Master Thesis

Models for the Steiner Tree Packing Problem

Michael Sausen

Master Thesis DKE 13-20

Thesis submitted in partial fulfillment
of the requirements for the degree of Master of Science
of Operations Research at the Department of Knowledge
Engineering of the Maastricht University

Thesis Committee:

Dr. Jean Derks
Prof. Dr. Marco Lübbecke

Maastricht University
Faculty of Humanities and Sciences
Department of Knowledge Engineering
Master Operations Research

November 21, 2013

Abstract

The Steiner tree packing problem is a long studied problem in combinatorial optimization. In contrast to many other problems, where an enormous progress has been made in the practical problem solving, the Steiner tree packing problem remains very difficult. Most heuristics schemes are ineffective and even finding feasible solutions is already *NP*-hard. What makes this problem special, is that in order to reach an overall optimal solution non-optimal solutions to the underlying *NP*-hard Steiner tree problems must be used. Any non-global approach to the Steiner tree packing problem is likely to fail. Integer programming is currently the best approach for computing optimal solutions.

The goal of this master thesis is to give a survey of models relating to the Steiner tree packing problem from the literature. In addition, a closer look at a model for the switchbox routing problem in VLSI-Design will be given.

Contents

List of Figures **III**

1 Introduction **1**

2 Foundations **3**
2.1 Graph Theory . . . 3
2.1.1 Graphs . . . 3
2.1.2 Walks . . . 4
2.1.3 Trees . . . 4
2.1.4 Planar Graphs . . . 5
2.1.5 Grid Graphs . . . 6
2.1.6 Hypergraphs . . . 7
2.1.7 Flows and Cuts . . . 8
2.2 Linear Programming . . . 11

3 Steiner Tree Packing Problem **13**
3.1 Steiner Tree Problem . . . 13
3.2 Steiner Tree Packing Problem . . . 16
3.3 Complexity . . . 18
3.4 Related Problems . . . 19
3.4.1 Edge-Disjoint Steiner Tree Packing . . . 19
3.4.2 Vertex-Disjoint Steiner Tree Packing . . . 22
3.4.3 Element-Disjoint Steiner Tree Packing . . . 25

4 VLSI-Design **27**
4.1 Routing Problem in VLSI-Design . . . 27
4.2 Decomposition of the Routing Problem . . . 28
4.2.1 Global Routing . . . 28

4.2.2 Detailed Routing 30
4.3 Related Problems 33
4.3.1 Switchbox routing 33

5 Integer Programming Models **37**
5.1 The Undirected Cut Formulation 37
5.2 The Directed Cut Formulation 39
5.3 The Explicit Formulation 40
5.4 The Multicommodity Flow Formulation 41

6 Approach **44**
6.1 A Cutting Plane Algorithm 45
6.1.1 Mathematical Formulations 45
6.1.2 The Algorithm 47
6.1.3 Primal Heuristic 48
6.1.4 Results 54

7 Implementation **57**
7.1 The Algorithm 57
7.2 Code 59
7.3 Results 69
7.4 More Examples 72

8 Conclusion **74**

Bibliography **76**

List of Figures

2.1 An undirected graph with 6 vertices and 7 edges 4
2.2 A labeled tree 5
2.3 Planar and Non-Planar Graphs 6
2.4 A Grid Graph 7
2.5 A hypergraph 8
2.6 A graph showing flow and capacity 9
2.7 A cut on a graph 10

3.1 An arbitrary graph 15
3.2 Steiner tree 15
3.3 Miminimum Steiner tree 16
3.4 Gadget for high degree nodes [Aaz08] 21
3.5 Routing paths via the gadget [Aaz08] 21

4.1 Dividing a routing area in subareas 29
4.2 Channel routing area 30
4.3 Switchbox routing area 31
4.4 General routing area 31
4.5 Knock-knee model 32

6.1 A minimal enclosing rectangle 51
6.2 An example for criterion 3 53
6.3 An example for criterion 5 53

7.1 Structure of a grid graph 60
7.2 Graph with not-used edges 68
7.3 Computing Time 70
7.4 Sample switchbox routing 71

Chapter 1

Introduction

The enormous progress in the development of electronic circuits have let this become a backbone of modern technology. For example, manufacturing, communication, measurement or control systems of the present generation are no longer conceivable without electronic controls.

An electronic circuit is a complex interconnection of semiconductor devices (so-called transistors). This interconnection is the physical implementation of a logic function. The current technical capabilities allow the integration of several million transistors on a few square centimeters. The magnitude and complexity of the problems that arise in the design of such circuits, provide a great challenge for the developers of electronic circuits. Naturally, methods from the fields of engineering, computer science and mathematics are required to solve these problems. For example, a number of problems which arise in the design of electronic circuits can be formulated as a combinatorial optimization problem. For this reason, various methods can be used from this mathematical field.

The problem of minimizing the length of a network or graph is one of the oldest optimization problems in mathematics, for example the Steiner tree packing problem. Many famous mathematicians of the past dealt with this problem.

This thesis gives a detailed introduction to the problem. First important terms and notations from the fields of graph theory and linear programming are summarized. During the thesis these terms and notations are required.

In order that the Steiner Tree Packing Problem can be defined, a definition of the underlying Steiner tree problem and its historical aspects are given in chapter 3. Then a mathematical definition of the Steiner tree packing problem is given. Furthermore, there are given some complexity-theoretic statements about this problem. After that, this thesis gives a short overview of different algorithms from the literature, that are related to the Steiner tree packing problem. These include the edge-disjoint, node-disjoint and element-disjoint Steiner tree packing problem.

The following part of the thesis deals with the application of the Steiner tree packing problem (see Chapter 4). The motivation for the study of Steiner tree packing problem comes from the design of electronic circuits. A part of the problem occurring there is the so-called routing problem. First the reader will get familiar with the technical terms occurring in the design of electronic circuits. Then some variants of the routing problem are presented, in which different restrictions are given. In addition, it is made clear at which points the study of the Steiner tree packing problem can make a contribution to solve the routing problem.

Subsequently, in order to come up with some mathematical formulations for solving the Steiner tree packing problem, a survey of different integer programming models is given in chapter 5, which focuses on computational aspects of the problem. This type of mathematical programming is a way to achieve a good solution.

In order to show how far an approach, which uses linear programming, can solve a problem of practical application, the cutting plane algorithm for solving the weighted Steiner tree packing problem is explained. First, the basic procedure of such a method is explained in chapter 6. This algorithm was developed to solve the switchbox routing problem by using integer linear programming and a heuristic to determine a feasible solution. Moreover, different test examples from the literature are used for validating this approach, and the results are discussed afterwards.

A similar heuristic as described in the cutting plane approach was implemented with Matlab. In chapter 7 a detailed description and an evaluation of the developed heuristic is provided.

Chapter 2

Foundations

This chapter serves to explain important terms and descriptions which are required during the thesis. First some terms of the graph theory will be discussed and then a small insight into the linear programming is given. In addition, in this thesis different terms from the complexity theory are used. It is assumed that the reader is familiar with these terms and is reffered to the book of Goldreich [Gol10].

2.1 Graph Theory

In mathematics graph theory is the study of graphs, which are models representing pairwise relations between objects. Graphs are one of the prime objects of study in discrete mathematics. The following important graph-theoretic terms and descriptions are presented. In addition, some definitions and notations are introduced. For more detailed graph-theoretic summary the reader is refered to [Ruo13] or [BM08].

2.1.1 Graphs

A graph is an ordered pair $G = (V, E)$ comprising a set V of vertices or nodes together with a set E of edges or lines. Every edge has two endvertices and is said to connect or join the two endvertices. An edge can thus be defined as a set of two vertices. The two endvertices of an edge are also said to be adjacent to each other.

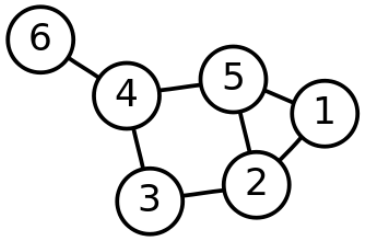


Figure 2.1: An undirected graph with 6 vertices and 7 edges

A graph may be undirected (Figure 2.1), meaning that there is no distinction between the two vertices associated with each edge, so that (v_i, v_j) and (v_j, v_i) denote the same edge, or its edges may be directed from one vertex to another. A graph can be a weighted graph $G = (V, E, c)$ with an edge funtion $c : E \to \mathbb{R}$. For each edge terms like costs, weight, lenght, etc. are used which can also be denoted by $c(v_i, v_j)$ or c_{ij}. A graph is called finite if V and E are finite, otherwise G is infinite. In this thesis only finite graphs are used.

2.1.2 Walks

A walk is an alternating sequence of vertices and edges, beginning and ending with a vertex. Each vertex is an end vertex of the edge that precedes it and the edge that follows it in the sequence. A walk is closed if its first and last vertex are the same and open if they are different. A walk is called a *path* if no vertices and no edges are repeated. If a closed walk has no repeated vertices or edges, it is also called a *cycle*. Furthermore a cycle with directed edges where all the edges are traveled in the same direction is called a *circuit*.

2.1.3 Trees

A tree is a connected acyclic graph. A graph is called acyclic, if it has no cycles. An acyclic graph is also called a forest. For a directed tree it is furthermore assumed that each vertex has at most one incoming edge. The *degree* of a vertex is the number of edges incident to the vertex, such that a

vertex of degree 1 is called a *leaf* or a pendant vertex. An edge incident to a leaf is a leaf edge or a pendant edge. A non-leaf vertex is an internal vertex. Sometimes, one vertex of the tree is distinguished, and called the *root*, in which case the edges have a natural orientation, towards or away from the root. Such a tree is called a *routed tree*.

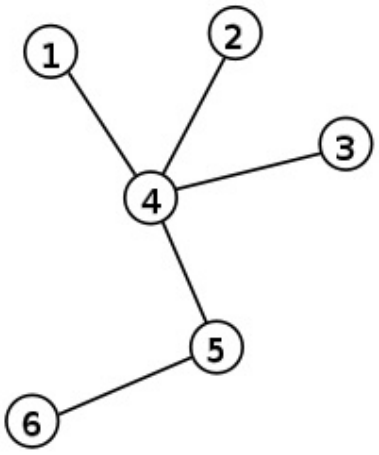


Figure 2.2: A labeled tree

In figure 2.2 a labeled tree with 6 vertices and 5 edges is shown. Nodes 1, 2, 3, and 6 are leaves, while 4 and 5 are internal vertices.

2.1.4 Planar Graphs

A graph $G = (V, E)$ can be drawn in a plane, by representing each vertex by a point in the plane, and assigns a curve or a straight line to each edge, that connects the points, which are representing the two end vertices of the edge. A graph is called planar if it can be drawn in the plane such that two edges (more precisely, they representing curves) intersect at most in their end vertices.

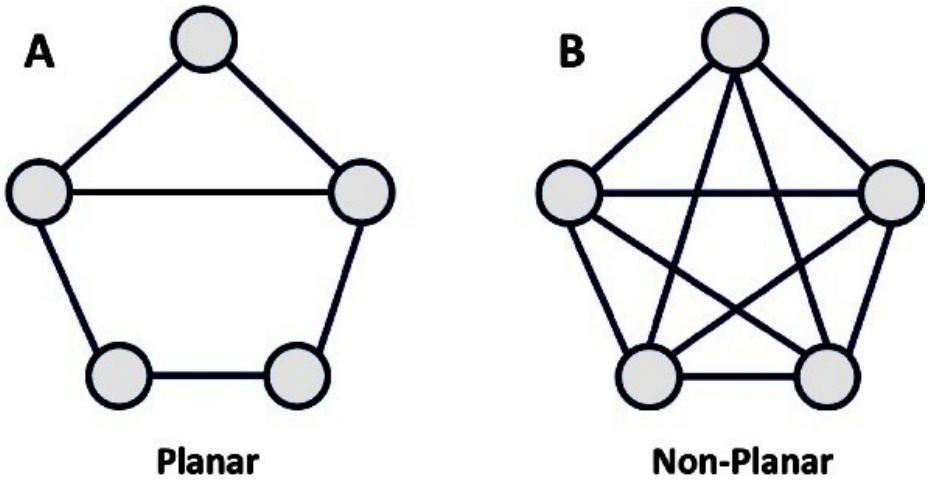


Figure 2.3: Planar and Non-Planar Graphs

Such a representation of a planar graph (Figure 2.3 (A)) in the plane is called an embedding of G in the plane. There can be different embeddings for a graph G. The example B is a non-planar graph, because the edges intersect with each other, it cannot reconfigured in a manner that would make it planar.

