. A common distance metric is the Euclidean distance, where the distance function d for the vectors x and y is defined

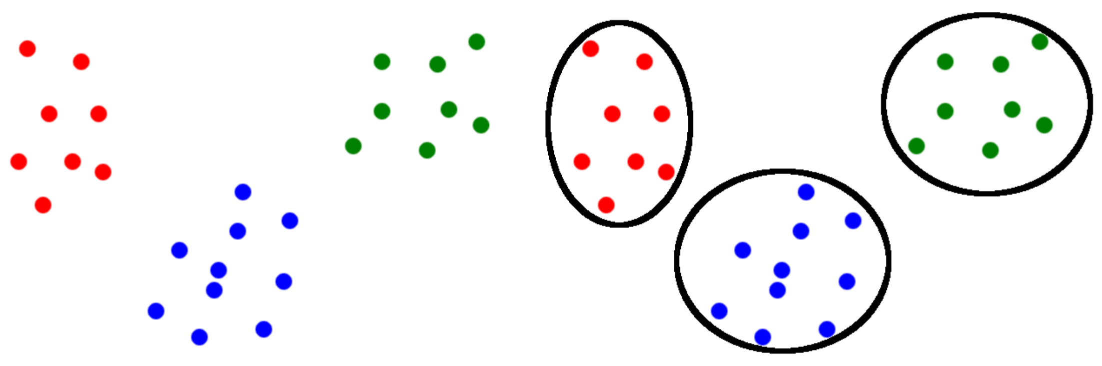

(a) Original set. (b) Result of the cluster analysis.

Figure 3.2: Cluster analysis example. Elements are clustered according to their color property.

---

as follows:

$$d(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Other metrics for example are the Manhattan distance, Hamming distance or Mahalanobis distance (e.g., where normalization of the vector is needed).

Girvan-Newman Clustering

The Girvan-Newman algorithm [6] is based on finding components in a network. It was successfully applied to various social networks. One usecase is the detection of online communities.

A simple approach to cluster a network is to detect its connected components. Since real world networks are highly connected so that usually just one giant component is contained, the *betweenness* measure is used instead.

Betweenness can be defined on the nodes V and on the edges E of a graph $G = (V, E)$. In the following σ_{st} is defined as the number of shortest paths (SP) between the nodes s and t, $\sigma_{st}(v)$ is defined as the number of SP between the nodes s and t that "run through" the vertex v and $\sigma_{st}(e)$ is defined as the number of SP between the nodes s and t that "run along" the edge e.

This leads to the following formula definitions [9, 4]:

Node betweenness of vertex v: $\sum\limits_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$

Edge betweenness of edge e: $\sum\limits_{s \neq t \in V} \frac{\sigma_{st}(e)}{\sigma_{st}}$

Girvan-Newman algorithm in pseudocode is shown in Algorithm 3.1.

The algorithm works as follows: At the beginning the graph consists of one component. In each iteration the algorithm determines the edge e with the highest betweenness value. This edge is removed. Then the algorithm checks if the graph can be split, which means that one component can be separated. If this is the case, the original component (before splitting) that

Algorithm 3.1: The Girvan-Newman algorithm.

```
input  : A connected graph $G = (V, E)$
output: A hierarchical clustering of the graph $G$
1 while edges left in graph $G$ do
2     calculate betweenness values for edges $e \in E$;
3     delete edge $e$ with highest betweenness;
4     if graph splits in more components then
5         build a new cluster for each component;
6         add each new cluster as subcluster to its parent cluster;
7     end
8 end
9 return cluster hierarchy;
```

stands for a cluster is split into two subclusters, which are hierarchically attached to the original cluster. This gives two corresponding subdendrograms. After that the next iteration is started. This procedure is continued until no edges are left so that the graph is fully transformed into a dengrogram and the hierarchical clustering process is finished.

Note that no edge weights are considered. Only the shortest paths influence the betweenness.

To illustrate the Girvan-Newman algorithm a detailed example follows.

Example

Given an undirected weighted graph $G = (V, E)$ with vertices

$$V = \{1, \ldots, 11\}$$

edges

$$\begin{aligned} E = \{ \quad & (1,2),(1,3),(2,3),(2,4),(3,4), \\ & (4,5),(5,6),(5,7),(6,7),(7,8), \\ & (8,9),(8,10),(9,10),(9,11),(10,11) \quad \} \end{aligned}$$

and (edge) weight function

$$w(e) = 1, \forall e \in E$$

which leads to the adjacency matrix in Table 3.4. The graph is shown in Figure 3.3.

To calculate the edge betweenness values the first step is to calculate the number of shortest paths between nodes. An efficient method that is based on Breadth-First-Search is described in [6]. The resulting SP values are shown in Table 3.1.

Next, the proportion of the number of SP passing trough each edge has to be calculated. For each node pair (s, t) with $s \neq t \in V$ the set of all SP has to be determined and for each edge $e \in E$ the number of paths that contain e will give the value for $\sigma_{st}(e)$.

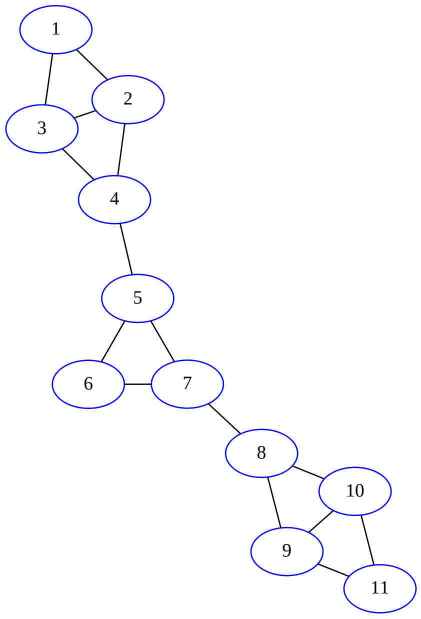
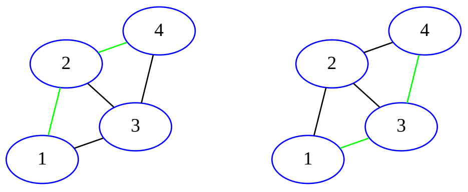


Figure 3.3: Visual representation of the example graph.

	1	2	3	4	5	6	7	8	9	10	11
1		1	1								
2	1		1	1							
3	1	1		1							
4		1	1		1						
5				1		1	1				
6					1		1				
7					1	1		1			
8							1		1	1	
9								1		1	1
10								1	1		1
11									1	1	

Figure 3.4: Adjacency matrix of the example graph. Empty values indicate that there is no edge between the specific nodes (0-values were omitted for readability).

Figure 3.5: The example graph.



(a) First shortest path (green) between node 1 and node 4.

(b) Second shortest path (green) between node 1 and node 4.

Figure 3.6: Both shortest paths between the nodes 1 and 4.

For example for $s = 1$ and $t = 4$ there are 2 shortest paths (therefore $\sigma_{1,4} = 2$), namely $\langle (1,2), (2,4) \rangle$ and $\langle (1,3), (3,4) \rangle$. This set of shortest paths contains the edges $(1,2)$, $(1,3)$, $(2,4)$ and $(3,4)$. Each of these edges occurs in exactly one shortest path (between 1 and 4), so $\sigma_{1,4}((1,2)) = \sigma_{1,4}((1,3)) = \sigma_{1,4}((2,4)) = \sigma_{1,4}((3,4)) = 1$. Therefore, each of the proportions $\frac{\sigma_{1,4}(e)}{\sigma_{1,4}}$ is $\frac{1}{2}$. See Figure 3.6 for illustration.

The resulting betweenness values after the first iteration are shown in Table 3.2 and the graph

	1	2	3	4	5	6	7	8	9	10	11
1		1	1	2	2	2	2	2	2	2	4
2			1	1	1	1	1	1	1	1	2
3				1	1	1	1	1	1	1	2
4					1	1	1	1	1	1	2
5						1	1	1	1	1	2
6							1	1	1	1	2
7								1	1	1	2
8									1	1	2
9										1	1
10											1
11											

Table 3.1: Number of shortest paths between nodes (σ_{st}). Only the upper triangle matrix is shown, the other half is symmetric since the graph is undirected.

	1	2	3	4	5	6	7	8	9	10	11
1		10	10								
2			2	24							
3				24							
4					56						
5						10	50				
6							10				
7								56			
8									24	24	
9										2	10
10											10
11											

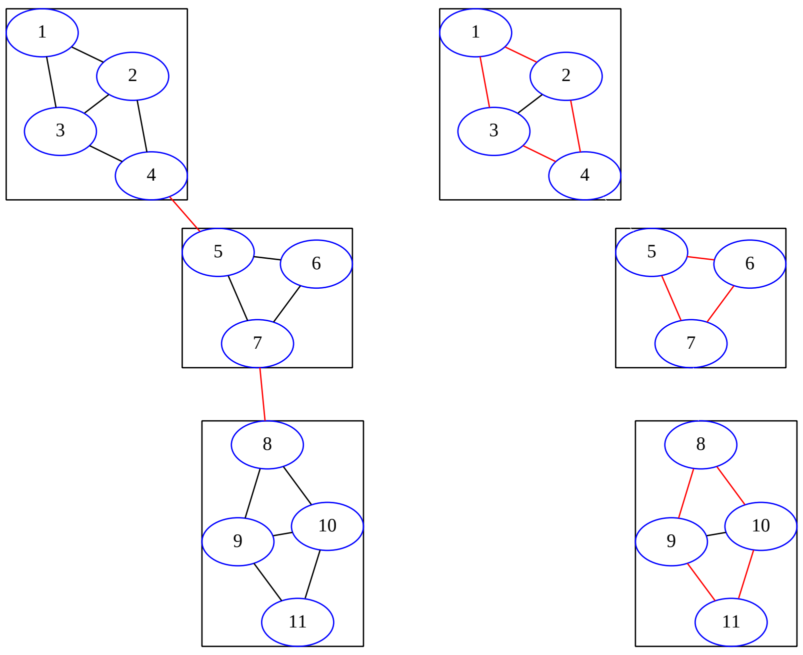
Table 3.2: Betweenness values after the first iteration.

in Figure 3.7a. The edges $(4, 5)$ and $(7, 8)$ have the highest betweenness value of 56. If binary clustering is desired, only one of those edges will be removed, otherwise all edges with the highest value are removed from the graph.

In this case the graph splits into three components.

After that the betweenness calculation process starts again (for each component). For the next iteration, the exact values are omitted, but the resulting graph is shown in Figure 3.7b.

The result of the clustering process can now be seen in Figure 3.8.



(a) Graph after the first iteration. (b) Graph after the second iteration.

Figure 3.7: Example graph after iteration 1 and 2. Edges of highest betweenness value (red) are removed at that iteration.

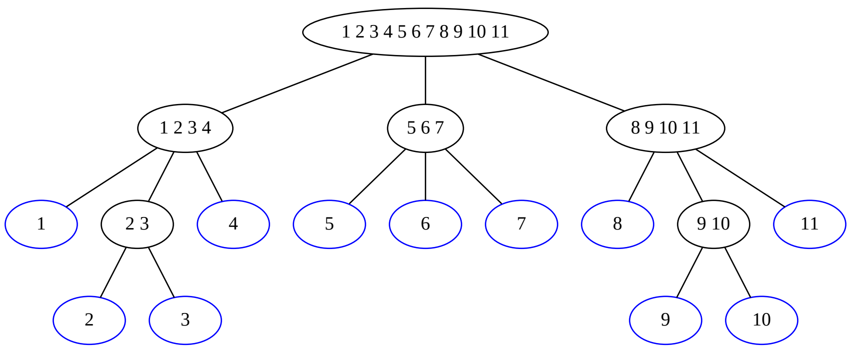


Figure 3.8: Resulting clustering of the example graph respresented as a dendrogram.