When a graph is drawn without any intersection, any cycle that surrounds a region without any edges reaching from the cycle into the region forms a *face*. Two faces on a planar graph are adjacent if they share a common edge. A face that contains the entire graph is called *outer face*. In this thesis the outer surface of the graph G is denoted by the edge set O_G.

2.1.5 Grid Graphs

A graph $G = (V, E)$ is called a grid graph, if the set of vertices V can be numbered, such that $V \subseteq \{(i, j) | i = 1, ..., h; j = 1, ..., b\}$ with $(h, b) \in \mathbb{N} \times \mathbb{N}$. The number h is called the grid height and the number b, the grid width of G. It is obvious that each grid graph is planar.

For a grid graph G it is normally assumed a certain embedding, in which each edge of the form $[(i,j),(i+1,j)]$ is represented by an vertical line and each edge of the form $[(i,j),(i,j+1)]$ is represented by an horizontal line (Figure 2.4).

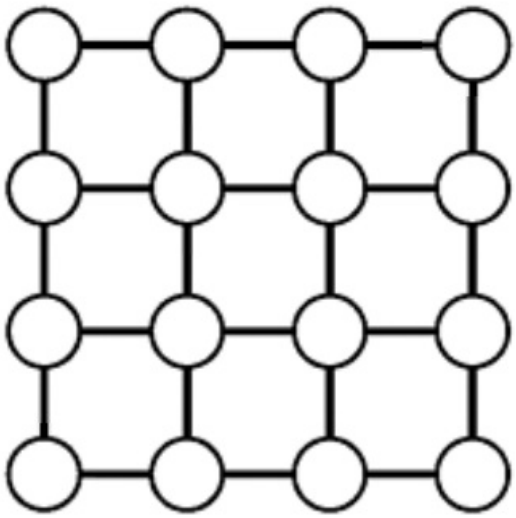


Figure 2.4: A Grid Graph

2.1.6 Hypergraphs

A hypergraph is a generalization of a graph in which an edge can connect any number of vertices. Formally, a hypergraph $\mathcal{H}$ is a pair $\mathcal{H} = (V, \varepsilon)$ where V is the node set of $\mathcal{H}$ and ε is a collection of non-empty subsets of V. A subset $Z \in \varepsilon$ is called a *hyperedge* of $\mathcal{H}$. Given a partition $\mathcal{P} = \{V_1, ..., V_t\}$ of V into non-empty subsets, a hyperedge $Z \in \varepsilon$ is called a crossing hyperedge if it intersects at least with two subsets of P and otherwise it is called an internal hyperedge. The number of sets V_i in $\mathcal{P}$ can be denoted by $|\mathcal{P}|$, and $e(\mathcal{P})$ denotes the number of crossing hyperedges corresponding to the partition $\mathcal{P}$. A hypergraph is *bipartite* if and only if its vertices can be partitioned into two classes U and V in such a way that each hyperedge with cardinality at least two contains at least one vertex from both classes.

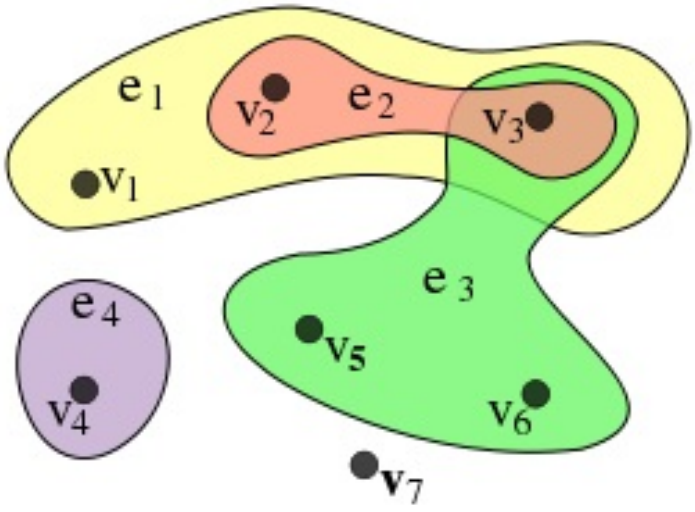


Figure 2.5: A hypergraph

In figure 2.5 a hypergraph with seven vertices and four partitions is given. The hyperedges e_1, e_2 and e_3 are crossing hyperedges and $e_4$is an internal hyperedge.

2.1.7 Flows and Cuts

Flows

In graph theory, each edge of a network has a capacity and each edge can receive a so-called *flow*. The amount of flow on an edge cannot exceed the capacity of the edge. A flow must satisfy the restriction, that the amount of flow into a node is equals to the amount of flow out of it, except when it is a *source*, which is the initial node, or a *sink*, which is the target node.

Given a directed graph $G = (V, E)$ in which every edge $(u, v) \in E$ has a non-negative capacity $c(u, v)$. If $(u, v) \notin E$, it is assumed that $c(u, v) = 0$. A flow with the source s and the sink t is a function $f : E \to \mathbb{R}_{\geq 0}$ with the following three properties for all nodes u and v:

1) The capacity constraints $f(u, v) \leq c(u, v)$, which presupposes that the flow along an edge cannot exceed its capacity. The flow from u to v must be the opposite of the flow from v to u.

2) The conservation constraint implies that $\sum_{w \in V} f(u, w) = \sum_{w \in V} f(w, u)$, unless $u = s$ or $u = t$. The flow to a node is zero, except for the source node, which produces flow, and the sink node, which consumes flow.

Notice that $f(u, v)$ is the flow from u to v. If the graph represents a physical network, and if there is a real flow of, for example four units from u to v, and a real flow of three units from v to u, then $f(u, v) = 1$.

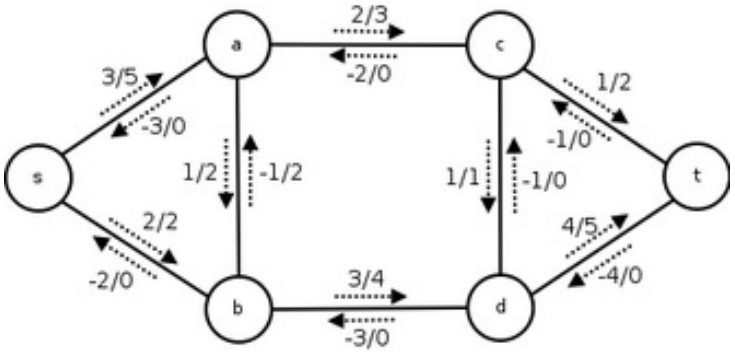


Figure 2.6: A graph showing flow and capacity

In figure 2.6 a graph is shown, where the flow and capacity of an edge is denoted by f/c.

Cuts

A *cut* is a partition of the vertices of a graph into two disjoint subsets. The *cut-set* of the cut is the set of edges, whose end points are in different subsets of the partition. Edges are said to be crossing the cut, if they are in its cut-set.

In a network the cut requires the source and the sink to be in different subsets. In this case the cut-set only consists of edges going from the source's side to the sink's side.

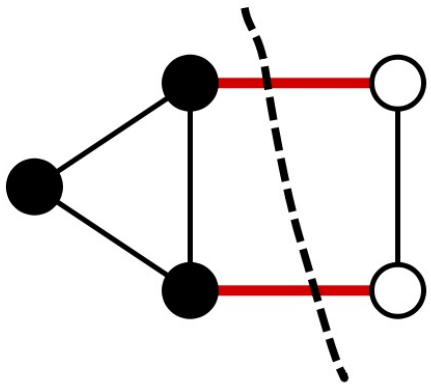


Figure 2.7: A cut on a graph

In figure 2.7 a cut is shown, where one partition has black vertices and the other partition has white vertices. The red edges are in the cut-set, because their end points are in different subsets of the partition.

2.2 Linear Programming

Linear programming [Van08] (shorter: LP) is a mathematical method for determining a way to achieve the best outcome in a given mathematical model for some list of requirements represented as linear relationships. Linear programming is a specific case of mathematical programming.

More formally, linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints. Let $Ax \leq b$ ($Ax = b$) be a system of linear inequalities (equalities), with A a real $m \times n$ - matrix and $b \in \mathbb{R}^m$. The set of solutions $\{x \in \mathbb{R}^n | Ax \leq b\}$ from a system of inequalities is then called the feasible region of the linear program, which is a *polyhedron*. It is a intersection of finitely many half spaces, each of which is defined by a linear inequality. Its objective function is a real-valued affine function defined on this polyhedron. A linear programming algorithm finds a point in the polyhedron, where this function has the smallest (or largest) value if such a point exists.

Linear programs are problems, that can be expressed in the standard form:

minimize $c^T x$
subject to $Ax \leq b$
and $x \geq 0$

x represents the vector of variables, which has to be computed, c and b are vectors of (known) coefficients and A is a (known) matrix of coefficients. The expression, which has to be minimized in this case is called the objective function (here $c^T x$). The inequalities $Ax \leq b$ and $x \geq 0$ are the constraints which specify a convex polyhedron over which the objective function is to be optimized.

If all of the unknown variables are required to be integers, then the problem is called an *integer programming* or *integer linear programming* (shorter: ILP) problem. In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in many practical situations *NP*-hard. A special case, $0 - 1$ integer linear programming, in which unknowns are binary, is one of the Karp's 21 *NP*-complete problems.

Linear Programming Relaxation

The *linear programming relaxation* [MG07] (shorter: LP-Relaxation) of an integer program is the problem, that arises by replacing the constraint $x_i \in \{0, 1\}$, that each variable must be either 0 or 1 by a weaker constraint $0 \leq x_i \leq 1$, that each variable belong to the interval $[0, 1]$.

The resulting relaxation is a linear program, hence the name. The relaxation technique transforms an *NP*-hard optimization problem (integer programming) into a related problem, that is solvable in polynomial time (linear programming); the solution to the relaxed linear program can be used to gain information about the solution to the original integer program.

Chapter 3

Steiner Tree Packing Problem

In this chapter, a mathematical definition of the Steiner Tree Packing Problem is given. In addition, some designations and properties are introduced, which will be useful later on. Some statements about the complexity of the problem are given and, moreover, related problems from the literature are studied.

3.1 Steiner Tree Problem

Before the Steiner Tree Packing Problem can be defined, the notion of Steiner tree is needed. The actual Steiner tree problem will be discussed. Also the historical aspects of this problem are offered.

The Steiner problem is the combinatorial variant of the much older Euclidean Steiner problem, which asks for a minimal tree that connects a given set of points in the plane. This problem has already been discussed, before 1640 by Fermat [Str10]:

Given three points in the plain, find a fourth point T, such that the length from this point to the three given points is minimal.

Torricelli solved this problem and the point has since been known as the Torricelli-point. Torricelli's method was to construct equilateral triangles on each side of the triangle made up by the original points. Circles circumscribing the equilateral triangles intersect in the point that is sought.

Jacob Steiner (1796-1863) considered a generalization of this problem for n points (the generalized Fermat problem), which is the origin of the (Euclidean) Steiner problem. These two problems are identical only in the case $n = 3$. In 1836 the Steiner problem is believed to have been presented first by Gauß. The first (terminating) algorithm for the Euclidean Steiner problem was given by Melzak [Mel61]. More information on the Euclidean Steiner problem and its history is contained in Hwang et al. [HRW92].

In 1971 Hakimi and Levin, independently of each other, gave a formulation of the Steiner tree problem [Hak71, Lev71]. Since then hundreds of articles have been published concerning different variants of this problem.

(Steiner Tree Problem)

Given: An undirected graph $G = (V, E)$ with edge cost $c : E \to \mathbb{R}_+$, and a set of vertices $T \subseteq V$, called terminals.

Problem: Find an edge set S spanning T of minimum cost $c(S) := \sum_{e \in S} c(e)$.

Given a set of terminals, a tree is called a Steiner tree, if all leafs are terminals. It combines all terminals and it is allowed to use non-terminal vertices of the graph (the *Steiner nodes*) in order to determine the Steiner tree.

To illustrate the difference between a graph, a Steiner tree and a minimum Steiner tree, all three variants are shown graphically in an example below.

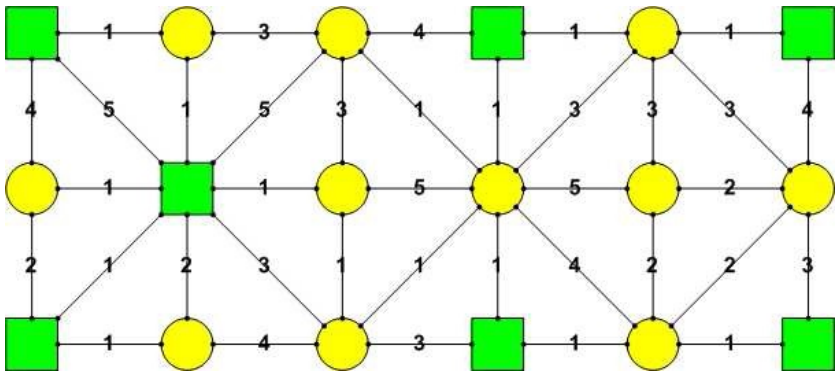


Figure 3.1: An arbitrary graph

In figure 3.1 is shown a graph G, which has a set of vertices and a set of edges. The green squares are terminal symbols, the yellow circles are the other nodes and the connecting lines have to be seen as edges with assigned length.

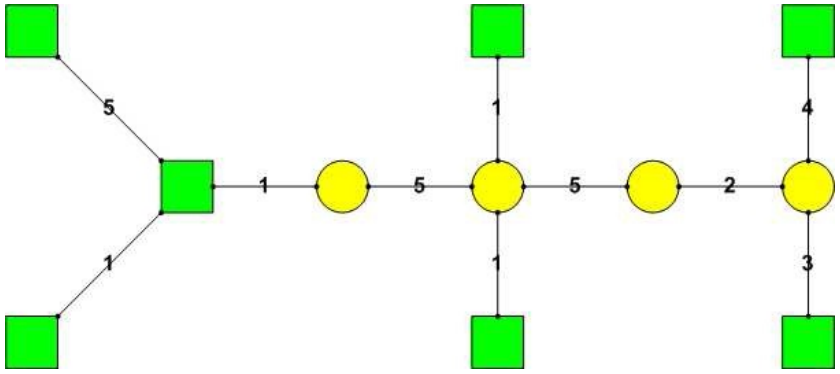


Figure 3.2: Steiner tree

This graph (Figure 3.2) is a Steiner tree, since it contains all the terminals and also all the leaves of the tree are occupied by terminal symbols.

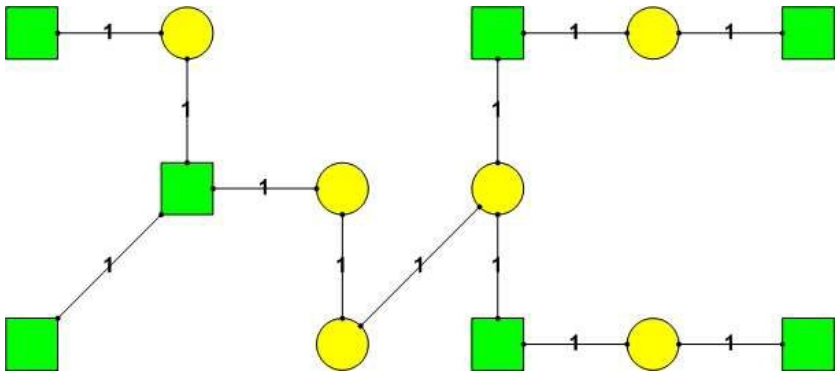


Figure 3.3: Miminimum Steiner tree

This graph is a minimal Steiner tree, meaning that the sum of the edge costs is minimal. All the leaves of the tree are terminals.

The Steiner tree problem in graphs is known to be *NP*-complete. But there are some special cases, which can be solved in polynomial-time. This includes the case to find the shortest connection between two terminals (the shortest path problem). The other case is, when all vertices are terminals, then the problem is just to find a minimum spanning tree.

3.2 Steiner Tree Packing Problem

The Steiner tree packing problem is an extension of the Steiner tree problem. Instead of having one set of terminals, we have k non-empty disjoint sets $T_1, ..., T_k$, called *nets*, that have to be *packed* into the graph simultaneously, which means that the resulting edge sets $S_1, ..., S_k$ have to be edge-disjoint.

(Steiner Tree Packing Problem)

Given: A planar graph $G = (V, E)$ and a list of terminal sets $\mathcal{N} = (T_1, ..., T_N)$, where $N \geq 1$ and $T_k \subseteq V$ for all $k = 1, ..., N$.

Problem: Find for each $k = 1, ..., N$, an edge set $S_k \subseteq E$, which spans T_k, such that $(V(S_k), S_k)$ is a Steiner tree and $S_1, ..., S_N$ are pairwise edge-disjoint.

The list of vertex sets $\mathcal{N}$ is called a netlist. The number N denotes the cardinality of the netlist. A vertex set T_k is called a terminal set and the elements of T_k terminals. All vertices, which are used by the edge set S_k, are denoted by $V(S_k)$. It is often used *net* k instead of terminalset T_k. The notations net, terminal, etc. originate from the VLSI-Design (Chapter 4), which is the main application area of the Steiner tree packing problem. Usually the edge capacities $c_e = 1$, if $c_e > 1$, the problem will be extended to multiple layers. An instance of the Steiner tree packing problem is then defined by the triple $(G, \mathcal{N}, c)$. That means, when refering to an instance of a Steiner tree packing problem, so a triple $(G, \mathcal{N}, c)$ is always meant.

Often there are additionally given weighted edges, searching for a solution of this problem, which is minimal with respect to the weightings. This weighted problem is called the weighted Steiner tree packing problem.

(Weighted Steiner Tree Packing Problem)

Given: A planar graph $G = (V, E)$, with non-negative weighted edges $w_e \in \mathbb{R}_+$. A list of terminal sets $\mathcal{N} = (T_1, ..., T_N)$, with $N \geq 1$ and $T_k \subseteq V$ for all $k = 1, ..., N$.

Problem: Find for each $k = 1, ..., N$, an edge set $S_k \subseteq E$, which spans T_k, such that $(V(S_k), S_k)$ is a Steiner tree. The resulting edge sets $S_1, ..., S_k$ have to be edge-disjoint while the weighted sum of $\sum_{k=1}^{N} \sum_{e \in S_k} w_e$ have to be minimal.

The notation $(G, \mathcal{N}, c, w)$ is an instance of the weighted Steiner tree packing problem and is also called a weighted instance of the Steiner tree packing problem.

3.3 Complexity

It is not surprising, that the (weighted) Steiner tree packing problem is *NP*-complete (*NP*-hard). In this context the decision problem, whether the problem is solvable or not, is *NP*-complete and the optimization problem, where an optimal solution is searched, is *NP*-hard. This problems has a lot of special cases, which are already *NP*-complete or *NP*-hard. In the following a small selection of special cases is given.

The case $N = 1$ will be considered for the weighted Steiner tree packing problem, such that the Steiner tree problem (Section 3.1) will be obtained. Karp [Kar10] showed that the Steiner tree problem is *NP*-hard. Garey and Johnson [GJ77] show that the so-called rectangular Steiner tree problem is also *NP*-hard. For this problem several points are given in a plane and the goal is to find a tree of minimum cost connecting the points with horizontal and vertical lines. From this it follows directly, that the Steiner tree problem also remains *NP*-hard, when the underlying graph G is planar or a grid graph and all edges have the same weights.

For example the Steiner tree packing problem contains the problem of finding k edge-disjoint paths. This problem is obtained if it is additionally required that all terminal sets have a cardinality of two. Kramer and van Leeuwen [Kv80] have provided a proof of *NP*-completeness for this problem. Another special case of the Steiner tree packing problem arises, when only two terminal sets are specified. Also in this case this problem is *NP*-complete [KPS90].

Moreover, there are *NP*-complete problem statements for very specific instances, which very often occur in applications derived from the VLSI-Design (Chapter 4). Let $G = (V, E)$ be a complete, rectangular grid graph and $\mathcal{N}$ a netlist, whose terminals are only located on the four borders of the grid graph. Sarrafzadeh [Sar87] showed that it is *NP*-complete to decide, whether there is a Steiner tree packing for such a problem. This result remains valid, if each terminal set has at most cardinality k for a fixed $k > 3$. For the case $k = 2$, this problem can be solved in polynomial time.

These examples should give an impression about the actual difficulty of the problem. It can be said that the (weighted) Steiner tree problem is *NP*-hard. There are also special cases, which can be solved in polynomial time.

3.4 Related Problems

The Steiner tree packing problem is a long studied topic. Here a short overview of different ideas and algorithms for the Steiner tree packing problem is given. The various algorithms were divided roughly into the edge-disjoint, the vertex-disjoint and the element-disjoint cases.

3.4.1 Edge-Disjoint Steiner Tree Packing

The edge-disjoint case is usually treated, when referring to a Steiner tree packing problem. Most of the edge-disjoint Steiner tree packing problems in planar graphs are *NP*-complete. Even when restricted to paths, which means, that all nets have only two terminals, the problem remains *NP*-complete. In the following some algorithms of this problem are presented.

(Max-Steiner-Tree-Packing Min-Steiner-Cut Algorithm)

This approximation algorithm for the Steiner tree packing problem was given by Lau [Lau07]. The goal is to find a largest collection of edge-disjoint Steiner trees of an undirected graph G. In this approximation algorithm the main idea is an approximate min-max relation between the maximum number of edge-disjoint Steiner trees, that each connect the terminals and the minimum size of an edge-cut that disconnect some pair of terminals.

Given an undirected graph G and a subset of vertices $S \subseteq V(G)$, called terminals. The terminals S are also called black vertices, while the vertices $V(G) \backslash S$ are called white vertices (Steiner vertices). An edge is called a white edge if it connects two white vertices.

The algorithm consists of two parts. In the first step the given graph G with ℓ white edges will be transformed into at most $\ell + 1$ graphs $\{G_1, ..., G_{\ell+1}\}$, such that each graph has no white edge by using Edmonds' matroid partition algorithm [Edm65]. In the second step for each subgraph G_i with $i = 1, ..., \ell + 1$, all Steiner trees are determined. Afterwards the solutions of the subgraphs are combined.

Then the Steiner tree packing problem is formulated by the following linear program, which was introduced by Jain, Mahdian and Salacatipour [JMS03]:

$$\textbf{maximize} \quad \sum_{T \in \mathcal{T}} x_T$$

$$\textbf{subject to} \quad \sum_{\substack{T \in \mathcal{T} \\ e \in E}} x_T \leq c_e \qquad \forall T \in \mathcal{T} : x_T \geq 0$$

In this formulation $\mathcal{T}$ denotes the collection of all Steiner trees in a graph $G = (V, E)$, and c_e is the given capacity of the edge $e \in E$. The size of $\mathcal{T}$ grows exponentially to the number of vertices in V.

(Edge-Disjoint Steiner Trees)

This approximation algorithm for packing edge-disjoint Steiner trees in planar graphs was introduced by Aazami [Aaz08]. Based on the element-disjoint Steiner tree packing algorithm (Section 3.4.3), which also was given by Aazami, this egde-disjoint version of the problem on planar graphs was developed. The goal is to find a maximum cardinality set of edge-disjoint Steiner trees, such that each tree contains every terminal node.

A graph is said to be k-edge connected if it remains connected whenever at most k edges are removed. The following statement is the main result of this algorithm:

Let $G = (V, E)$ be an undirected planar graph, let $R \subseteq V$ be the set of terminals, and assume that R is k-edge connected. Then there are at least $\lfloor \frac{k}{4} \rfloor - 1$ edge-disjoint Steiner trees in G. Moreover, there is an algorithm with a running time of $\mathcal{O}(|V|^{4.5})$ that finds at least $\lfloor \frac{k}{4} \rfloor - 1$ edge-disjoint Steiner trees in G.