K-means Clustering

K-means is a popular clustering technique that partitions the node set of a graph into k distinct clusters. For each cluster one node acts as a special element (the centroid) from which the distance can be calculated.

K-means in pseudocode can be found in Algorithm 3.2.

Algorithm 3.2: Pseudocode of the K-means algorithm.

```
input  : The set $V$ of elements to be clustered. A weight function $w$. The number of
         expected clusters $k$.
output: An assignment from elements to centroids.
1 Choose $k$ distinct elements from $V$ as the centroids;
2 repeat
3     Assign each element from $V$ to the nearest centroid;
4     Calculate the new center of each cluster;
5 until stop criterion;
6 return assignment;
```

A typical stop critertion is met if the centroids did not change between two iterations, or if a maximum number of iterations was executed.

The initial centroids (first step in pseudocode) are usually chosen randomly. The Floyd-Algorithm 3.6 is one possible method to find such a random selection. K-means++ is a variant that chooses the initial centroids in a more uniformly distributed way to avoid a selection where the centroids are too close.

A fast method for K-means clustering is described in [17].

Partitioning Around Medoids is a variant of K-means. The medoids are analogous to the centroids from K-means. This algorithm systematically checks new medoid assignments by swapping current medoids with non-medoids one-by-one and checks if this new assignment gives an improvement. The pseudocode is shown in Algorithm 3.3.

Kernighan-Lin Clustering

The Kernighan-Lin-Algorithm [11] is another approach to solve the graph partitioning problem. It splits the set of vertices of a weighted graph into two subsets. The subsets have to be disjoint and of equal size. The sum of weights of the edges between the subsets has to be minimized. For the pseudocode see Algorithm 3.4.

More formally: Given a weighted graph $G = (V, E)$ with weight function w_e a partition of V into the sets A and B with $A \cap B = \emptyset$ and $|A| = |B|$ should be found, with the following objective function:

$$\min \sum_{(a,b) \in E : a \in A \wedge b \in B} w_{(a,b)}$$

Algorithm 3.3: Pseudocode of the Partitioning Around Medoids algorithm.

```
input  : The set $V$ of elements to be clustered. A weight function $w$. The number of
         expected clusters $k$.
output: An assignment from elements to medoids.

Choose $k$ distinct elements from $V$ as the medoids;
repeat
    Assign each element from $V$ to the nearest medoid;
    foreach medoid $m$ do
        foreach element $e$ that is no medoid do
            Swap $m$ and $e$;
            if new minimal cost assignment found then
                Save assignment as new minimum;
            end
        end
    end
until medoids not changed;
return assignment;
```

V must contain an equal number of elements (if this is not the case, an artificial element can be added).

Additionally the following terms are introduced:

The external cost E_a of an element $a \in A$ is defined as $\sum\limits_{b \in B} w_{(a,b)}$.

The internal cost I_a of an element $a \in A$ is defined as $\sum\limits_{b \in A} w_{(a,b)}$.

The cost difference D_a is defined as $D_a = E_a - I_a$.

If a node $a \in A$ is moved to B and a node $b \in B$ is moved to A, the cost reduction can be calculated by $D_a + D_b - 2w_{(a,b)}$. This formula is an important ingredient for the Kernighan-Lin algorithm, which tries to maximize the cost reduction.

In each iteration of the algorithm the cost difference for each element is calculated (i.e., the cost difference if the element is moved to the other set). For $\frac{|V|}{2}$ a pair of items (one item from each set) is searched to maximize the cost reduction. This gives a sequence of cost reductions with $\frac{|V|}{2}$ items. Note that cost reduction can also be positive. The subsequence starting at the first element that has the best total cost reduction is chosen in each iteration. The corresponding items are then swapped between the sets. These iterations are repeated until no cost reduction (gain) can be found.

Merging Rings

Some ring-finding heuristics tend to produce small rings. To improve the resulting cost, and – even more important – for finding rings in lower layers, rings that are close to the upper bound are better. Merging rings is a simple improvement heuristic, that tries to connect pairs of rings by concatenating their lower chains. There are four possibilities how the chains can

Algorithm 3.4: Pseudocode of the Kernighan-Lin algorithm.

```
input  : The set $V$ of elements to be partitioned. A weight function $w$.
output: The resulting partitions $A$ and $B$.
split $V$ into equal initial sets $A$ and $B$;
repeat
    $\bar{A} \leftarrow A$;
    $\bar{B} \leftarrow B$;
    foreach $a \in \bar{A}$ do
        compute $D_a$;
    end
    foreach $b \in \bar{B}$ do
        compute $D_b$;
    end
    for $p \leftarrow 1$ to $\frac{|V|}{2}$ do
        find $\bar{a} \in \bar{A}$ and $\bar{b} \in \bar{B}$ that maximize cost reduction $\bar{g}$;
        $a_p \leftarrow \bar{a}$, $b_p \leftarrow \bar{b}$, $g_p \leftarrow \bar{g}$;
        move $\bar{a}$ to $\bar{B}$;
        move $\bar{b}$ to $\bar{A}$;
        update affected $D$ values;
    end
    find $k$ that maximizes $gain \leftarrow \sum_{i \leftarrow 1}^{k} g_i$;
    if $gain > 0$ then
        move $a_1 \dots a_k$ to $B$;
        move $b_1 \dots b_k$ to $A$;
    end
until $gain \leq 0$;
return $A$ and $B$;
```

be concatenated at their endpoints. A reversal of the order of edges in the sequence might be necessary, depending on the representation of chains.

2-Opt Heuristic

The 2-Opt heuristic is an optimization heuristic, that is often used to improve TSP solutions. For all pairs (therefore 2) of edges $\{(a,b),(c,d)\}$ the algorithm checks whether a new ordering of the four considered vertices $\{(a,d),(c,b)\}$ improves the solution or not. If a solution in Eucledian space contains closing edges (i.e., they are part of the same cycle and they “cross”), it can be improved by the 2-Opt heuristic.

A generalization of 2-Opt is the k-Opt heuristic, where an enhancement of k edges are checked. A common variant of the k-Opt heuristic is the Lin-Kernighan algorithm [16]. If k-

Opt heuristic with higher k is used better solutions can be found, but the processing is also much higher. The variant 3-Opt is usually a good compromise between runtime and optimization gained.

Multilevel Heuristics

The multilevel paradigm can be used for optimization problems, especially in the case of combinatorial optimization problems [20]. The approach is to coarsen the problem to get an approximate solution. Each coarsening iteration stands for a level in the multilevel algorithm.

Various methods exist that make use of this paradigm. Concerning problems in the field of graph theory for example, one could try to coarsen the graph (e.g., by reducing nodes) and to solve the problem on the reduced graph.

In this thesis a multilevel approach will be used for the variation of the bound constraint (i.e., the restricted ring size b_k^u). Note that a valid solution for a ring bound $\bar{b_k^u}$ with $\bar{b_k^u} \leq b_k^u$ is also a valid solution for the problem with ring bound b_k^u.

Floyd Algorithm

Selecting a random subset of items is a problem that occurs in various situations. A good example would be the test instance generator (from Section 5.1). In this case k neighbors should be chosen randomly for each node (where k can also vary randomly).

A naive approach would be to choose items randomly until k distinct items have been selected, see Algorithm 3.5. Nevertheless, much better approaches exist (e.g., [19], [7] and [10]), especially when a large fraction of all the items has to be chosen. In this case the naive algorithm would have to generate many random values until one element is selected, that was not selected before, since the probability of choosing an item, that was already chosen grows with every successful iteration, making this approach unusable for practical applications.

Algorithm 3.5: Naive Sampling Algorithm [2]

```
input  : The number of integers $k$ that should be selected out of $n$.
output : A set $S$ of randomly selected integers.
1 $S \leftarrow \emptyset$;
2 while $|S| < k$ do
3     $t \leftarrow$ RandomInteger($1, n$);
4     if $t \notin S$ then
5         insert $t$ in $S$;
6     end
7 end
8 return $S$;
```

In this thesis Floyd's algorithm [2] was chosen to generate random subsets, see Algorithm 3.6. In each iteration an element is chosen. Either a random number from 1 to j is added, or the current value of the iterator variable j.

Algorithm 3.6: Floyd's Iterative Sampling Algorithm [2]

```
input  : The number of integers $k$ that should be selected out of $n$.
output: A set $S$ of randomly selected integers.
S ← ∅;
for $j \leftarrow n - k + 1$ to $n$ do
    $t \leftarrow$ RandomInteger(1,$j$);
    if $t \notin S$ then
        insert $t$ in $S$;
    else
        insert $j$ in $S$;
    end
end
return $S$;
```

Algorithm 3.7: Floyd's Permutation Algorithm [2]

```
input  : The number of integers $k$ that should be selected out of $n$.
output: A sequence $S$ (i.e., permutation) of randomly selected integers.
$S \leftarrow \langle\rangle$;
for $j \leftarrow n - k + 1$ to $n$ do
    $t \leftarrow$ RandomInteger(1,$j$);
    if $t \notin S$ then
        prefix $t$ to $S$;
    else
        insert $j$ in $S$ after $t$;
    end
end
return $S$;
```

The permutation version, of the Algorithm 3.7, uses the same basic approach, but uses a list (sequence) structure instead of a set to obtain an order of the elements. The insertion order differs, as can be seen in pseudocode.

For implementation purposes the elements, from which a subset has to be selected, are contained in some collection data structure. The mapping from integers to elements can be done quite easily by constructing an ordered data structure (if it is not already ordered) and interpreting the integer as the element's index. Therefore, generic methods can be implemented, that produce those random subsets.

Full example for Floyd's permutation algorithm with mapped elements (see Algorithm 3.7):

Index	1	2	3	4	5	6	7	8	9
Element	A	B	C	D	E	F	G	H	I

Execution ($k = 3$):
Start: $n = 9$, $S = \langle\rangle$
Iteration 1: $j = 7$, $t =$ Randomly chosen number 6, $S = \langle F \rangle$
Iteration 2: $j = 8$, $t =$ Randomly chosen number 3, $S = \langle C, F \rangle$
Iteration 3: $j = 9$, $t =$ Randomly chosen number 3 (conflict), $S = \langle C, I, F \rangle$

CHAPTER 4

Heuristic Solutions / Implementations

This chapter uses the theoretical foundation of the previous chapter and explains how the techniques are combined to solve the problem of this thesis. Some insight on the implementation is also given.

4.1 Overview

The generic approach used for this thesis is explained in pseudocode (see Algorithm 4.1), the heuristics mentioned there will be described in detail later within this chapter.

First the input graph is clustered hierarchically and then rings are searched according to the hierarchy. For clustering three algorithms were implemented: The Girvan-Newman algorithm, K-means-clustering and Kernighan-Lin-clustering. Girvan-Newman is already a hierarchical clustering technique, but the other two had to be slightly adopted to produce the desired dendrogram. Therefore they are applied in a recursive manner, which means that starting from one cluster (i.e., the whole graph) the same clustering technique is applied to each subcluster again until all clusters are split into singletons (i.e., containing only one node that cannot be clustered anymore).

The second step is to find rings in the dendrogram, according to the constraints of the Hierarchical Ring Network Problem. For this purpose three heuristics were developed. They are applied layer-by-layer (i.e., from layer 2 to layer 3). Per layer the heuristics are applied sequentially, and each heuristic is repeatedly executed as long as nodes can be assigned to rings. The first layer is treated separately because it is a Hamiltonian Cycle Problem and need not consider uplink constraints.

The first heuristic walks through the cluster hierarchy and tries to find valid rings within clusters. In each cluster that is investigated the Hamiltonian Path Problem is applied to check for rings (i.e., the lower chain of subrings) within this cluster. The order in which the heuristic visits each cluster was chosen to vary depending on the upper bound. Resulting rings strongly depend on the quality of the clustering.

Algorithm 4.1: Generic algorithm.

```
input  : A graph $G = (V, E)$. A weight function $w$.
output: A solution graph.
1  Hierarchical clustering;
   // Solve Hamiltonian Cycle Problem on $V_1$ nodes
2  Find Hamiltonian cycle in $V_1$;
3  if no Hamiltonian cycle found then
4      return Error!
5  end
6  for $l \leftarrow 2$ to $3$ do
       // Variate Ringsize Heuristic
7      Heuristic1($l$);
8      foreach ring $r$ in $V_{l-1}$ do
           // Subtour Heuristic
9          Heuristic2($l$,$r$);
10     end
       // Node Insertion Heuristic
11     Heuristic3($l$);
12     if nodes in $V_l$ left then
13         return Error!
14     end
15     Merge rings;
16 end
17 Improvement heuristics;
18 return solution graph;
```

Since it is already difficult to find valid solutions at all (because the graph is in general not complete), the second heuristic was intended to consume as many remaining nodes as possible. It starts from existing rings (from the previous heuristic) and tries to find chains that can be added to those rings. A depth-first-seach (DFS) based method is used to find the chains that start and end in rings from the upper layer. Because this DFS can have a very long running time, it had to be slightly modified.

To insert remaining nodes in rings the third heuristic tries to insert remaining nodes between edges of rings from the layer.

In some cases rather small rings are created. This can lead to difficulties in the lower layers to satisfy the dual-homing constraint. Therefore an improvement algorithm is applied, that merges rings at the end of each layer iteration.

The last phase after all nodes were assigned to rings is local improvement. Therefore, techniques like 2-Opt, 3-Opt and node exchange between rings were implemented. Node exchange is a local search technique that swaps two nodes from two distinct rings from the same layer if this improves the total cost.

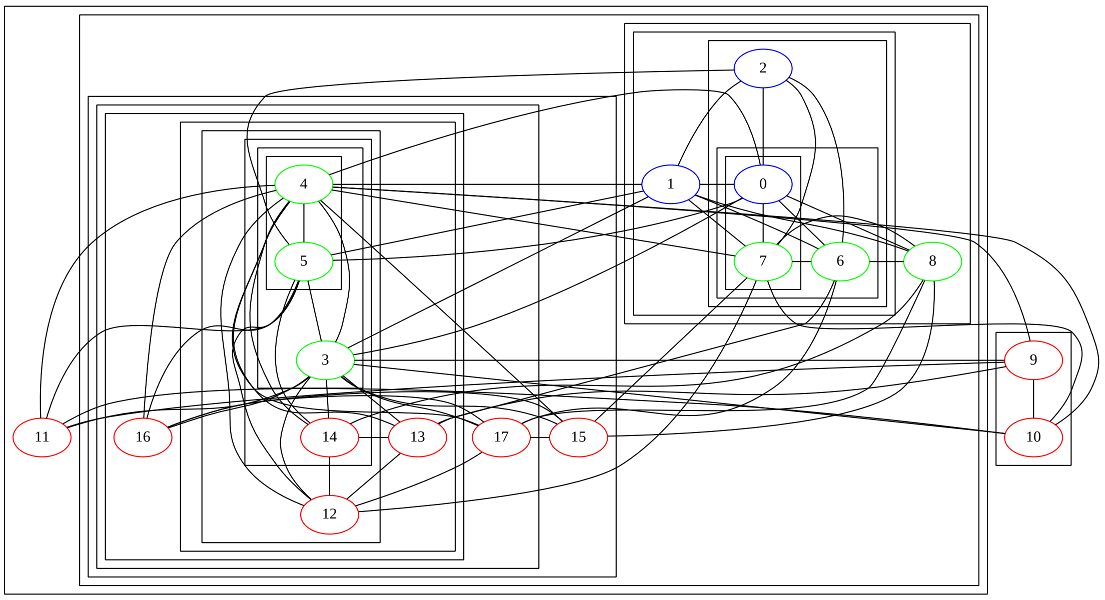


Figure 4.1: Girvan-Newman clustering of the sample instance (see Figure 1.1).

An example input instance is illustrated in Figure 1.1 and one possible solution is shown in Figure 1.2.

4.2 Hierarchical Clustering Techniques

The goal of the first step in Algorithm 4.1 is to cluster the input data hierarchically to obtain a dendrogram structure. This section explains how to obtain a dendrogram from an input graph.

Girvan-Newman Hierarchical Clustering

The Girvan-Newman algorithm already produces a dendrogram structure as a result of the clustering process. The output of the algorithm is a tree of regions, where each region is a cluster that contains the subclusters as subregions. This determines the hierarchy, because nodes of a subregion are hierarchically lower than the nodes of their parent region.

For illustration purposes the sample instance (see Figure 1.1) was clustered with this method and its result is shown in Figure 4.1. A corresponding dendrogram can be seen in Figure 4.2.

K-means Hierarchical Clustering

To get a hierarchy from K-means clustering, $k = 2$ was chosen. The initial graph is seen as one cluster, which is then clustered by the K-means algorithm. The result are subclusters of the initial single root-cluster. This process is repeated recursively for each of the subclusters until

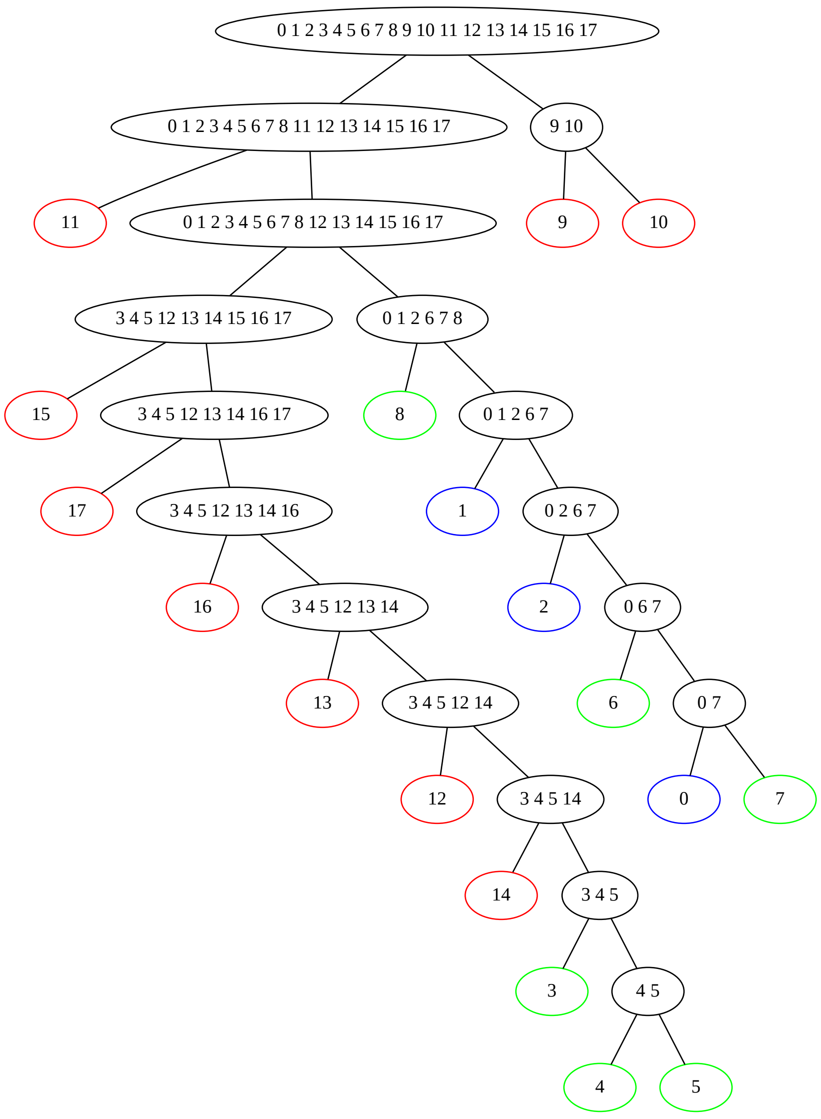


Figure 4.2: Dendrogram respresentation of the sample instance (see Figure 1.1) that was clustered by the Girvan-Newman algorithm.

only single node clusters are left. The result can also be seen as a binary tree that determines the dendrogram structure.

Kernighan-Lin Clustering

The Kernighan-Lin algorithm partitions a graph into two clusters. The process of recursive application of the algorithm is analogous to K-means hierarchical clustering described in the previous section.

4.3 Heuristics for Finding Rings in the Dendrogram

In this section various heuristics will be described that will use the dendrogram to extract rings for the solution.

Heuristic1: Variate Ringsize Heuristic

The intention of `Heuristic1` (see Algorithm 4.2 and Algorithm 4.3) is to determine clusters with a maximum number of b^u_k nodes at level k and to find a ring, i.e., solve the Hamiltonian Path Problem (HPP) within this cluster. If the HPP can be solved and the uplink constraints can be satisfied, a valid ring was found. Finding a Hamiltonian path is sufficient, but if a complete tour is found it is easier to choose the connections to the upper-level hubs. For larger chains the use of Ant Colony Optimization or Genetic Algorithms would lead to better tours, but since the chains are rather small, a simple approach is recommendable.

Since the graph is not complete, typical TSP heuristics cannot be used directly. A possible solution would be to extend the cluster to a complete subgraph by adding edges with very high cost (like ∞) and then to apply heuristics like the nearest-neighbour-algorithm. In this thesis the HPP was solved directly on the non-complete graph. It works by iterating over distinct pairs of nodes and applying DFS – starting at the first node of the pair – in each iteration. If all nodes can be visited in the iteration and the second node from the pair can be visited as last one a valid Hamiltonian path was found. If there is even a closing edge between the two nodes of the pair a Hamiltonian cycle was found. Note that the order in which the DFS iterates over neighbors is not determined. Ordering the neighbors (for example by their distance) may improve the time needed for computation.

An essential enhancement is the variation part. The maximum ring size constraint is tightened such that it varies from 2 to the bound b^u_k. This enhances the probability of finding a cluster that contains a valid ring.

A fundamental part of the heuristic is to find subdendrograms of subregions with nodes of the appropriate level. The algorithm uses depth-first-search until it reaches a subdendrogram of appropriate size, which is the varying bound. Whenever such a subdendrogram is reached, the algorithm tries to solve the HPP. If it is successful, it searches for valid uplinks (two uplinks with disjoint endpoints that link to the same parent ring). If such valid uplinks can be found the ring is added to the solution.