The graph G is first reduced to a planar graph G' with Steiner nodes of degree at most four. To achieve this, a Steiner node of degree more than four is repeatedly replaced by a so-called *gadget*, which retains the connectivity

and planarity. The gadget and its properties are well known from the literature [MP93, NS08]. The new Steiner nodes have a degree of at most four. For example let v be a Steiner node of degree seven in G. The node will be replaced by a gadget as shown in figure 3.4.

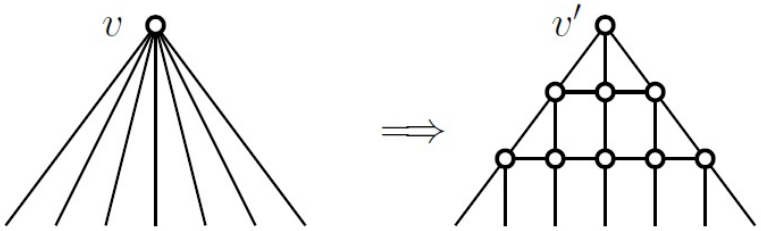


Figure 3.4: Gadget for high degree nodes [Aaz08]

Let v be a Steiner node of degree $d > 4$ in G. The gadget has $\lceil \frac{d}{2} \rceil - 1$ rows including the row with the previously given Steiner node, which is now called v'. Through this process the graph now has one less Steiner node of degree more than four. The set of terminal nodes and their degree stay the same. Any set of edge-disjoint paths using edges which belongs to v can be rerouted via the gadget (Figure 3.5).The set of terminals R is k-connected in the obtained graph, where every Steiner node is of degree less or equal four.

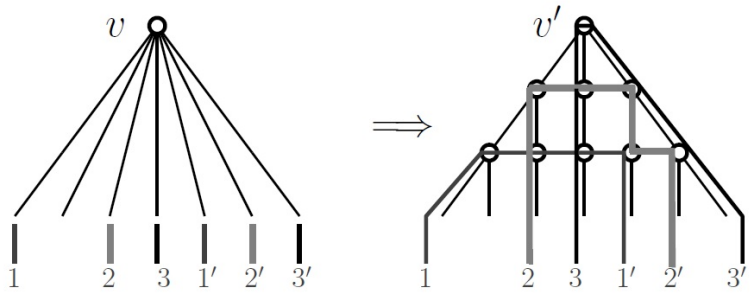


Figure 3.5: Routing paths via the gadget [Aaz08]

Given a pairing of edges, used in the paths, going through v, one of the extreme pairs will be routed first. An extreme pair is the pair, which uses the left most edge or the right most edge. This pair uses the first horizontal row of the gadget. The next pair is routed along the vertical edges to the next horizontal row. Some of the paths may be need to be *shifted*, like the path labeled $(2, 2')$, which is shifted on the first row to the left (Figure 3.5).

An edge that connects two terminals can be subdivided by introducing a Steiner node. Aazami assumes that G' does not contain such an edge connecting two terminals. By applying the element-disjoint Steiner tree packing algorithm (Section 3.4.3) to G', there are $\lfloor \frac{k}{4} \rfloor - 1$ element-disjoint Steiner trees in G'. These Steiner trees are obviously edge-disjoint. Then it can be said that G has at least $\lfloor \frac{k}{4} \rfloor - 1$ edge-disjoint Steiner trees.

3.4.2 Vertex-Disjoint Steiner Tree Packing

In the vertex-disjoint case, each node of the graph G is only allowed to occur in one of the Steiner trees. The vertex-disjoint Steiner tree packing problem in planar graphs is known to be *NP*-complete. Even when restricted to paths, which means that all nets have only two terminals, the problem remains *NP*-complete. Robertson and Seymour [RS90] showed, that the vertex-disjoint paths packing problem is solvable in polynomial time if the number of paths k is fixed. A linear-time algorithm for planar graphs, which goodness strongly depends on the number of paths, has been introduced by Reed, Robertson, Schrijver and Seymour[RRSS93].

There are polynomial time algorithms for the vertex-disjoint Steiner tree packing problem in planar graphs, where the terminals lie only on one or two borders of the graph. The algorithms, which are presented in the following, where the terminals lie only on one border of the graph may be seen as a key idea for simple linear-time algorithms for similar problems.

(Vertex-Disjoint One-Face Steiner Tree Packing Problem)

Given: A planar graph $G = (V, E)$, $|V| = n$, and pairwise vertex-disjoint sets $N_1, ..., N_k \subseteq V$. The graph G is embedded in the plane, such that all terminals lie on the boundary of the outer face of G.

Problem: Find, for each $i = 1, ..., k$, a Steiner tree T_i for N_i such that $T_i, ..., T_k$ are pairwise vertex-disjoint.

It should be noted, that in this problem statement a feasible not an optimal solution is searched. The algorithm, given by Liao and Sarrafzadeh [LS91], has two steps. In the first step the topological solvability is tested and in the second step the layout of the nets is determined. It is important that enough edge capacity is available and that the nets have a nested structure, which means that the nets are not allowed to intersect each other. The topological solution is a collection of Steiner trees for the nets that can be drawn disjointly in the plane outside the outer face.

The topological solvability can be decided by the simple *Stack Algorithm*. It checks whether the nets are nested. In this algorithm the terminals are scanned in anti-clockwise direction along the outer face boundary, beginning with some arbitrary terminals. Every new visited terminal is pushed onto the stack. If the pushed terminal is the last non-visited terminal of the corresponding net, it is tested if all terminals of the net lie on top of the stack. If this is not the case, the problem is not topologically solvable. Otherwise, all terminals of the net are popped. When all the terminals of all nets are visited, and there was no conflict before, the instance is topologically solvable only if the stack is empty. For net N_i the first terminal is denoted by s_i and the last terminal by t_i.

Stack Algorithm

Input:
A net set $\mathcal{N} = \{N_i | i = 1, ..., k\}$ and a planar graph G, where s_i and t_i are on the outer face.
STACK := $\varnothing$

Output:
Topological solvable or unsolvable.

begin
arbitrarily choose a vertex v as the starting point;
the first terminal s_i is the one closest to v in anti-clockwise direction of the boundary of G;
set $j = i$ and $a = |N_j| - 1$;
walk anti-clockwise direction along outer face boundary;
if visited vertex $v \in N_j$;
then $a = a - 1$;
if $a = 0$, such that v is a last terminal t_i;
then POP until s_i is popped;
else if $a \neq 0$;
then PUSH v on the stack;
else if $v \in N_k$ for $k \neq j$ and $a \neq 0$;
then return topological unsolvable;
else if $v \in N_k$ for $k \neq j$;
then $j = k$ and $a = |N_j| - 1$;
end;
return topologically solvable;

The Stack Algorithm can be implemented to run in linear time. The one-face layout algorithm, which tries to get a solution, is now based on the Stack Algorithm. This algorithm is explained in the following. In order to get a correct spanning of the nets, they are considered in the order, in which they have been deleted from the stack. After a net is routed, the boundary is corrected by deleting all used edges and vertices, and all edges incident to them.

(One-face Layout Algorithm)

This algorithm can be interpreted as a right-first search or a depth-first search, where in each search step the edges are searched from right to left. A backtrack and remove step consists of a backtrack step, where in addition the searched edge is deleted from the graph. For technical reasons all s_i are determined to have degree one in G. In the following the *One-Face Layout Algorithm* is formulated:

Algorithm

for $i := 1$ **to** k **do**
  let p_i initially consist of the unique edge incident to s_i;
  $v :=$ the unique vertex adjacent to s_i;
  while not all terminals of N_i are visited and $v \neq s_i$ **do**
    if at least one edge incident to v is not yet searched **then**
      let $\{v, w\}$ be the next edge after the leading edge of p_i w.r.t. v;
      if w is just occupied by some tree different from N_i **then**
        perform a backtrack and remove step;
      else add $\{v, w\}$ to p_i;
      $v := w$;
    else perform a backtrack and remove step;
    $v :=$ the leading vertex of p_i;
  if $v = s_i$ **then** stop; **return** unsolvable;
return $(p_1, ..., p_k)$;

The running time of this algorithm is also $\mathcal{O}(n)$.

3.4.3 Element-Disjoint Steiner Tree Packing

This approximation algorithm for packing element-disjoint Steiner trees in planar graphs was introduced in [Aaz08]. An element describes either an edge or a Steiner node. The goal is to find a maximum cardinality set of element-disjoint Steiner trees, such that each tree contains every terminal node. The method consists of two steps. In the first step the given graph G will be transformed into a bipartite graph. In the second step the bipartite graph is considered as a hypergraph and apply a method of Frank et al. [FKK03] to decompose the set of hyperedges ε into a number of disjoint sets $\varepsilon_1, \varepsilon_2, \varepsilon_3, ...$, where each set is a Steiner tree of the bipartite graph. Each of these bipartite Steiner trees will be transformed back to a Steiner tree of the original graph.

A graph is k-element connected if it remains connected whenever at most k edges or Steiner nodes are removed. The following statement is the main result of this algorithm:

Let $G = (V, E)$ be an undirected planar graph, let $R \subseteq V$ be the set of terminals, and assume that R is k-element connected. Then there are at least $\lfloor \frac{k}{2} \rfloor - 1$ element-disjoint Steiner trees in G. Moreover, there is an algorithm with a running time of $\mathcal{O}(|V|^{4.5})$ that finds at least $\lfloor \frac{k}{2} \rfloor - 1$ element-disjoint Steiner trees in G.

The bipartite Steiner tree packing problem is defined to be a subproblem of the element-disjoint Steiner tree packing problem, such that the graph is bipartite, all terminals are in one part of the bipartition and all Steiner nodes are in the other part. Given such an instance of the bipartite Steiner tree packing problem $G = (R, U; E)$, where R is the set of terminal nodes and U is the set of Steiner nodes. A hypergraph $\mathcal{H} = (V, \varepsilon)$ (Section 2.5) is associated to G as follows. The node set of $\mathcal{H}$ is given by the set of terminal nodes R of G and for each Steiner node $u \in U$ there is a hyperedge Z_u, that contains the set of neighbors of u in G. An instance of the bipartite Steiner packing problem with any hypergraph can be introduced by $G_{\mathcal{H}}$.

In the following an algorithm is described, where a planar graph G is given with the goal to find element-disjoint Steiner trees:

Algorithm:

1. In the first step, reduce the given graph $G = (V, E)$ to an instance G' of the bipartite Steiner tree packing problem; note that G' is obtained from G by removing or contracting white edges in G.

2. In the second step, using results of Frank et al. [FKK03], the associated hypergraph $\mathcal{H}$ of G' is decomposed by using Edmonds' matroid partition algorithm [Edm65], into the maximum number of partition-connected subhypergraphs. The independence test in this algorithm is to check if the given hypergraph is a hyperforest.

3. Each partition-connected subhypergraph corresponds to an Steiner tree in G'. By uncontracting the edges in G that were contracted in the first step of the algorithm, at least $\left|\frac{k}{2}\right| - 1$ element-disjoint steiner trees in G will be obtained.

Chapter 4

VLSI-Design

The motivation for the study of Steiner tree packing problem comes from the field of the design of electronic circuit, which is also called Very Large Scale Integration (shorter: VLSI). An important subproblem that occurs in the design of electronic circuits is the routing problem. In this chapter, the routing problem is considered in more detail. The problems which arise are identified and some solution approaches are given [JLPP02, GMW96a, GMW97, KH12].

4.1 Routing Problem in VLSI-Design

The VLSI-Design can be divided into two phases. A circuit has to fulfill a given task. The task is a complex logical function, which is comprised of many elementary logical operations. The *logical design* specify, which of the logical functional units may be used and defines the connection, which have to be realized between the units. These units specifying a logical function called *cells*. The connections, which have to be realized are called *networks*. The list of cells, together with the list of networks, forms the input for the second phase, the *physical design*. The task here is to realize the logical design physically, this means that the cells have to be arranged on a given surface and to realize the network, which is connected by electrical lines. The task is made more difficult, because depending on the technology specific design rules (for example: minimum distance of certain cells or networks) must be observed and an objective function, where the resulting surface must be minimized. Generally this problem can be divided into two subproblems,

which are solved one after the other. The first is the *placement problem*, which deals with the arrangement of the cells and the second is the *routing problem*, which deals with the realization of the network. In this thesis only the routing problem is from intrest.

As mentioned above, the routing problem is to connect the cells on the routing area subject to certain technical side constraints. The objective usually is to minimize the overall routing length. A net is called *routed* if its terminals are connected. A net where k is the number of terminals, is defined as a *k-terminal net*. If $k > 2$, the term *multiterminal net* is often used.

For the routing there are usually several levels available, the so-called *layers*. If a net changes a layer, a hole must be drilled. Such a hole is called a *via*. The routing of a net is usually on horizontal and vertical lines, the so-called *tracks* to which the contacts of the nets must be assigned. If the routing of the nets is not on such tracks, this can also be called a *grid-free routing*.

4.2 Decomposition of the Routing Problem

In practice, the routing problem is usually decomposed, so that large scales, which can occur at this problem, can be treated. In a first step, the networks homotopy is determined. It is defined how the cells have to be connected. This step is also known as the *global routing*. The second step is the *detailed routing*. Here the nets are assigned to the layers and tracks according to the homotopy specified in the global routing step. This decoposition scheme gives rise to many variants of the routing problem.

4.2.1 Global Routing

For modelling the global routing problem, the routing area is subdivided into subareas and these are represented by nodes in a graph. There are many variants to make such a subdivision, which among other things depends on the underlying technology and the methods used. An example of how such an routing area may be divided is illustrated in figure 4.1.

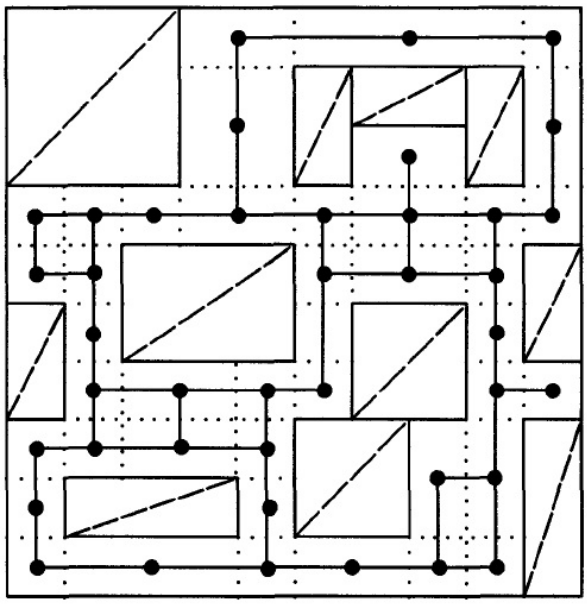


Figure 4.1: Dividing a routing area in subareas

The enclosing rectangle represents the given area. The rectangular units, which contain a diagonal represent the cells. The routing area is subdivided into rectangular subareas. This subdivision of the routing area is represented by a graph as follows. For each subarea, a node is defined and an edge between two nodes is introduced, if their corresponding subareas are adjacent. Let $G = (V, E)$ be the graph that results from this construction, where each edge e receives a capacity $c_e \in \mathbb{N}$, which indicats how many nets can use this connection. In addition, each edge gets a weight $w_e \in \mathbb{R}_+$, indicating the distance between the centers of the corresponding subareas. The task now is to find a routing for each net in G such that the capacity conditions are preserved and the total routing length is minimized. The formulated task here corresponds to an instance of the weighted Steiner tree packing problem.

4.2.2 Detailed Routing

After having solved the global routing problem, the detailed routing problem must be solved. This means, that every subarea, which are represented by nodes and are connected by edges, must be routed in detail. The homotopy, which was determined at the global routing, must be considered. There are a lot of different detailed routing models which are studied in the literature. In most cases, these problems can be formulated in a grid graph. This thesis will be restricted to problems in such grid graphs.

The detailed routing problem can be classified according to two independent criteria. In the following the two (A, B) criteria are explained and various models are introduced.

A: Structure of the routing area
One criterion distinguishes the detailed routing problems in the construction of the routing areas. In the following different resulting models are presented:

(Channel routing)
Given a complete, rectangular grid graph. The terminals of the nets may be located on two opposing borders of the graph (Figure 4.2). The size of the routing area is not fixed in advance.

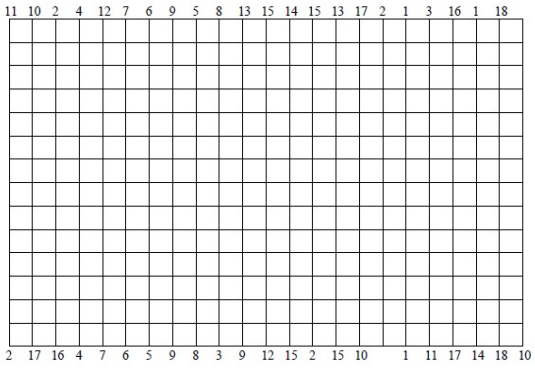


Figure 4.2: Channel routing area

(Switchbox routing)
Given a complete, rectangular grid graph. The terminals of the nets may be located on all borders of the graph (Figure 4.3). The size of the routing area is fixed here.

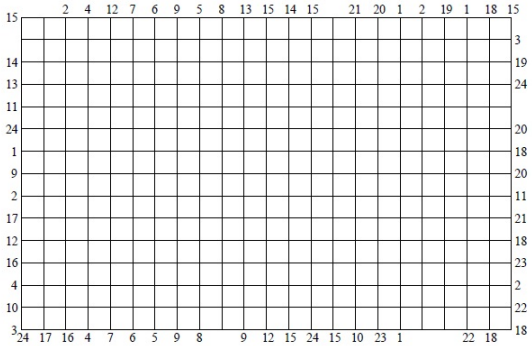


Figure 4.3: Switchbox routing area

(General routing)
Given an arbitrary grid graph. Here the graph may contain holes or have a non-rectangular structure. The size of the routing area is fixed and the terminals may be located arbitrarily at the holes or borders (Figure 4.4). It should be noted that in contrast to the above two models, the homotopy of nets is not trivial and therefore must be taken into account.

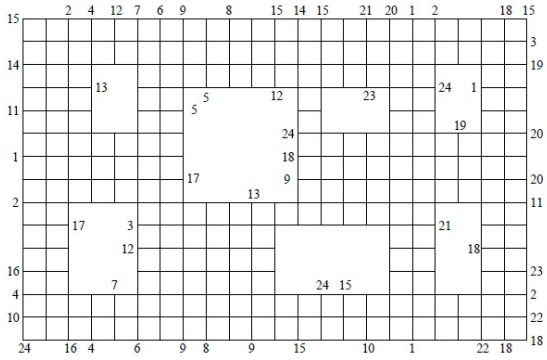


Figure 4.4: General routing area

B: Layer structures
The other criterion distinguishes the detailed routing problems by extent to which the layers are taken into account when the connections of the nets are assigned to the tracks. In the following are presented different resulting models for this criterion:

(Multiple layer model)
A k-dimensional grid graph can be obtained by stacking k copies of a grid graph on top of each other and connecting the corresponding nodes by perpendicular lines. The nets have to be routed in a node-disjoint way. This model certainly comes very close to the technology requirements, but has the disadvantage that, in general, the resulting graphs are very large. The number k has to be chosen minimal.

(Manhattan model)
Given a planar grid graph. The nets have to be routed in a node-disjoint way, with the additional restriction that nets that meet at some node are not allowed to bend at this node. The so-called knock-knees are not allowed (Figure 4.5).

(Knock-knee model)
Given a planar grid graph. The task is to find an edge-disjoint routing of the nets. Here the restriction of the Manhattan model is omitted, which means that knock-knees are allowed. It often arise shorter connections, than in the previous model, but the main disadvantage is that the assignement to layers is neglected. Brady and Brown have designed an algorithm, that guarantees, that any solution in this model can be routed on four layers [BB84]. It was shown, that it is *NP*-complete to decide if a realization on three layers is possible.

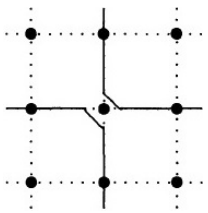


Figure 4.5: Knock-knee model

These two types of categorization (A and B) with its models can be combined arbitrarily. For example, the switchbox routing problem in the knock-knee model, which will be of interest in the approach presented in chapter 6. In graph-theoretic terms this combination can be formulated as a problem of finding edge-disjoint Steiner trees in a complete grid graph, where all terminals may be located on all borders of the graph. Additionally is the fact that depending on the model different objective functions can be optimized. Possible objective functions are, minimizing of the routing area, minimizing of the routing length or minimizing the number of vias. In the case of switchbox routing, minimizing of the routing length is typically the objective, because here the routing area is fixed. The objective for the channel routing problem is usually the minimization of the routing area and for the general routing problem the minimization of the number of vias can be used as the objective function.

4.3 Related Problems

As seen in this chapter, a large number of subproblems of the routing problem can be derived. For all these models different approaches are studied and discussed in the literature. In this section different approaches for the switchbox routing problems, which are formulated as an instance of the Steiner trees packing problem, are given.

4.3.1 Switchbox routing

(Exact Method)

In this context an exact method is understood as an algorithm that always finds a solution, if there exists a solution or it decides in finite time that there is no solution. Until now exact methods for the switchbox routing problem are known for the case, where only two terminals have to be routed in the knock-knee model. Because in this case the problem, to route N nets, is equivalent to the edge-disjoint packing problem.

Given a planar graph $G = (V, E)$ and a net list $\mathcal{N} = \{T_1, T_2, ..., T_N\}$ with $T_K = \{s_k, t_k\}$ for $k = 1, 2, ..., N$. The outer face of G is denoted by the edge set O_G. Robertson and Seymour [RS90] denoted $f(X) = \delta_G(X)$ as an induced cut $X \subseteq V$, where $\delta_G(X)$ is the set of edges in G with one end node in X and another in $V \backslash X$ (see Section 2.1.7). In the following an algorithm is described:

Algorithm

Input:
A planar graph $G = (V, E)$ and a net list $\mathcal{N}$ of nets $\{s_k, t_k\}$ with $k = 1, .., N$.
Output:
N edge-disjoint paths $P_1, P_2, ..., P_N$, such that s_k and t_k are connected through P_k for $k = 1, 2, ..., N$, else there exists no solution.

(1) if G is not 2-vertex-connected, decompose G in components, which are 2-vertex-connected and use the following steps on every component
(2) let $e \in O_G$ be an edge, located in an induced cut of X, which is minimal regarding f
(3) if $f(X) < 0$, then STOP (there exists no solution)
(4) number $V(O_G)$ cyclically, such that $V(O_G) = \{v_0, v_1, ..., v_k = v_0\}$ with $e = v_0 v_1$
(5) if $f(X) = 0$, then
(6) choose a vertex minimal cut X' with $v_i \in X'$ and $v_0 \notin X'$;
(7) choose $[v_i, v_j] \in \delta_H(X')$ with $v_j \notin X'$, j maximum;
(8) if $f(X) > 0$, then
(9) choose $[v_i, v_j] \in F$ arbitrarily;
(10) set $G := G \backslash e$ and $\mathcal{N} := \mathcal{N} \backslash \{v_i, v_j\} \cup \{\{v_j, v_0\}, \{v_i, v_1\}\}$
(11) if $E \neq \varnothing$, go to (1)
(12) if $s_k \neq t_k$ for $\{s_k, t_k\} \in \mathcal{N}$, then STOP (there exists no solution)
(13) identify the paths $P_1, P_2, ..., P_N$ according to the decomposition in (10)
(14) STOP.