The rings found by this heuristic depend strongly on the quality of the clustering. With good clustering techniques this heuristic can find rings very efficiently.

Algorithm 4.2: `Heuristic1`: Variate Ringsize Heuristic

```
input  : The current level $k$.
output: All rings that were found in this heuristic.

1  for $varsize \leftarrow 2$ to $b_k^u$ do
       // Check TSP in cluster according to hierarchy with
          maximum size $varsize$ over unused nodes of $V_k$
2      ClusterDFS($cluster, k, varsize$);
3  end
4  return solution rings;
```

Algorithm 4.3: `ClusterDFS`: The DFS part of the Variate Ringsize Heuristic.

```
input  : The current (sub-)cluster $cluster$. The current level $k$. The current variating
         upper bound $varsize$.

1  fetch unused nodes of level $k$ in cluster;
2  if $|unused\ nodes| \leq varsize$ then
3      try to find Hamiltonian path in unused nodes;
4      if path found then
5          try to find 2 valid uplinks;
6          if uplinks found then
7              add ring to partial solution;
8              mark nodes as used;
9          end
10     end
11 else
12     foreach subcluster in cluster do
13         ClusterDFS($subcluster, k, varsize$);
14     end
15 end
```

Heuristic2: Subtour Heuristic

`Heuristic2`, see Algorithm 4.4, works on rings of the previous layer starting with layer 2. This means for layer 2 the previous ring is the backbone ring (i.e., layer 1 ring). For level 3 the set of rings contains all rings from the current partial solution that were found in level 2 (from all heuristics).

For each ring the heuristic tries to find rings of lower level that are connected to this ring. If the (parent) ring is of level $k-1$, all unused nodes of level k are investigated in the search.

The algorithm works as follows: For all distinct pairs of nodes (v, u) of level $k-1$ from the parent ring r, try to find a path of unused nodes of level k between v and u. A pseudocode for iterating over the pairs can be found in Algorithm 6.1.

To find a path simple depth first search is used. Only unused nodes from the layer $V_k \cup u$ are considered. The search starts at node v and looks for node u. A path has to fulfill the ring size constraint. This also means that the depth of DFS is restricted by the maximum ring size b_k^u. The ring size constraint b_k^l also prohibts that the DFS only takes the edge between v and u if it exists. To improve the possibility of finding a path by this DFS the neighbors are ordered according to their insertion cost, which is the cost of the edge of the current node to the neighbor plus the cost from the neighbor to the target node. Another enhancement is that neighbors within the same cluster are preferred.

Algorithm 4.4: `Heuristic2`: Subtour Heuristic

```
input  : The current level $k$. The parent ring $r$ to be investigated.
output: All rings that were found in this heuristic.
// Check if a path of unused nodes from $v$ to $u$, $u,v \in V_{k-1}$
   can be found in $V_k$
1  for $i \leftarrow 1$;$i < |parent\ ring\ r| - 1$; $i \leftarrow i + 1$ do
2      for $j \leftarrow i + 1$;$j < |parent\ ring\ r|$; $j \leftarrow j + 1$ do
3          $u \leftarrow i^{th}$ node in parent ring;
4          $v \leftarrow j^{th}$ node in parent ring;
5          try to find restricted path from $u$ to $v$ using DFS;
6          if path found then
7              add ring to partial solution;
8              mark nodes as used;
9          end
10     end
11 end
12 return solution rings;
```

Heuristic3: Node Insertion Heuristic

After the other heuristics were executed a repair heuristic is applied for unassigned nodes. `Heuristic3`, see Algorithm 4.5, tries to insert the remaining nodes into rings that were found by the previous heuristics, without violating any constraints.

There are many ways in which order the rings should be visited. The simplest way would be to iterate in the same order as the rings were added to the solution. To reduce the cost of adding a node a greedy method that visits the rings by the minimum distance between node and ring is recommendable, but it has the disadvantage, that it reduces the probability of finding rings in lower levels slightly. Another method is to order the rings by the number of nodes already contained and begin with the ring with least nodes. In this implementation the first method was chosen because it provided the best prerequisites for finding valid rings and could in some cases enhanced by the node exchange heuristic (similar to the second choice).

For each remaining node v from layer $k = l(v)$ the heuristic checks the lower chain of each ring R_k. If any edge $e = (i, j)$ in the lower chain can be replaced by edges $(v, i) \in E_k$ and $(v, j) \in E_k$ the node can be successfully inserted and marked as used.

Algorithm 4.5: `Heuristic3`: Node Insertion Heuristic

```
   input  : The current level $k$
 1 foreach unused node $v$ in $V_k$ do
       // Check if node can be inserted in any ring from level $k$
 2     foreach ring $r$ in partial solution from level $k$ do
 3         foreach edge $e$ in lower chain of ring $r$ do
 4             if node $v$ can be inserted between the start of $e$ and the end of $e$ then
 5                 insert node $v$ between endpoints of edge $e$ into ring $r$;
 6                 mark node $v$ as used;
 7                 remove edge $e$;
 8                 continue in first loop with next unused node;
 9             end
10         end
11     end
12 end
```

CHAPTER 5

Test Results and Critical Reflection

In the end the algorithm presented in Chapter 4 was intensely tested. Therefore, a set of test instances was generated. Table 5.1 shows, which properties were chosen for the instances. For example the first block ($|V| < 60$) can be explained as follows: The graph of a testinstance of this block less than 60 nodes. In layer 1 there are 3 to 5 nodes which is about 10 percent of all nodes. In the second layer there should be 10 to 15 nodes which makes about 20 to 30 percent of all nodes. The rest should be layer 3 nodes. Instances of this block should be solved with a level 2 bound-parameter of 5 (i.e., $b_2^u = 5$) and with a level 3 bound-parameter of 5 and 7, respectively.

Each single configuration was performed 30 times on a single core of an Intel Xeon E5540 with 2,53 GHz and 3 GB RAM.

Two kinds of test instances were generated, random instances and instances based on the TSPLIB. See section 5.1 for more information.

5.1 Test Instance Generation

Good test instances are needed to measure the quality of the algorithms and their solutions. To provide realistic instances the network needs to be big enough and the connectivity should have a natural distribution.

Random Instance Generator

The random instance generator assumes a normalized circular area, where the radius r is 1. The polar coordinate system is used to determine positions of nodes. If there should be n rings in level two, the graph is split into segments of $\frac{2\Pi}{n}$. Each of those segments is then split again into the according number of level 3 rings. After that, one ring is generated in each of the segments, where the coordinates of the nodes are generated randomly. The parameter values like the number of nodes in a chain, or the number of rings that are connected to a ring are also determined in a restricted random way.

$\lvert V\rvert < 60$	L1:	3 – 5 nodes	$\sim 10\%$
	L2:	10 – 15 nodes	$20\% - 30\%$
b_2^u	L2:	5	
b_3^u	L3:	5 / 7	
$60 \leq \lvert V\rvert < 100$	L1:	5 – 8 nodes	$\sim 7\%$
	L2:	20 – 30 nodes	$20\% - 30\%$
b_2^u	L2:	5/8	
b_3^u	L3:	5/8/12	
$100 \leq \lvert V\rvert < 200$	L1:	8 – 12 nodes	$\sim 5\%$
	L2:	35 – 50 nodes	$17\% - 25\%$
b_2^u	L2:	8 / 12	
b_3^u	L3:	8 / 12 / 15	
$200 \leq \lvert V\rvert < 500$	L1:	12 – 17 nodes	$\sim 3\%$
	L2:	80 – 120 nodes	$18\% - 24\%$
b_2^u	L2:	12 / 17	
b_3^u	L3:	12 / 17 / 20	

Table 5.1: Test instance specification.

Up to this point a valid solution is generated that has to be extended to a reasonable input instance.

A desireable property is 2-connectivity for each level. This can be achieved by computing two edge disjoint Minimum Spanning Trees (MSTs) on a complete version of the graph. The edges of both MSTs are added to the test instance.

Then additional random edges are added to the graph. Those are added within each Type R cluster, between nodes of the same layer and between nodes of different layers (i.e., more random uplinks).

At last all edges between V_1 and V_3 are removed.

An overview of the generated testinstances that were used for evaluation can be found in Table 6.1a.

TSPLIB Instances

TSPLIB95[1] is a collection of TSP instances. Also instances of the Hamiltonian Cycle Problem (HCP), Asymmetric Traveling Salesman Problem (ATSP), Sequential Ordering Problem (SOP) and Capacitated Vehicle Routing Problem (CVRP) are available.

[1]`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

The specification for the TSPLIB format is available in the online documentation. A simple adapter was implemented to import the TSPLIB files into the test instance generation framework.

To assign levels to the instances a simple partitioning algorithm is applied: K-means. For the first layer K-means is applied with specified k-value. The centroid nodes are assigned to layer 1. For the second layer this process is repeated for the whole graph, but nodes that were already assigned to layer 1 are not part of the consideration. The remaining nodes are assigned to layer 3.

Edges must be added too, since they are usually either not part of the TSPLIB instances or contain no valid solution for the Hierarchical Ring Network Problem. The layer 1 nodes build a complete graph. For layer 2 and 3 two edge disjoint MSTs are created to ensure 2-connectivity. Then edges are added to ensure a valid solution. As a last step random edges are added.

The instances have to be postprocessed, so that no edges between layer 1 and 3 exist.

The generated test instances can be found in Table 6.1b.

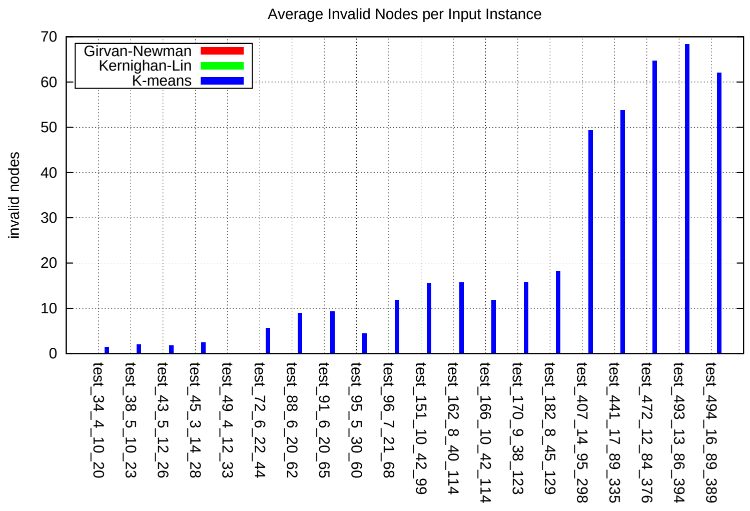


Figure 5.1: The average number of nodes that could not be assigned to rings, see Table 6.2.

5.2 Random Test Instances

The primary and very challenging goal in the Hierarchical Ring Network Problem is to find a valid solution. With the Girvan-Newman clustering and Kernighan-Lin each single run for each test instance could be solved by the proposed heuristics. With K-means 768 of 3000 test runs could not be solved completely. The average number of nodes that could not be assigned to rings is shown in Figure 5.1.

For the valid solutions the resulting costs are investigated next. An overview of the average costs (over valid solutions) are shown in Figure 5.2. It can be seen that Girvan-Newman clustering usually gives the lowest total cost, followed by K-means. Kernighan-Lin performs slightly worse, but with the smallest deviaton of the three algorithms. Therefore, the result of Kernighan-Lin is more predictable. The results for each of the clustering algorithms were also statistically tested with the student t-test at a significance level of 5% (see Table 6.4). Those tests confirm the observations that Girvan-Newman and Kernighan-Lin lead to better results on average.

The average computation time needed for the solutions is shown in Figure 5.3. Instances with up to 200 nodes could be solved within some seconds. On average Kernighan-Lin performed best. Girvan-Newman needed the most time, which was partially due to the fact that the clustering algorithm itself was the most expensive regarding computation effort.

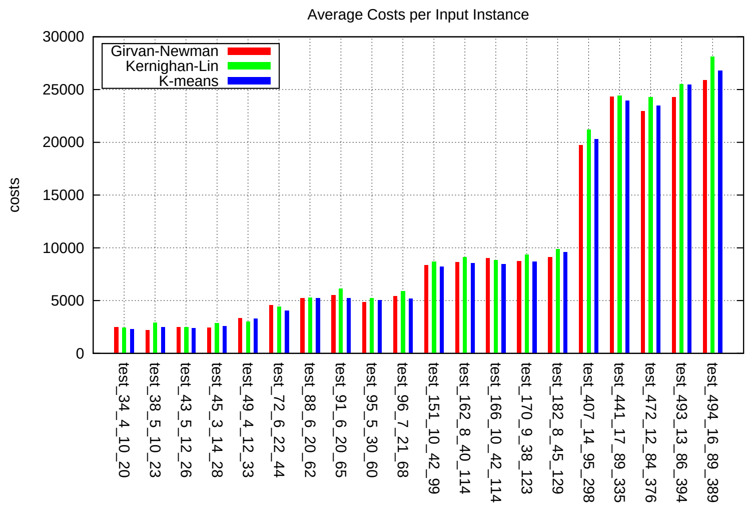


Figure 5.2: The average resulting cost of the solutions, see Table 6.4.

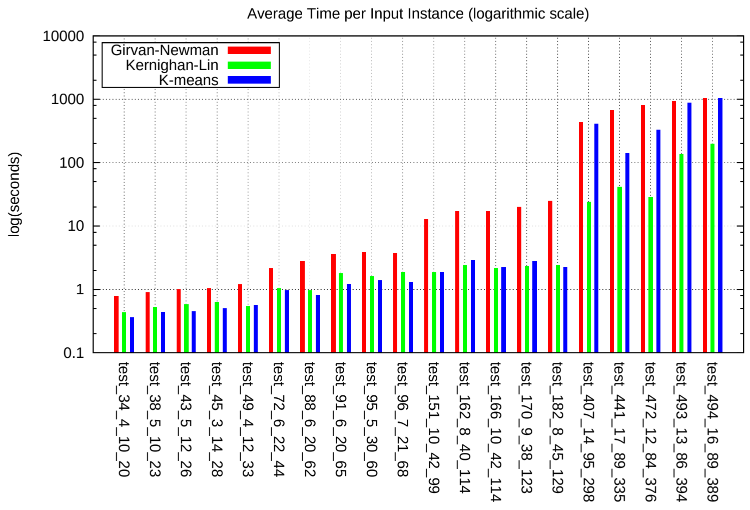


Figure 5.3: The average time in seconds needed to find a solution, see Table 6.5.

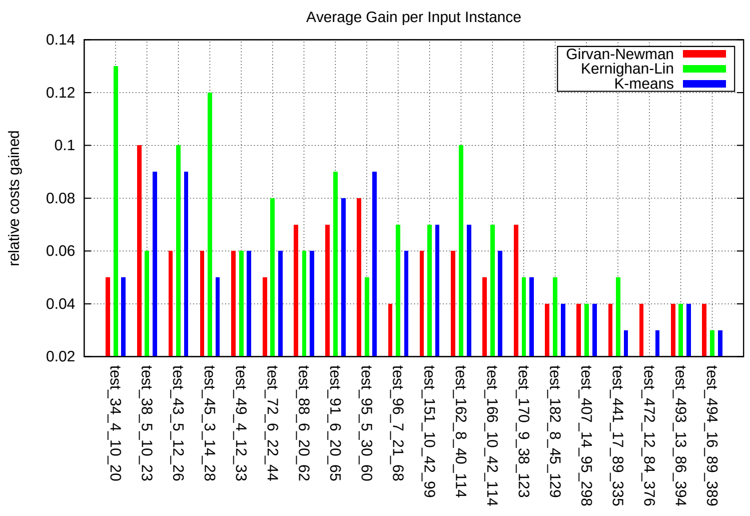


Figure 5.4: The average relative gain achieved (cf. Table 6.3 and Table 6.4). This shows how much the optimization step could improve the solution.

The cost reduction that could be achieved by the improvement heuristics is shown in Figure 5.4. Instances that were solved with the Kernighan-Lin clustering could usually be improved most, altough the total cost was higher than the total cost of other algorithms. The costs before and after the optimization were tested with the student t-test at a significance level of 5% (see Table 6.3 and Table 6.4). The relative performance of the clustering techniques is the same before and after the optimization. But it can also be seen from the statistical tests, that the difference between Girvan-Newman and Kernighan-Lin is smaller after the optimization so that the total costs are more similar.

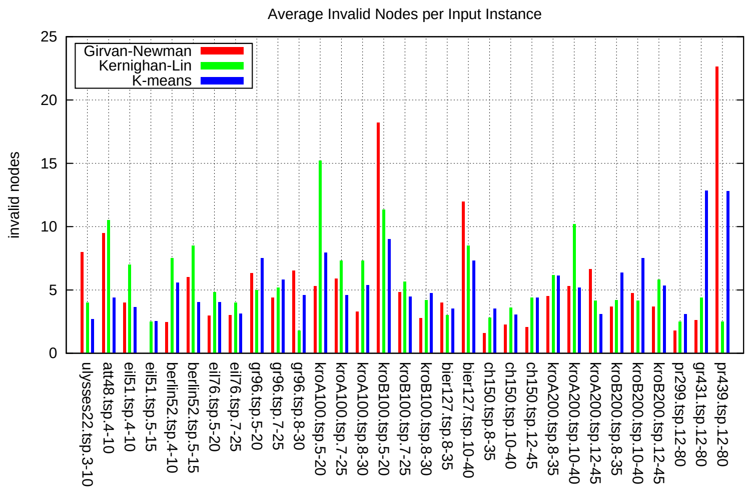


Figure 5.5: The average number of nodes that could not be assigned to rings, see Table 6.6.

5.3 TSPLIB-based Test Instances

A valid solution for the TSPLIB-based test instances could not be found in all cases. Here the student t-tests were omitted, because for some instances too few valid solutions were available to compare all the clustering heuristics. For each instance at least one of the clustering techniques lead to a valid solution. In Figure 5.5 the average number of nodes that could not be assigned to rings can be seen. For most test instances only few nodes could not be assigned to rings.

The average cost per instance can be seen in Figure 5.6. Missing bars indicate that the corresponding clustering algorithm could not even find a valid solution in 1 of the 30 test runs performed. Since some instances have very low total costs, an appropriate scaled view of those instances can be seen in Figure 5.7.

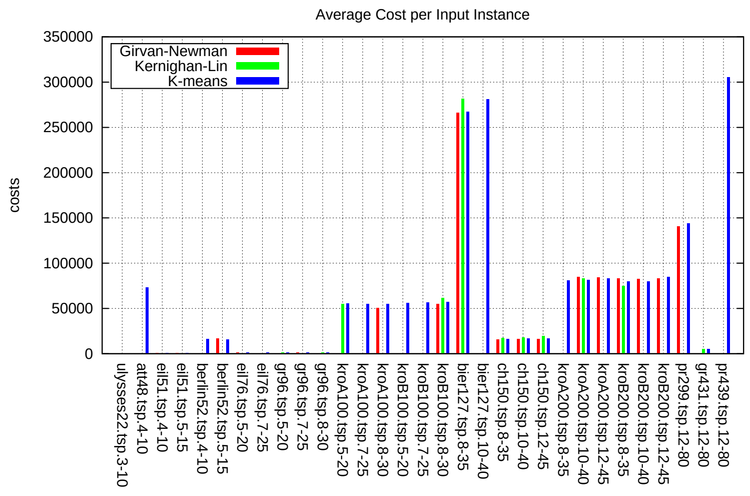


Figure 5.6: The average resulting cost of the solutions, see Table 6.4.

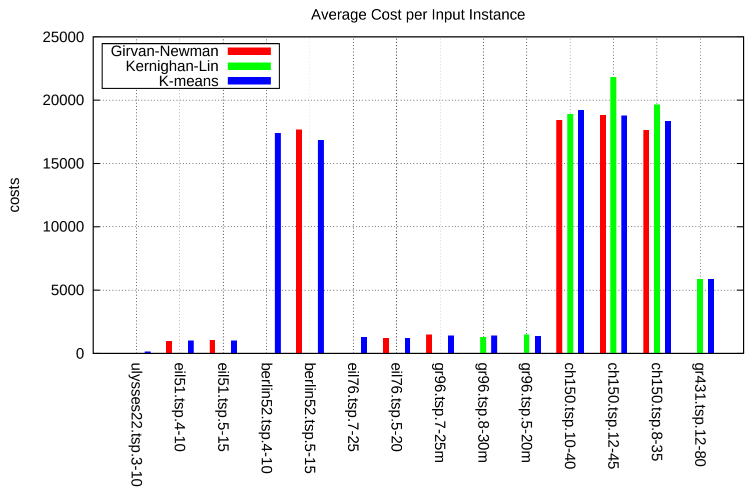


Figure 5.7: The average resulting cost of the solutions of instances with low total cost.

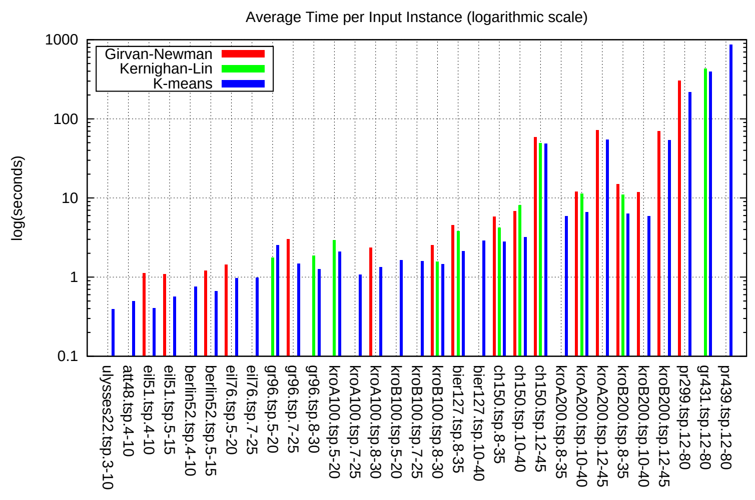


Figure 5.8: The average time in seconds needed to find a solution, see Table 6.9.

The time overview (see Figure 5.3) shows that the instances, where valid solutions were found, could be solved fast. Most of the smaller instances could be solved in less than 1 second.