The difficult part of this algorithm is to show, that the choice of the cut X in step (6) and the choice of the nets in step (7) guarantee, that $G \backslash e$ and $\mathcal{N} := \mathcal{N} \backslash \{v_i, v_j\} \cup \{\{v_j, v_0\}, \{v_i, v_1\}\}$ are solvable, if G and $\mathcal{N}$ are solvable too. The reduction of the problem in step (10) ensures, that $\mathcal{N} \subseteq V(O_G)$. Finally, it should be noted for determining the ways in step (13), that the way P with the terminals $\{v_i, v_j\}$ in step (10) is made up of the ways P_1 with $\{v_j, v_0\}$, the way P_2 with $\{v_i, v_1\}$ and the edge e.

(Heuristics)

A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution, when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In this section some heuristic methods are described.

A greedy switch-box router

This approach introduced by Luk [Luk85] uses the Manhattan model for the switchbox routing problem. Luk tries to extend a channel routing algorithm to a switchbox routing problem. The idea is to pass through the columns of the routing area from left to right. In each column (iteration) the nets are assigned to the horizontal tracks. The nets, that already use horizontal tracks will be routed by taking into account the terminals, which have to be connected. Nets, which have a terminal in the current column are assigned to free horizontal tracks. If there are not enough horizontal tracks, then additional tracks will be introduced. Also, additional columns are inserted at the end, if the rightmost terminals can not be connected. Every decision of the algorithm is based on very heuristic arguments. This is especially true for the selection of the direction (from left to right, top to bottom, or in each case vice versa) in which the grid graph is traversed.

BEAVER: A computational geometry-based tool for switchbox routing

This approach, called BEAVER [Coh88], uses the multiple layer model. In this case just two layers. It proceeds in four phases. In the first phase only terminals are considered, which can be associated with exactly one change of direction, a so-called *bend*. In the second step, linear connections with two changes of direction are also allowed to route two components of a net. Then, an attempt is made to route the remaining components by a Maze router. The Maze router is a connection routing method, where the goal is to find a path that go from point A to point B. The last phase assigns connections, whose location is not yet clearly established. It tries to minimize the number of vias. This approach with its four phases is very time-intensive. But each of the four phases expires sequentially, which means that there is a defined order in which the nets are processed.

Chapter 5

Integer Programming Models

In this chapter different, mathematical formulations for the Steiner tree packing problem are given. A survey of these integer programming models can be found in [Cho94]. In the following the same notations as introduced in section 3.2 is used.

5.1 The Undirected Cut Formulation

This formulation is used by Grötschel et al. in [GMW97]. A similar formulation as presented here, is given by Lengauer and Lügering in [Len90] and [LL93].

Given a weighted instance of the Steiner tree packing problem $(G, \mathcal{N}, c, w)$. In addition binary variables χ_e^k are introduced, for all $k = 1, ..., N$ and $e \in E$.

$$\chi_e^k = \begin{cases} 1 & \text{if edge } e \text{ is in the Steiner tree spanning net } k, \\ 0 & \text{otherwise.} \end{cases}$$

Each partitition $(W, V \setminus W)$ of the nodes V of the graph G defines a cut. It is called a *Steiner cut* for net k if $|W \cap T_k| \geq 1$ and $|T_k \setminus W| \geq 1$. Steiner cuts are based on the idea that a Steiner tree contains a path from the terminal $t \in T_k$ to every other terminal $T_k \setminus \{t\}$. Thus every cut separating the ter-

minal t from any other terminal $T_k \backslash \{t\}$ must contain at least one edge from the Steiner tree. Let $\delta(W)$ be the set of edges in G with one end node in W and another in $V \backslash W$. For a net k with $k = 1, ..., N$ and each associated Steiner cut $(W, V \backslash W)$, the *Steiner cut inequality* is defined. In generel the Steiner cut inequalities ensure that for each terminal set T_k, all Steiner cuts are covered with at least one edge.

In order that the capacity is large enough to satisfy the requirements of all connections, which have to be housed, the *capacity inequalities* are introduced. It can also be said that by using capacities on the edges the formulation can be extended to model an arbitrary number of layers.

In the following an integer programming formulation to model an edge-disjoint routing for the weighted Steiner tree problem is given:

$$\min \sum_{k=1}^{N} \sum_{e \in E} w_e \chi_e^k$$

$$\text{(i)} \sum_{e \in \delta(W)} \chi_e^k \geq 1, \text{ for all } W \subset V,\ W \cap T_k \neq \varnothing,\ T_k \backslash W \neq \varnothing.$$

$$\text{(ii)} \sum_{k=1}^{N} \chi_e^k \leq c_e, \text{ for all } e \in E.$$

$$\text{(iii)} \ \chi_e^k \in \{0, 1\}, \text{ for all } e \in E,\ k = 1, ..., N.$$

In the above mentioned integer programming model the inequalities (i) are called *Steiner cut inequalities* and inequalities (ii) are called *capacity inequalities*. The model can be further strengthened with several valid inequalities, which are described by Grötschel et al. in [GMW96a, GMW97, GMW96b].

This formulation contains only $|E| + |E| \times N$ variables, but exponential many inequalities (order of $N \times 2^{|V|}$ constraints of type (i)).

5.2 The Directed Cut Formulation

This formulation was introduced by Wong in [Won84]. A similar formulation as presented here, is given by Bienstock and Bley in [BB00] or by Althaus, Polzin and Daneshmand in [APD03].

Given a weighted instance of the Steiner tree packing problem $(G, \mathcal{N}, c, w)$, the formulation can be restated on a corresponding directed graph as follows: Given $G = (V, E)$ with edge set E, construct the directed graph $D = (V, A)$ with arc set A, where arcs $a = (s, t)$ and $a' = (t, s)$ are in A if and only if edge $e = st \in E$. For each $k \in \mathbb{N}$ one vertex $r_k \in T_k$ will be chosen as root. An arborescence is a directed, rooted tree in which all edges point away from the root, such that an arborescence rooted at r_k is said to be a *Steiner arborescence*, if it spans each node in T_k. A routing on D consists of a set of Steiner arborescences with one for each net. In the following the variable y_a^k for each net k and arc $a \in A$ are introduced:

$$y_a^k = \begin{cases} 1 & \text{if arc } a \text{ is in the Steiner arborescence spanning net } k, \\ 0 & \text{otherwise.} \end{cases}$$

In the following an integer programming formulation of the directed version is given:

$$\min \sum_{a \in A} \sum_{k=1}^{N} w_e y_a^k$$

$$\text{(i)} \sum_{a \in \delta(W)} y_a^k \geq 1, \text{ for all } W \subset V,\ W \cap T_k \neq \varnothing,\ T_k \backslash W \neq \varnothing.$$

$$\text{(ii)} \sum_{k=1}^{N} (y_a^k + y_{a'}^k) \leq c_e, \text{ for all } a \in A.$$

$$\text{(iii)} \ y_a^k \in \{0, 1\}, \text{ for } k = 1, ..., N.$$

Define a cut $(W, V \backslash W)$ to be a *directed Steiner cut* if $r_k \in W$ with $|W \cap T_k| \geq 1$ and $|T_k \backslash W| \geq 1$. Define $\delta(W)$ to be the set of arcs directed from W to $V \backslash W$ with one end in W and the other in $V \backslash W$. For each net k

and directed Steiner cut $(W, V \setminus W)$, the directed Steiner cut inequality (i) is obtained.

Consider an undirected edge $e = st \in E$ and let $a = (s, t)$ and $a' = (t, s)$ be the corresponding arcs in A. The capacity is consumed, if either a or a' are contained in the directed routing. The capacities inequalites for this model are given in inequalities (ii) [Cho94].

5.3 The Explicit Formulation

This formulation has been considered by Lengauer and Lügering [LL93]. A similar formulation has also been considered by Raghavan and Thompson [RT87].

Here the notation is a bit different as in the other formulations. The idea is that for each set of edges, which is a Steiner tree with respect to a terminal set a variable is introduced. Let $S_i = \{T_{ij} | j = 1, ..., n_i\}$ be the set of all possible Steiner trees in the graph $G = (V, E)$, where $T_{i1}, T_{i2}, ..., T_{in_i}$ is an enumeration of the spanning tree of the ith terminal set. n_i may be exponential in $|V|$. In this formulation a variable z_{ij} for each Steiner tree T_{ij} for net i is explicity defined. In the following the variable z_{ij} is introduced:

$$z_{ij} = \begin{cases} 1 & \text{if Steiner tree } T_{ij} \text{ is chosen to span net } i, \\ 0 & \text{otherwise.} \end{cases}$$

The total number of variables, where k is the number of terminal sets, is given by:

$$\sum_{i=1}^{k} n_i$$

The weight l_{ij} of a Steiner tree T_{ij} is given by:

$$l_{ij} = \sum_{e \in T_{ij}} w_e$$

In the following an integer programming formulation of the explicit version is given:

$$\min \sum_{i=1}^{k} \sum_{j=1}^{n_i} l_{ij} z_{ij}$$

$$\text{(i)} \sum_{j=1}^{n_i} z_{ij} = 1, \text{ for } i = 1, ..., k.$$

$$\text{(ii)} \sum_{i=1}^{k} \sum_{j: e \in T_{ij}} z_{ij} \leq c_e, \text{ for all } e \in E.$$

$$\text{(iii)} \ z_{ij} \in \{0, 1\}, \text{ for } i = 1, ..., k, \ j = 1, ..., n_i.$$

The constraint (i) ensures, that exactly one Steiner tree is chosen for each net i, where all nets are edge-disjoint to each other. In constraint (ii) the requirement for the capacities is given.

A disadvantage of this formulation is certainly the number of variables. The numbers n_i are generally exponential in the size of the input data. Exceptions are problem instances whose underlying graph is very small.

5.4 The Multicommodity Flow Formulation

The multicommodity flow formulation as proposed by Wong [Won84] has the advantage that it has only a polynomial number of variables and constraints. A similar formulation is given by Koch in [KH12]. The multicommodity flow formulation can be derived from the directed formulation.

Given an undirected graph $G = (V, E)$, the directed graph $D = (V, A)$ is formed as in section 5.2. For each net k, one of the nodes $r_k \in T_k$ is declared as the root. For each net k and terminals $s \in T_k \backslash \{r_k\}$, a commodity (k, s) is defined. Given an arc $a = (f, h)$, let χ_{fh}^{ks} represent the flow of commodity (k, s) on arc a. In order to ensure a Steiner tree for each net $k \in \{1, ..., N\}$, a flow of one unit of commodity (k, s) from r_k to s for all $s \in T_k \backslash \{r_k\}$ have to be ensured. This is done by using the following flow constraints:

$$\sum_{f \in V} (\chi_{fh}^{ks} - \chi_{hf}^{ks}) = \begin{cases} -1 & \text{if } h = r_k, \\ 1 & \text{if } h = s, \\ 0 & \text{if } h \neq r_k, s. \end{cases}$$

If in the above mentioned constraints $(f, h) \notin A$, the variable χ_{fh}^{ks} is simply ignored. If a multicommodity flow is found, which satisfies these constraints for $s \in T_k \backslash \{r_k\}$, the arcs with positive flow contain a Steiner arborescence spanning net k. In order to capture this, integer variables $y_{k,a}$ are defined:

$$y_{k,a} = \begin{cases} 1 & \text{if there is a positive flow of any of the commodities } (k, s) \\ & \text{for } s \in T_k \backslash \{r_k\} \text{ on arc } a, \\ 0 & \text{if the flow on arc } a = 0. \end{cases}$$

In the following an integer programming formulation of the multicommodity flow version is given:

$$\min \sum_{a \in A} \sum_{k=1}^{N} w_a y_{k,a}$$

(i) $\chi_{fh}^{ks} \leq y_{k,a}$, for $a = (f, h) \in A$, $k = 1, ..., N$, $s \in T_k \backslash \{r_k\}$.

(ii) $\sum_{k=1}^{N} (y_{k,a} + y_{k,a'}) \leq c_e$, for $a = (s, t)$, $a' = (t, s)$, $e = [s, t] \in E$,

(iii) $y_{k,a} \in \{0, 1\}$, for $k = 1, ..., N$, $a = (f, h) \in A$,

(iv) $\chi_{fh}^{ks} \in \{0, 1\}$, for $k = 1, ..., N$, $s \in T_k \backslash \{r_k\}$, $fh \in A$.

In constraints (ii) the capacities inequalities are given, let $a = (f, h)$ and $a' = (h, f)$ be the corresponding arcs in A. The capacity is consumed, if either a or a' are contained in the directed routing.

Let C be the total number of commodities created, such that the number of variables $|E| + |E| \times N + |A| \times C$ and the number of linear constraints $|V| \times C + |E| \times C + |E|$ can be given.

Results using this model to compute optimal routings can be found in Koch [KH12]. The advantage of the formulation is that it models all the layers simultaneously. On the other hand the size of the graph grows rapidly with the number of terminals.

Chapter 6

Approach

In this chapter, an approach, which is based on finding a solution with the help of integer linear programming is presented in more detail. In the approach the process is described, the mathematical formulations and heuristics are given.

In the following specific notations are still needed. It is considered a $N \times |E|$-dimensional vector space $\mathbb{R}^E \times ... \times \mathbb{R}^E$. For this vector space the symbol $\mathbb{R}^{\mathcal{N} \times E}$ is used. Each component of a vector $x \in \mathbb{R}^{\mathcal{N} \times E}$ is indexed by x_e^k with $k \in \{1, ..., N\}$ and $e \in E$. The vector $x^k \in \mathbb{R}^E$ with $k \in \{1, ..., N\}$ denotes a vector $(x_e^k)_{e \in E}$.

In the approach it is generally assumed that an instance of a weighted Steiner tree packing problem is given that defines a switchbox routing problem. Which means that a complete rectangular grid graph with edge capacities $c_e = 1$ with $e \in E$ and a netlist $\mathcal{N} = (T_1, ..., T_N)$ with $\mathcal{N} \subseteq V(O_G)$, where O_G is the outer face, is given. Furthermore, for each edge a weight w_e is given. In all examples of the literature, the weights are equal to 1. Most of the examples in the literature have a large grid graph and a large number of nets, which causes some problems in solving the problems computationally. This is because the number of variables for such examples is very large.

In order to solve problems of this magnitude, it is necessary to have a very fast and robust code for solving linear programs. Grötschel et al. [GMW96a] used in their implementation of the linear programs the simplex method, which was developed by Bixby [Bix91]. Nonetheless, solving the linear programs

appears quite difficult, because there are probably alternative optimum solutions and the linear programs occur highly degenerate. That means it exist a lot of different corners of the corresponding bases to a polyhedron. After a base exchange of the simplex method, it is often the case that the new basis is defined at the same area as before, such that no progress is achieved in the objective function.

Another problem occurs, when the actual linear program finds no optimal solution. In order to strengthen the linear program, it needs further inequalities, which can be determined by separation algorithms. Here the problem is, that not every separation algorithm is exact.

6.1 A Cutting Plane Algorithm

The cutting plane algorithm for the Steiner tree packing problem was described by Grötschel, Martin and Weismantel [GMW96a]. This algorithm is used to solve the switchbox routing problem in the knock-knee model by using separation algorithms and a LP-based primal heuristic.

6.1.1 Mathematical Formulations

In the following a mathematical formulation and its motivation is given. The switchbox routing problem in knock-knee model can be modeled as the Steiner tree packing problem with the additional restrictions:

- $G = (V, E)$ is a complete rectangular grid graph
- edge capacities $c_e = 1$ for all $e \in E$

By taking these restrictions into account the switchbox routing problem can finally be formulated as follows:

Instance:

A graph $G = (V, E)$ and a list of node sets $\mathcal{N} = (T_1, ..., T_N)$, $N \geq 1$, with $T_k \subseteq V$ for all $k = 1, ..., N$.

Problem:

Find edge sets $S_1, ..., S_N \subseteq E$ such that:

(i) S_k is a Steiner tree in G for T_k for all $k = 1, ..., N$,

(ii) $\sum_{k=1}^{N} |S_k \cap \{e\}| \leq 1$ for all $e \in E$,

(iii) $\sum_{k=1}^{N} |S_k|$ is minimal.

The integer linear program for the weighted Steiner tree problem [GMW97] (also see section 5.1) can be formulated as follows, wherein the constraint, that each variable must be either 0 or 1 is replaced by a weaker constraint, that each variable belong to the interval $[0, 1]$:

$$\min \sum_{k=1}^{N} \sum_{e \in E} w_e y_e^k$$

(i) $\sum_{e \in \delta(W)} y_e^k \geq 1$, for all $W \cup V,\ W \cap T_k \neq \varnothing,\ T_k \backslash W \neq \varnothing$.

(ii) $\sum_{k=1}^{N} y_e^k \leq 1$, for all $e \in E,\ k = 1, ..., N$.

(iii) $1 \geq y_e^k \geq 0$, for all $e \in E,\ k = 1, ..., N$.

Given that the capacity $c_e = 1$, this linear programming formulation can also be stated as a knock-knee one-layer model.

6.1.2 The Algorithm

In this section a cutting plane method for solving the weighted Steiner tree packing problem will be introduced. First the basic idea and the basic procedure of such a cutting plane method will be described. It will turn out, that the core of a cutting-plane algorithm is to solve the separation problems.

Procedure

The idea of the cutting plane algorithm is the following. For the start a small set of inequalities is taken, for example the trivial and the capacity inequalities. Let y be an optimal solution for the linear objective function, such that y describes a lower bound for the optimum value of the weighted Steiner tree packing problem. If y is feasible, then y is an optimal solution for the problem.

If y is not feasible, there exists a valid inequality, that is violated by y. At this point the separation problem will be used in order to find a valid inequality, that is violated by y. If an inequality can be determined, it will be added to the linear program and subsequently it will be solved again. Such a procedure of iteratively solving linear programs and adding violated constraints is generally called a *cutting plane algorithm*.

A cutting plane algorithm ends either with an optimum solution or at least with a lower bound of the Steiner tree packing problem. It can not be expected that the cutting plane method ends with an optimal solution, because not all known classes of valid inequalities have an exact algorithm. If the cutting plane algorithm ends with a solution y, which is not allowed, so the process can be embedded in an enumeration scheme.

The overall problem will be divided into two subproblems by fixing some variable to zero in one subproblem and in the other subproblem the same variable will be set to one. This procedure generates a binary tree, also called *branching tree*, where each subproblem is represented by a node. A key advantage of this approach is, that each cutting plane found for a subproblem is also valid for all other subproblems. The whole methode is commonly known as a *branch and cut algorithm*.

A branch and cut algorithm can only be used to solve practical problems efficiently if the generated branching tree is kept small. In order to achieve this, exact separation algorithms or at least good separation heuristics are necessary. Additionally, a good heuristic for finding a feasible solution is very important.

The separation algorithms and the associated correctness proofs are quite complicated. In order to not exceed the scope of the thesis, here the proofs and descriptions for the different separation algorithms are omitted. For a detailed discussion of this issue the reader is refered to [GMW96a] and [GMW93].

6.1.3 Primal Heuristic

The aim of this section is to explain the algorithm for determining a feasible solution for the Steiner tree packing problem developed by Grötschel, Martin and Weismantel. Based on this, a heuristic has been implemented in Matlab, which is presented in chapter 7.

It is known, that the Steiner tree packing problem is *NP*-complete even if the given instance defines a switchbox routing problem [Kv80]. Therefore, the restriction was to develop a heuristic. From the literature, no heuristic method for solving the switchbox routing problem in the knock-knee model are known. There are just algorithms, that find a solution if one exists, but only under the condition that all terminal sets have a cardinality of two (Section 4.3.1). The idea of the heuristic is to make use of the information given by the actual solution of the cutting plane algorithm.

The Algorithm

The developed heuristic is a sequential algorithm. Each terminal of a net is considered as an (isolated) component. Now iteratively two components of a net will be connected in a determined order. The procedure is not that a net will be fully routed, but only two components of the same net will be connected per iteration. Here the determined order plays an essential role

for the success of such a procedure. In this algorithm the order will be determined by the solution vector y of the actual linear program. More precisely, a function f depending on y is defined according to which two components are selected. The function will be explained later in more detail. An attempt is made to connect the two selected components on a shortest path. This path is in general not uniquely determined, since the underlying graph is a complete rectangular grid graph. Therefore, a number of criteria have been created in order to limit the choices. A detailed description of these details will be given later. If it is possible to connect the two selected components on a shortest path by taking the mentioned criteria into account, the two components will be connected, the graph and all relevant datastructures will be updated and the next pair of components, which have to be connected, will be selected. Else the function f is recalculated and a new order will be determined by taking into account the already connected components. This procedure will be iterated either until all nets are connected or no more connections are possible. In the following a detailed description of the algorithm is given with the solution vector y and the notation as introduced in section 2.2:

Input:

A complete rectangular $h \times b$ grid graph $G = (V, E)$ with edge capacities $c_e = 1$ and edge weights $w_e \in \mathbb{R}_+$, $e \in E$. Furthermore, a net list $\mathcal{N} = \{T_1, ..., T_N\}$ and a vector $y \in \mathcal{R}^{\mathcal{N} \times E}$, $y \geq 0$.

Output:

A feasible solution of the weighted Steiner tree packing problem $(G, \mathcal{N}, \mathbb{1}, w)$ or the message "No feasible solution found".

(1) Set $S_k := \varnothing$ for $k = 1, ..., N$.
(2) Determine the graph $\hat{G} = (V, \hat{E})$ with $\hat{E} := \{e \in E | c_e > 0\}$.
(3) Compute shortest paths for all pairs of nodes in $\hat{G}$.
(4) For $k = 1, ..., N$ perform the following steps (5) to (6):
(5) If $S_k = \varnothing$, then
determine s_k, $t_k \in T_k$ such that
$$f_{y^k}(s_k, t_k) = \min_{\substack{s,t \in T_k \\ s \neq t}} f_{y^k}(s, t);$$
set $T_k' := T_k \backslash \{t_k\}$.

(6) Else
determine $s_k \in T'_k$, $t_k \in V(S_k)$ such that
$f_{y^k}(s_k, t_k) = \min\limits_{\substack{s \in T'_k \\ t \in V(S_k)}} f_{y^k}(s, t)$

(7) As long as further connections are possible perform the following steps:

(8) Determine $k_0 \in \{1, ..., N\}$ with
$f_{y^{k_0}}(s_{k_0}, t_{k_0}) = min\{f_{y^k}(s_k, t_k) | k = 1, ..., N\}$.

(9) Try to connect s_{k_0} with t_{k_0} via a shortest path by taking the criteria into account.

(10) If the connection via a shortest path is possible, then
let W be the chosen path;
set $S_{k_0} := S_{k_0} \cup W$, $T'_{k_0} := T'_{k_0} \backslash \{s_{k_0}\}$ and $c_e := 0$ for all $e \in W$;
if $T'_{k_0} = \varnothing$, set $f_{y^{k_0}} := \infty$;
else determine another pair (s_{k_0}, t_{k_0}) similar to (6).

(11) Else go to (2);

(12) If all terminal sets are connected, return the feasible solution $(S_1, ..., S_N)$.

(13) Otherwise, print the message "No feasible solution found".

(14) STOP.

Let S_k be the edge set, that was already determined for connecting T_k, T'_k the set of not yet connected terminals, $\hat{G}$ the underlying graph and W the edge set of the chosen shortest path, where $W(s_k, t_k)$ is the shortest path from s_k to t_k in $\hat{G}$ with respect to w.

In the heuristic a distinction is made between two cases $S_k = \varnothing$ and $S_k \neq \varnothing$. The algorithm allows at most one component for each net, which has more than one terminal. If $S_k \neq \varnothing$, there is already a component containing more than one terminal. The task is then to connect the remaining terminals of T'_k to this component. If $S_k = \varnothing$, it has not yet determined which terminals are connected first.

$$f_{y^k} := |w(W(s,t)) - \sum_{e \in E(s,t)} w_e y_e^k|,$$

The above mentioned function f_{y^k} is used to find the two components which have to be connected. In general, the two components are searched, which have the smallest value resulting from the function. The first part of the function is the weighted sum of the shortest path from s to t. To explain the other part of the function a graph must be determined, which is the smallest rectangular grid graph containing both components, also called *minimal enclosing rectangle*. Inside the minimal enclosing rectangle the weighted sum will be computed, where only the edges with $y_e^k > 0$ are considered. The edges, which are inside the minimal enclosing rectangle are denoted by $E(s,t)$.