The improvement gained by the local search (see Figure 5.9) is for most instances higher than the gain achieved for the random instances. In this case Kernighan-Lin cannot benefit as much from the improvement as it is the case with the random instances.

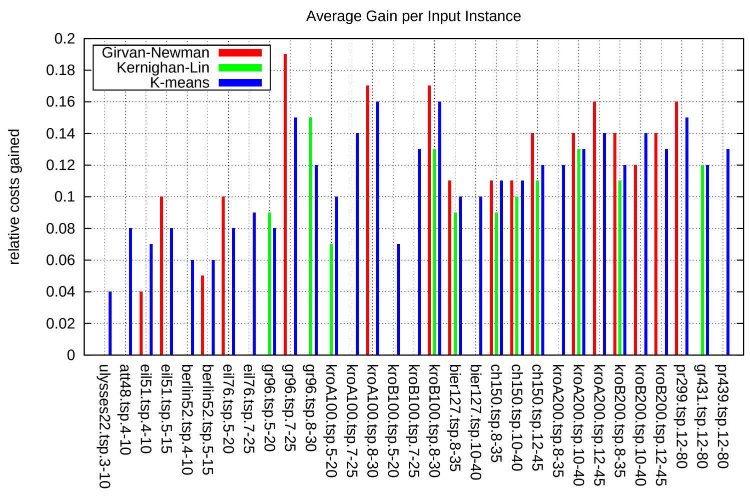


Figure 5.9: The average relative gain achieved(cf. Table 6.7 and Table 6.8). This shows how much the optimization step could improve the solution.

CHAPTER 6

Summary and Future Work

Using heuristics for finding hierarchical ring structures on non-complete graphs is a difficult task. In this thesis heuristic approaches for solving the Hierarchical Ring Network Problem were investigated. A formal definition for this problem was given. Some related work was explained too.

Clustering heuristics were implemented to get a hierachical structure of the input instances. This hierarchy was an important basis for the subsequent solution heuristics.

For clustering three algorithms were implemented, the Girvan-Newman algorithm, K-means and Kernighan-Lin clustering. K-means and Kernighan-Lin had to be adopted to get a hierarchical structure, because the original algorithms would have created only a single clustering layer. The result of this process was represented as a dendrogram.

Next three heuristics were implemented to form rings out of the nodes, according to the constraints. The Variate Ringsize Heuristic uses the hierarchy (from the previous clustering step) to find find rings within clusters. The clusters are chosen to match a variating ringsize constraint. More precisely, the variation takes place between 2 and the maximum bound b^u_k. The second heuristic uses depth first search (DFS) to find chains that can be attached to existing rings. This heuristic has shown to be inefficient in some cases in terms of performance. This gives room for improvements in future work. A third heuristic was implemented that tries to add single remaining nodes to rings, that could not be used in the first two heuristics. That heuristic is simple, but also necessary, because some instances could only be completely solved with the help of this heuristic.

As a last step local improvement heuristics were applied. 2-Opt was worth applying, altough not all instances could be improved by this post optimization. Merging rings was an important improvement, since many rings of small size were found. Accepting small rings was crucial to find solutions at all, but reduced the probability to find valid rings in the third layer. By merging rings this disadvantage could be lifted.

Using hierarchical clustering for network design is a noteworthy approach. Considering it for more applications would be worth the effort. Moreover, it would be interesting to adopt and apply the algorithms designed for this thesis to related problems.

In the case of the Hierarchical Ring Network Problem more clustering heuristics could be investigated. For some clustering candidates see [21]. There is still room for improvements for finding rings with support of cluster hierarchies. One could consider centrality measures like modularity to find good subclusters. Metaheuristics like the Variable Neighborhood Search (VNS) or Greedy Randomized Adaptive Search Procedures (GRASP) may enhance the generic solution algorithm. Clever use of clustering techniques can speed up heuristics for finding rings and improve the solutions.

Appendix

Input Data Format Description

As an input format the Graph Modelling Language (GML) is used. A detailed description of the format can be found at the website [1].

The format is text based and has a tree structure. It is recursively defined as a list of key/value pairs where the key is an identifier and the value can be of type integer, real, string or list(!), where a list has to be enclosed in square brackets.

A graph starts with the key `graph` and a list as value. This value (i.e. the list) contains `node` properties which must have an `id` property of type integer. A graph might also contain edges, that can have an `id` property and additionally a `source` and a `target` property that reference the id-values of previously defined nodes.

In this thesis nodes will have two additional properties. First the coordinates have to be specified with `x` and `y` properties within a `graphics` and a `center` property. The `graphics` property is often found in GML files as an information for the graphical representation. Here it is also needed to calculate distances. The center property is a way of expressing that the coordinate stands for the center of the node (and not for example for the upper left corner). The second additonal property `layer` specifies the layer every node belongs to. It can have a value between 0 and 2 which stands for the layer 1 to 3, resp. The edges do not require any additional properties.

Example file:

```
graph [
        node [
                id 0
                graphics [
                        center [
                                x 11.59845781564943
                                y 25.504837521414355
                        ]
                ]
                layer 0
        ]
```

[1] `http://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf`

```
        node [
                id 1
                graphics [
                        center [
                                x -13.616151400557554
                                y 23.787374664549787
                        ]
                ]
                layer 1
        ]

        edge [
                id 1
                source 0
                target 1
        ]
]
```

Additional Techniques

Loop Transformation

In imperative programming languages loops are usually written in iterative form, with the statements the language provides, like `while`, `do while` or `for`. Due to the fact that the algorithms used for this thesis use backtracking, this approach is often problematic, because one usually has to provide methods to undo the operations done so far.

Therefore, it is proposed to use recursive variants of loops. In each step only one iteration is done, so that the backtracking can be implemented much easier.

In the following a typical `while`-loop is considered to be implemented recursively:

The `while` statement has the form

```
while ( Expression ) Statement
```

Unwinding it one time would lead to the following structure:

```
if ( Expression ) {
  Statement
  while ( Expression ) Statement
}
```

To get recursion, some function is needed. Therefore, the loop is (without restriction) contained in a method:

```
function iterative() {
  while ( Expression ) Statement
}
```

This can now be written as a recursive function:

```
function recursive() {
  if ( Expression ) {
    Statement
    recursive()
  }
}
```

In practice additional context has to be provided by method parameters and also additional statements occur. Both should not impose problems on this transformation.

Distinct Pairs Iterator

Combinatorial enumeration of distinct pairs over a set or sequence is a task that is often needed. To avoid mistakes a simple skeleton code is provided in Algorithm 6.1.

Algorithm 6.1: Iterating over all distinct pairs of nodes in the list.

```
input : The list of input elements $list$.
1 for $i \leftarrow 1$ to $|list| - 1$ do
2     for $j \leftarrow i + 1$ to $|list|$ do
3         do something with distinct index pair $(i, j)$, where the pair of elements is
          $(list[i], list[j])$;
4     end
5 end
```

Test Result Tables

An overview of the tested instances is given in Table 6.1.

instance	$\|V\|$	$\|E\|$
test_34_4_10_20	34	143
test_38_5_10_23	38	169
test_43_5_12_26	43	183
test_45_3_14_28	45	193
test_49_4_12_33	49	210
test_72_6_22_44	72	341
test_88_6_20_62	88	423
test_91_6_20_65	91	426
test_95_5_30_60	95	464
test_96_7_21_68	96	450
test_151_10_42_99	151	768
test_162_8_40_114	162	816
test_166_10_42_114	166	834
test_170_9_38_123	170	854
test_182_8_45_129	182	914
test_407_14_95_298	407	2145
test_441_17_89_335	441	2301
test_472_12_84_376	472	2426
test_493_13_86_394	493	2512
test_494_16_89_389	494	2570

(a) Generated random test instances.

instance	$\|V\|$	$\|E\|$
ulysses22.tsp.3-10	22	67
att48.tsp.4-10	48	167
eil51.tsp.4-10	51	172
eil51.tsp.5-15	51	185
berlin52.tsp.4-10	52	197
berlin52.tsp.5-15	52	201
eil76.tsp.5-20	76	281
eil76.tsp.7-25	76	299
gr96.tsp.5-20	96	383
gr96.tsp.7-25	96	395
gr96.tsp.8-30	96	384
kroA100.tsp.5-20	100	429
kroA100.tsp.7-25	100	421
kroA100.tsp.8-30	100	420
kroB100.tsp.5-20	100	429
kroB100.tsp.7-25	100	432
kroB100.tsp.8-30	100	426
bier127.tsp.8-35	127	559
bier127.tsp.10-40	127	559
ch150.tsp.8-35	150	672
ch150.tsp.10-40	150	684
ch150.tsp.12-45	150	681
kroA200.tsp.8-35	200	940
kroB200.tsp.10-40	200	964
kroA200.tsp.12-45	200	1005
kroB200.tsp.8-35	200	952
kroA200.tsp.10-40	200	964
kroB200.tsp.12-45	200	1005
pr299.tsp.12-80	299	1566
gr431.tsp.12-80	431	2152
pr439.tsp.12-80	439	2194

(b) Generated test instances based on the TSPLIB.

Table 6.1: Test instance overview.

Results from the Random Instances

Table 6.2: The average number of invalid nodes per instance and their standard deviation are shown in this table. For Girvan-Newman and Kernighan-Lin all instances could be solved.

instance	$\|V\|$	$\|E\|$	Girvan-Newman		Kernighan-Lin		K-means	
			mean	dev	mean	dev	mean	dev
test_34_4_10_20	34	143	0.00	0.00	0.00	0.00	1.50	0.50
test_38_5_10_23	38	169	0.00	0.00	0.00	0.00	2.00	1.41
test_43_5_12_26	43	183	0.00	0.00	0.00	0.00	1.80	0.40
test_45_3_14_28	45	193	0.00	0.00	0.00	0.00	2.50	1.12
test_49_4_12_33	49	210	0.00	0.00	0.00	0.00	0.00	0.00
test_72_6_22_44	72	341	0.00	0.00	0.00	0.00	5.63	3.99
test_88_6_20_62	88	423	0.00	0.00	0.00	0.00	9.00	2.77
test_91_6_20_65	91	426	0.00	0.00	0.00	0.00	9.36	3.78
test_95_5_30_60	95	464	0.00	0.00	0.00	0.00	4.50	2.87
test_96_7_21_68	96	450	0.00	0.00	0.00	0.00	11.83	4.83
test_151_10_42_99	151	768	0.00	0.00	0.00	0.00	15.58	6.06
test_162_8_40_114	162	816	0.00	0.00	0.00	0.00	15.74	6.98
test_166_10_42_114	166	834	0.00	0.00	0.00	0.00	11.92	5.06
test_170_9_38_123	170	854	0.00	0.00	0.00	0.00	15.88	6.34
test_182_8_45_129	182	914	0.00	0.00	0.00	0.00	18.29	5.76
test_407_14_95_298	407	2145	0.00	0.00	0.00	0.00	49.34	9.73
test_441_17_89_335	441	2301	0.00	0.00	0.00	0.00	53.73	12.95
test_472_12_84_376	472	2426	0.00	0.00	0.00	0.00	64.68	16.85
test_493_13_86_394	493	2512	0.00	0.00	0.00	0.00	68.38	14.84
test_494_16_89_389	494	2570	0.00	0.00	0.00	0.00	62.08	13.50

Table 6.3: This table contains the average cost values (and their standard deviation) before the optimization. The clustering heuristics were compaired pairwise with the statistical student t-test on a significance level of 5%. p_A stands for the comparison between Girvan-Newman and Kernighan-Lin, p_B between Kernighan-Lin and K-means, and p_C between Girvan-Newman and K-means.

instance	$\|V\|$	$\|E\|$	Girvan-Newman		p_A	Kernighan-Lin		p_B	K-means		p_C
			mean	dev		mean	dev		mean	dev	
test_34_4_10_20	34	143	2592.32	206.15	<	2800.34	68.63	>	2399.71	201.33	>
test_38_5_10_23	38	169	2419.22	136.37	<	3079.41	71.93	>	2728.22	224.59	<
test_43_5_12_26	43	183	2613.77	128.24	<	2718.09	33.73	>	2623.52	155.11	$\approx$
test_45_3_14_28	45	193	2591.00	157.50	<	3264.10	82.50	>	2715.53	144.99	<
test_49_4_12_33	49	210	3517.08	395.33	>	3191.40	95.74	<	3467.01	336.20	$\approx$
test_72_6_22_44	72	341	4817.46	225.08	$\approx$	4804.19	489.00	>	4338.89	367.64	>
test_88_6_20_62	88	423	5656.04	371.71	$\approx$	5623.56	407.55	$\approx$	5601.14	442.94	$\approx$
test_91_6_20_65	91	426	5961.23	380.60	<	6755.90	375.13	>	5665.59	542.62	>
test_95_5_30_60	95	464	5251.04	373.36	<	5472.78	370.04	$\approx$	5522.57	523.15	<
test_96_7_21_68	96	450	5656.75	352.16	<	6313.49	491.00	>	5474.53	503.09	>
test_151_10_42_99	151	768	8843.29	506.19	<	9309.03	348.64	>	8769.04	448.31	$\approx$
test_162_8_40_114	162	816	9170.89	423.98	<	10106.08	378.97	>	9188.13	397.35	$\approx$
test_166_10_42_114	166	834	9497.89	396.48	$\approx$	9514.29	354.33	>	8940.10	522.41	>
test_170_9_38_123	170	854	9447.74	457.26	<	9872.69	595.08	>	9164.44	648.80	>
test_182_8_45_129	182	914	9515.14	155.78	<	10325.14	274.09	>	9947.06	612.15	<
test_407_14_95_298	407	2145	20568.98	566.98	<	21988.48	335.30	>	21092.78	975.90	<
test_441_17_89_335	441	2301	25263.67	920.29	<	25617.17	634.99	>	24757.12	968.42	>
test_472_12_84_376	472	2426	23841.81	438.76	<	24860.34	578.74	>	24207.02	966.78	<
test_493_13_86_394	493	2512	25208.76	1049.73	<	26687.84	810.11	>	26409.71	1556.31	<
test_494_16_89_389	494	2570	26929.54	882.96	<	29110.80	1119.83	>	27680.42	1455.72	<

Table 6.4: This table contains the average cost values (and their standard deviation) after the optimization. The clustering heuristics were compaired pairwise with the statistical student t-test on a significance level of 5%. p_A stands for the comparison between Girvan-Newman and Kernighan-Lin, p_B between Kernighan-Lin and K-means, and p_C between Girvan-Newman and K-means.

instance	$\|V\|$	$\|E\|$	Girvan-Newman			Kernighan-Lin			K-means		
			mean	dev	p_A	mean	dev	p_B	mean	dev	p_C
test_34_4_10_20	34	143	2466.61	200.47	≈	2450.25	144.92	>	2272.31	199.92	>
test_38_5_10_23	38	169	2190.95	181.62	<	2907.08	81.20	>	2467.81	195.51	<
test_43_5_12_26	43	183	2456.40	91.79	≈	2454.64	27.06	>	2381.13	150.88	>
test_45_3_14_28	45	193	2425.55	174.76	<	2858.26	52.56	>	2570.18	152.93	<
test_49_4_12_33	49	210	3324.98	392.35	>	3001.75	31.20	<	3263.44	330.35	≈
test_72_6_22_44	72	341	4562.97	217.29	>	4422.07	570.24	>	4066.32	365.14	>
test_88_6_20_62	88	423	5250.94	388.25	≈	5268.41	384.92	≈	5236.42	408.65	≈
test_91_6_20_65	91	426	5530.19	339.18	<	6115.62	301.98	>	5208.86	504.47	>
test_95_5_30_60	95	464	4850.54	372.32	<	5216.63	369.79	>	5039.75	503.30	<
test_96_7_21_68	96	450	5414.95	315.59	<	5881.73	523.29	>	5154.12	464.56	>
test_151_10_42_99	151	768	8335.13	457.80	<	8700.35	336.04	>	8189.90	393.37	>
test_162_8_40_114	162	816	8618.47	370.67	<	9124.67	379.17	>	8568.53	374.32	≈
test_166_10_42_114	166	834	9001.16	373.79	>	8828.83	330.99	>	8439.90	524.49	>
test_170_9_38_123	170	854	8757.22	418.50	<	9358.19	551.41	>	8702.07	578.01	≈
test_182_8_45_129	182	914	9094.63	166.13	<	9853.01	332.87	>	9567.29	609.29	<
test_407_14_95_298	407	2145	19736.36	530.69	<	21177.50	318.80	>	20275.79	889.13	<
test_441_17_89_335	441	2301	24313.60	895.51	≈	24425.53	647.75	>	23942.06	985.10	>
test_472_12_84_376	472	2426	22960.97	528.67	<	24276.89	447.93	>	23470.15	932.35	<
test_493_13_86_394	493	2512	24266.31	1065.72	<	25497.62	737.20	>	25464.88	1442.48	<
test_494_16_89_389	494	2570	25886.62	880.14	<	28106.46	1099.50	≈	26775.41	1419.14	<

Table 6.5: The average time needed for finding a valid solution is listed here. Values are scaled in milliseconds.

instance	$\|V\|$	$\|E\|$	Girvan-Newman		Kernighan-Lin		K-means	
			mean	dev	mean	dev	mean	dev
test_34_4_10_20	34	143	779.42	215.24	426.35	130.70	360.33	112.49
test_38_5_10_23	38	169	897.00	202.17	523.07	154.99	435.93	106.29
test_43_5_12_26	43	183	999.70	215.95	572.67	164.19	445.07	130.41
test_45_3_14_28	45	193	1025.97	264.12	634.55	180.81	495.76	155.90
test_49_4_12_33	49	210	1197.80	247.52	543.40	193.32	567.65	212.85
test_72_6_22_44	72	341	2151.73	412.15	1020.72	358.66	962.91	375.22
test_88_6_20_62	88	423	2825.67	367.36	956.17	347.06	812.82	280.13
test_91_6_20_65	91	426	3522.37	564.02	1778.87	824.56	1214.82	426.99
test_95_5_30_60	95	464	3832.99	479.19	1585.72	556.56	1372.24	567.16
test_96_7_21_68	96	450	3689.13	659.73	1893.84	908.76	1312.68	495.13
test_151_10_42_99	151	768	12753.56	953.71	1861.11	627.29	1872.24	1173.21
test_162_8_40_114	162	816	16840.67	1110.27	2371.99	872.27	2922.67	2813.73
test_166_10_42_114	166	834	17064.56	1038.38	2165.17	595.70	2202.84	1453.69
test_170_9_38_123	170	854	19859.08	1443.88	2322.46	620.24	2749.57	2621.55
test_182_8_45_129	182	914	24914.21	1464.62	2421.59	838.50	2260.06	1243.20
test_407_14_95_298	407	2145	433817.37	32292.82	24021.82	19330.93	412309.07	2212615.53
test_441_17_89_335	441	2301	663820.41	38670.66	41232.09	21988.88	139776.22	216975.98
test_472_12_84_376	472	2426	802348.48	39844.76	28302.61	21940.88	331089.58	1638352.81
test_493_13_86_394	493	2512	922606.57	47059.81	135591.82	194549.87	880866.70	2279635.13
test_494_16_89_389	494	2570	1035306.86	54769.42	196222.99	303653.02	1041982.37	2782957.35

Results from the TSPLIB-based Instances

Table 6.6: The average number of invalid nodes per instance and their standard deviation are shown in this table.

instance	$\lvert V \rvert$	$\lvert E \rvert$	Girvan-Newman		Kernighan-Lin		K-means	
			mean	dev	mean	dev	mean	dev
ulysses22.tsp.3-10	22	67	8.00	1.00	4.00	1.00	2.70	1.78
att48.tsp.4-10	48	167	9.50	2.50	10.50	3.50	4.39	2.86
eil51.tsp.4-10	51	172	4.00	0.00	7.00	1.00	3.64	2.28
eil51.tsp.5-15	51	185	0.00	0.00	2.50	0.50	2.55	1.68
berlin52.tsp.4-10	52	197	2.45	1.40	7.50	1.50	5.56	4.20
berlin52.tsp.5-15	52	201	6.00	0.00	8.50	3.50	4.04	2.14
eil76.tsp.5-20	76	281	2.99	1.14	4.83	1.57	4.03	2.68
eil76.tsp.7-25	76	299	3.00	1.15	4.00	2.38	3.12	2.24
gr96.tsp.5-20	96	383	6.34	4.13	5.00	2.68	7.50	4.93
gr96.tsp.7-25	96	395	4.40	3.32	5.17	2.19	5.82	3.78
gr96.tsp.8-30	96	384	6.51	4.71	1.80	0.75	4.58	3.44
kroA100.tsp.5-20	100	429	5.30	3.00	15.20	7.60	7.96	4.85
kroA100.tsp.7-25	100	421	5.88	3.55	7.33	2.56	4.59	3.32
kroA100.tsp.8-30	100	420	3.29	1.86	7.33	2.13	5.39	3.96
kroB100.tsp.5-20	100	429	18.22	4.09	11.33	4.42	9.03	7.38
kroB100.tsp.7-25	100	432	4.83	2.74	5.67	3.64	4.49	2.93
kroB100.tsp.8-30	100	426	2.79	1.18	4.20	2.64	4.74	3.78
bier127.tsp.8-35	127	559	11.97	7.95	8.50	2.81	7.32	4.27
bier127.tsp.10-40	127	559	4.00	4.05	3.00	1.90	3.51	2.96
ch150.tsp.8-35	150	672	2.25	0.43	3.60	2.58	3.03	2.24
ch150.tsp.10-40	150	684	2.07	1.51	4.40	2.65	4.39	2.69
ch150.tsp.12-45	150	681	1.60	0.80	2.80	1.83	3.51	2.84
kroA200.tsp.8-35	200	940	5.31	2.24	10.20	3.87	5.17	3.75
kroA200.tsp.10-40	200	964	6.66	3.78	4.17	2.27	3.09	1.86
kroA200.tsp.12-45	200	1005	4.50	3.40	6.17	4.88	6.15	4.62
kroB200.tsp.8-35	200	952	4.75	1.48	4.17	3.98	7.53	6.74
kroB200.tsp.10-40	200	964	3.69	1.80	5.83	3.89	5.35	4.47
kroB200.tsp.12-45	200	1005	3.67	2.05	4.20	3.12	6.38	5.92
pr299.tsp.12-80	299	1566	1.80	1.60	2.50	1.12	3.11	2.61
gr431.tsp.12-80	431	2152	2.61	0.95	4.40	2.94	12.86	10.26
pr439.tsp.12-80	439	2194	22.64	20.39	2.50	1.61	12.81	8.70

Table 6.7: This table contains the average cost values (and their standard deviation) before the optimization. For these instances a statistical test was not performed, because too few valid solutions were available for a statistically significant statement. Missing values indicate, that no valid solution was found with the corresponding clustering technique.

instance	$\|V\|$	$\|E\|$	Girvan-Newman		Kernighan-Lin		K-means	
			mean	dev	mean	dev	mean	dev
ulysses22.tsp.3-10	22	67					139.29	1.52
att48.tsp.4-10	48	167					79829.42	2847.31
eil51.tsp.4-10	51	172	955.63	0.00			988.25	34.74
eil51.tsp.5-15	51	185	1029.26	8.88			1005.25	39.92
berlin52.tsp.4-10	52	197					17391.68	396.38
berlin52.tsp.5-15	52	201	17659.09	0.00			16845.10	301.33
eil76.tsp.5-20	76	281	1191.89	30.38			1201.44	30.96
eil76.tsp.7-25	76	299					1283.12	39.48
gr96.tsp.5-20	96	383			1275.39	0.00	1337.27	59.16
gr96.tsp.7-25	96	395	1470.70	24.84			1397.71	43.69
gr96.tsp.8-30	96	384			1472.15	0.00	1386.63	70.22
kroA100.tsp.5-20	100	429			58415.57	0.00	61696.27	2965.25
kroA100.tsp.7-25	100	421					63880.86	4204.33
kroA100.tsp.8-30	100	420	61055.40	2760.80			65012.50	4006.51
kroB100.tsp.5-20	100	429					60702.60	3332.30
kroB100.tsp.7-25	100	432					64867.73	3944.71
kroB100.tsp.8-30	100	426	66489.93	3399.81	70425.01	0.00	67952.05	3098.30
bier127.tsp.8-35	127	559	300284.54	8709.27	310733.37	0.00	295720.75	14806.29
bier127.tsp.10-40	127	559					311425.84	8964.34
ch150.tsp.8-35	150	672	17633.88	165.55	18892.15	0.00	18348.63	450.84
ch150.tsp.10-40	150	684	18395.72	244.73	19649.49	0.00	18777.73	511.83
ch150.tsp.12-45	150	681	18808.69	0.00	21815.09	0.00	19210.06	714.57
kroA200.tsp.8-35	200	940					92110.19	4388.56
kroA200.tsp.10-40	200	964	98762.71	1947.35	96122.10	0.00	93422.87	3042.44
kroA200.tsp.12-45	200	1005	100334.28	1120.95			96620.25	3626.95
kroB200.tsp.8-35	200	952	97032.00	585.51	83834.45	0.00	90279.74	3088.04
kroB200.tsp.10-40	200	964	94205.60	4828.49			93253.56	2734.53
kroB200.tsp.12-45	200	1005	96745.32	2475.37			97947.67	4372.56
pr299.tsp.12-80	299	1566	168499.60	2141.54			169112.22	7826.76
gr431.tsp.12-80	431	2152			5845.89	0.00	5837.78	312.91
pr439.tsp.12-80	439	2194					349691.59	6821.11

Table 6.8: This table contains the average cost values (and their standard deviation) after the optimization. For these instances a statistical test was not performed, because too few valid solutions were available for a statistically significant statement. Missing values indicate, that no valid solution was found with the corresponding clustering technique.

instance	$\|V\|$	$\|E\|$	Girvan-Newman		Kernighan-Lin		K-means	
			mean	dev	mean	dev	mean	dev
ulysses22.tsp.3-10	22	67					134.01	2.26
att48.tsp.4-10	48	167					73334.12	2873.18
eil51.tsp.4-10	51	172	918.38	0.00			921.05	33.10
eil51.tsp.5-15	51	185	926.76	7.05			921.20	33.51
berlin52.tsp.4-10	52	197					16367.85	657.32
berlin52.tsp.5-15	52	201	16806.68	0.00			15779.50	253.80
eil76.tsp.5-20	76	281	1068.30	19.20			1111.04	37.27
eil76.tsp.7-25	76	299					1167.91	39.99
gr96.tsp.5-20	96	383			1165.98	0.00	1231.78	54.51
gr96.tsp.7-25	96	395	1194.06	18.54			1191.70	30.60
gr96.tsp.8-30	96	384			1255.34	0.00	1216.58	42.73
kroA100.tsp.5-20	100	429			54562.78	0.00	55475.48	1797.28
kroA100.tsp.7-25	100	421					54673.38	2172.82
kroA100.tsp.8-30	100	420	50549.72	950.83			54572.78	1622.92
kroB100.tsp.5-20	100	429					56097.94	1790.72
kroB100.tsp.7-25	100	432					56475.88	1972.91
kroB100.tsp.8-30	100	426	54807.79	1818.98	61485.37	0.00	57300.69	1989.45
bier127.tsp.8-35	127	559	266143.87	2424.85	281308.72	0.00	267210.45	8912.25
bier127.tsp.10-40	127	559					280890.62	8280.53
ch150.tsp.8-35	150	672	15748.55	134.39	17117.63	0.00	16362.82	438.19
ch150.tsp.10-40	150	684	16359.75	303.18	17653.24	0.00	16674.03	415.89
ch150.tsp.12-45	150	681	16225.87	0.00	19394.48	0.00	16968.55	568.45
kroA200.tsp.8-35	200	940					80681.50	2341.70
kroA200.tsp.10-40	200	964	84877.65	1043.81	83283.31	0.00	81348.50	2206.81
kroA200.tsp.12-45	200	1005	84315.91	521.23			82788.86	2456.76
kroB200.tsp.8-35	200	952	83315.39	585.51	75002.83	0.00	79603.75	2120.49
kroB200.tsp.10-40	200	964	82344.44	2003.62			79925.20	2303.95
kroB200.tsp.12-45	200	1005	83292.68	2498.21			84901.59	3736.16
pr299.tsp.12-80	299	1566	140888.81	1981.84			143990.83	5334.69
gr431.tsp.12-80	431	2152			5164.10	0.00	5164.62	306.84
pr439.tsp.12-80	439	2194					305452.69	8985.96

Table 6.9: The average time needed for finding a valid solution is listed here. Values are scaled in milliseconds. Empty cells indicate, that no valid solution was found with the corresponding clustering technique.

instance	$\lvert V \rvert$	$\lvert E \rvert$	Girvan-Newman		Kernighan-Lin		K-means	
			mean	dev	mean	dev	mean	dev
ulysses22.tsp.3-10	22	67					393.82	63.11
att48.tsp.4-10	48	167					498.11	128.38
eil51.tsp.4-10	51	172	1124.17	188.53			406.00	127.76
eil51.tsp.5-15	51	185	1098.72	245.12			571.11	156.20
berlin52.tsp.4-10	52	197					753.33	234.89
berlin52.tsp.5-15	52	201	1213.20	148.36			667.33	157.56
eil76.tsp.5-20	76	281	1434.00	372.23			969.07	419.63
eil76.tsp.7-25	76	299					982.20	415.77
gr96.tsp.5-20	96	383			1764.80	517.24	2526.33	896.76
gr96.tsp.7-25	96	395	3042.17	1072.26			1483.35	626.91
gr96.tsp.8-30	96	384			1876.23	711.13	1256.29	482.36
kroA100.tsp.5-20	100	429			2951.10	1163.51	2099.75	860.32
kroA100.tsp.7-25	100	421					1080.97	413.59
kroA100.tsp.8-30	100	420	2340.43	663.15			1337.30	508.55
kroB100.tsp.5-20	100	429					1629.16	600.75
kroB100.tsp.7-25	100	432					1597.07	684.15
kroB100.tsp.8-30	100	426	2542.73	705.47	1561.40	646.42	1463.95	537.66
bier127.tsp.8-35	127	559	4528.10	640.24	3803.83	1077.28	2123.56	499.66
bier127.tsp.10-40	127	559					2873.33	763.96
ch150.tsp.8-35	150	672	5848.57	442.34	4233.70	1319.93	2808.27	631.01
ch150.tsp.10-40	150	684	6853.78	661.18	8196.17	1570.85	3195.87	715.06
ch150.tsp.12-45	150	681	58334.81	669.10	49185.70	1244.60	48985.03	1258.87
kroA200.tsp.8-35	200	940					5909.42	2410.54
kroA200.tsp.10-40	200	964	12013.37	686.94	11380.40	2191.00	6595.05	1906.44
kroA200.tsp.12-45	200	1005	71852.48	2082.34			54663.54	1913.65
kroB200.tsp.8-35	200	952	15057.07	688.18	11049.40	2038.93	6330.35	1150.47
kroB200.tsp.10-40	200	964	11906.62	1320.17			5920.29	1541.49
kroB200.tsp.12-45	200	1005	69448.70	1953.91			53501.21	2016.59
pr299.tsp.12-80	299	1566	305541.69	110524.01			217407.84	172403.10
gr431.tsp.12-80	431	2152			433551.97	8976.75	395681.80	291287.06
pr439.tsp.12-80	439	2194					864114.33	1015110.56

Bibliography

[1] A. Balakrishnan, T. L. Magnanti, and P. Mirchandani. The multi-level network design problem. Working papers 3366-91., Massachusetts Institute of Technology (MIT), Sloan School of Management, 1991.

[2] J. Bentley and B. Floyd. Programming pearls: a sample of brilliance. *Communications ACM*, 30:754–757, 1987.

[3] D. Bertsimas and R. Weismantel. *Optimization over Integers*. Dynamic Ideas, 1st edition, 2005.

[4] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

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

[6] D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[7] C. T. Fan, M. E. Muller, and I. Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57:387–402, 1962.

[8] B. Fortz. *Design of Survivable Networks with Bounded Rings*. PhD thesis, Universite libre de Bruxelles, 2000.

[9] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Science*, 99:7821–7826, 2002.

[10] T. G. Jones. A note on sampling a tape-file. *Communications ACM*, 5:343, 1962.

[11] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[12] J. G. Klincewicz. Hub location in backbone/tributary network design: a review. *Location Science*, 6(1-4):307 – 335, 1998.

[13] G. Laporte and I. R. Martín. Locating a cycle in a transportation or a telecommunications network. *Networks*, 50:92–108, 2007.

[14] E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, New York, 1985.

[15] C. Y. Lee and S. J. Koh. A design of the minimum cost ring-chain network with dual-homing survivability: A tabu search approach. *Computers & Operations Research*, 24(9):883 – 897, 1997.

[16] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the travelling-salesman problem. *Operations Research*, 21:498–516, 1973.

[17] R. Salman, V. Kecman, Q. Li, R. Strack, and E. Test. Fast k-means algorithm clustering. *CoRR*, abs/1108.1351, 2011.

[18] T. Thomadsen. *Hierarchical Network Design*. PhD thesis, Technical University of Denmark, 2005.

[19] J. S. Vitter. An efficient algorithm for sequential random sampling. In *ACM Transactions Mathematical Software*, volume 13, pages 58–67, New York, NY, USA, 1987. ACM.

[20] C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131:325–372, 2004.

[21] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: the state of the art and comparative study. *CoRR*, abs/1110.5813, 2011.