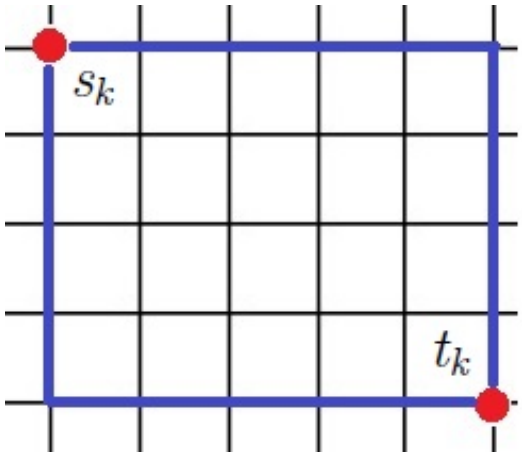


Figure 6.1: A minimal enclosing rectangle

In figure 6.1 an example of a minimal enclosing rectangle is shown. The mentioned rectangle is represented by blue lines. The red points represent the two components s_k and t_k.

Criteria for routing the shortest path

We consider now the execution of step (9) in the algorithm. Let k be the chosen net, s_k and t_k the nodes, which have to be connected via shortest path. First all neighbors v of s_k are determined, where $c_e = 1$. Among these, those nodes that lie on a shortest path from s_k to t_k are selected. Let η be the number of neighbor nodes, which may vary. If $\eta = 0$, then no connection on a shortest path is possible. If $\eta = 1$, then v is the unique node. Otherwise, if $\eta \geq 2$, there are a number of criteria, which help in selecting the node. Let L be the node list of candidates, such that the following criteria can be introduced:

(1) In the first criterion a way is searched, where the number of terminals is kept as small as possible. For this reason, the node in L is selected which is not a terminal.

(2)
$$y_{sv}^k = \max_{u \in L} y_{su}^k$$

Here the node is selected, where the corresponding value y_{sv}^k of the edge between s_k and the neighbor v in the solution vector y^k is as large as possible.

(3) In this criterion the not yet connected terminals of net k are taken into account. The neighbor v, whose distance to the not yet connected terminals is minimal, will be selected. The reason for this, is the attempt to keep the total length of the final Steiner tree for T_k as small as possible.

In figure 6.2 the two component s_k, t_k and a not yet used terminal are represented by red points. The green points represent the neighbors v_1 and v_2 of the component s_k. In this case v_2 is selected, because it is closer to the not yet used terminal.

(4) Choose $v \in L$ such that:

$$\sum_{k \neq k_0} y_{sv}^k - y_{sv}^{k_0} = \min_{u \in L} \sum_{k \neq k_0} y_{su}^k - y_{su}^{k_0}$$

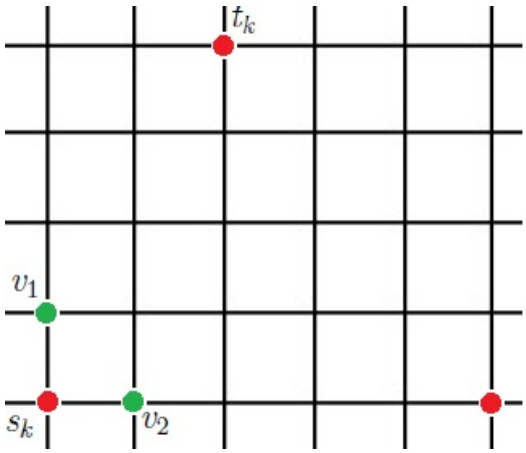


Figure 6.2: An example for criterion 3

With criterion (4), an attempt is made to avoid those edges that are likely to be preferred by another net.

(5) In the last criterion the node is selected, which gives less change in the direction.

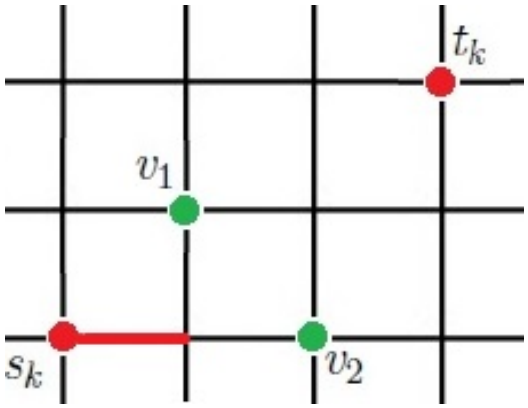


Figure 6.3: An example for criterion 5

In figure 6.3 an example for criterion (5) is sketched. In this case the neighbor v_2 will be selected, because in this case there would be no change of direction with respect to the last connection.

In the selection of the node v it is proceeded as follows. First, the first criterion is examined. If the minimum is uniquely determined, so let v be the corresponding node. If the minimum is not uniquely determined, all nodes that take the minimum in (1) will be used in criterion (2). If among these nodes the minimum in (2) is uniquely determined, then v is the corresponding node. Otherwise, the process continues according to the criteria (3) to (5). If in criterion (5) no node is uniquely determined, any node v is selected. After every iteration from (1) to (5) a node is determined, which is adjacent to s. Then the edge sv is added to the way W, set $s = v$ and iterate until $s = t_k$ or no neighbor node, which is located on a shortest path ($\eta = 0$), was found.

6.1.4 Results

In principle, the cutting plane algorithm can be used to find an optimal solution for the given switchbox routing problem or to determine that no solution exists. However, this may often not be guaranteed in an acceptable computing time. Therefore Grötschel et al. [GMW96a] has integrated a time barrier. If this time barrier is exceeded the process ends and gives out the until then best lower bound and the best feasible solution.

Grötschel et al. has given various results of different test examples. These switchbox examples have often been discussed in the literature. They include various difficulties, such as a very high density of the networks or a terminal intensive example, where all vertices of the outer face are terminals. The example *difficult switchbox* can be found in [BP83]. Examples *terminal intensive switchbox*, *dense switchbox* and *augmented dense switchbox* were introduced in [Luk85]. Finally, the examples *modified dense switchbox*, *pedagogical switchbox* and *more difficult switchbox* can be found in [Coh88].

In table 6.1 it is shown how large the test examples are and how large their corresponding nets are. In column 1 the name of the examples is listed. The

Name	height	width	nets	distribution of nets				
				2	3	4	5	6
difficult switchbox	15	23	24	15	3	4	1	1
more difficult switchbox	15	22	24	15	3	5		1
terminal intensive switchbox	16	23	24	8	7	5	4	
dense switchbox	17	15	19	3	11	5		
augmented dense switchbox	18	16	19	3	11	5		
modified dense switchbox	17	16	19	3	11	5		
pedagogical switchbox	16	15	22	14	4	4		

Table 6.1: Constellation of the Examples

height and the width of the corresponding grid graph can be found in column 2 and 3. The number of nets is listed in column 4. The last 5 columns provide information on the distribution of nets. For example column 6 indicates how many nets with 3 terminals are included in the problem.

Name	best solution	LP value	gap (%)	CPU-time (min)
difficult switchbox	464	464	0.0	1564:15
more difficult switchbox	452	452	0.0	983:23
terminal intensive switchbox	537	536	0.2	3755:44
dense switchbox	441	438	0.7	1017:43
augmented dense switchbox	469	467	0.4	4561:41
modified dense switchbox	452	452	0.0	387:03
pedagogical switchbox	331	331	0.0	251:58

Table 6.2: Results of the Examples

In table 6.2 the results of Grötschel et al. with their cutting plane algorithm are summarized. Column 2 gives the best feasible solution. The entries in column 3 are the objective function values of the linear program, when no further violated constraints are found. These values are lower bounds of the whole problem. In column 4 the percental derivation of the best solution from the lower bound is given. The last column reports on the running times, where the times are given in minutes.

The results are promising, such that for the given examples the best feasible solution deviates at most 0.7% from the optimal solution. Generally for problem instances arising in VLSI-Design there is only given a heuristic. The advantage of the cutting plane algorithm is, that the quality of the heuristically determined solution can be evaluated with the lower bound, which may include knowledge about the problem itself.

This approach has been shown to be quite promising, but there are still many open problems that need to be solved. In order to solve large scale problem instances as they occur in practice, the algorithm must be improved.

Chapter 7

Implementation

The explained heuristic in chapter 6.1.3 has been implemented in Matlab. Given that this is a LP-based heuristic, various changes have been made, such that at least the main idea was implemented. Instead of using the solution vector y, which is computed from a linear program, all values of y are set to 0. In general it can be said, that this simplification makes the heuristic independent of a linear program, such that it can be implemented within a Matlab environment.

7.1 The Algorithm

Each terminal of a net is considered as a component. Now iteratively always just two components of the same net will be connected. In which order the components are connected, is depending on a function f_k, which is explained later. The selected components have to be connected on a shortest path, where some criteria are important in order to limit the choices. If it is not possible to connect the two components, then another pair of components have to be connected. Is there no possible way to connect two components via shortest path, there should be a path via detour. If this is the case, there is no guarantee that the solution is optimal. The graph and all relevant datastructures will be updated and it will be continued with the next iteration. This precedure will be iterated either until all nets are connected or no more connections are possible. If it is not possible to connect two

components somehow, then the procedure will be interrupted, such that no solution was found. In the following a detailed description of the algorithm is given:

Input:

A complete rectangular *rows* $\times$ *columns* grid graph $G = (V, E)$ with edge capacities $c_e = 1$ and $e \in E$. Furthermore, a list of terminal sets $\mathcal{N} = \{T_1, ..., T_N\}$.

Output:

A feasible solution of the Steiner tree packing problem, given by the edge sets $(S_1, ..., S_N)$ or the message "No feasible solution found".

(1) Set $S_k := \varnothing$ and the not yet used terminals $T'_k = T_k$ for $k = 1, ..., N$, and the graph $\hat{G} = G$.
(2) For $k = 1, ..., N$ perform the following steps:
(3) If $S_k = \varnothing$, then
    determine $s_k,\ t_k \in T_k$ such that
    $f_k(s_k, t_k) = \min\limits_{\substack{s,t \in T_k \\ s \neq t}} f_k(s, t)$;
    set $T'_k := T_k \backslash \{s_k, t_k\}$.
(4) Else
    determine $s_k \in T'_k,\ t_k \in V(S_k)$ such that
    $f_k(s_k, t_k) = \min\limits_{\substack{s \in T'_k \\ t \in V(S_k)}} f_k(s, t)$
(5) Try to connect s_k with t_k via a shortest path by taking the criteria into account.
(6) If the connection via a shortest path is possible, then go to (9);
(7) Else determine another pair (s_k, t_k), go to (3);
(8) If the connection via a detour is possible, then
(9) Update datastructures:
    let W be the edge set of the chosen path;
    set $S_k := S_k \cup W,\ T'_k := T'_k \backslash \{s_k\}$;
    delete all $e \in W$ from the graph $\hat{G}$;
(10) Else print the message "No feasible solution found".
(11) If all terminal sets are connected, return the feasible solution $(S_1, ..., S_N)$.
(12) STOP.

$$f_k := |w(W(s_k, t_k))|$$

The function f_{y^k} presented in chapter 6 was changed to f_k and is also used to find the two components which have to be connected. The two components are searched, which have the smallest value resulting from the function. The function gives the weighted sum of the shortest path from s_k to t_k.

For routing the shortest path, those criteria were taken, which have no relation to the solution vector y. The criteria (1), (3) and (5) from chapter 6 were implemented.

7.2 Code

The implementation was programmed with Matlab, which is a high performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment, where problems and solutions are expressed in familiar mathematical notation.

Given that this is a graph-theoretic problem, it turned out to be usefull to use a toolbox. MATGRAPH is a toolbox provided by Scheinerman [Sch08] for working with simple graphs in Matlab. The goal of this toolbox was, to make interactive graph theory exploration simple and efficient.

In figure 7.1 a grid graph is shown, which was created with MATGRAPH. In order to understand the calculations in the following functions of the algorithm, it is intended to show how the vertices are arranged. The vertices are labeled with their corresponding number.

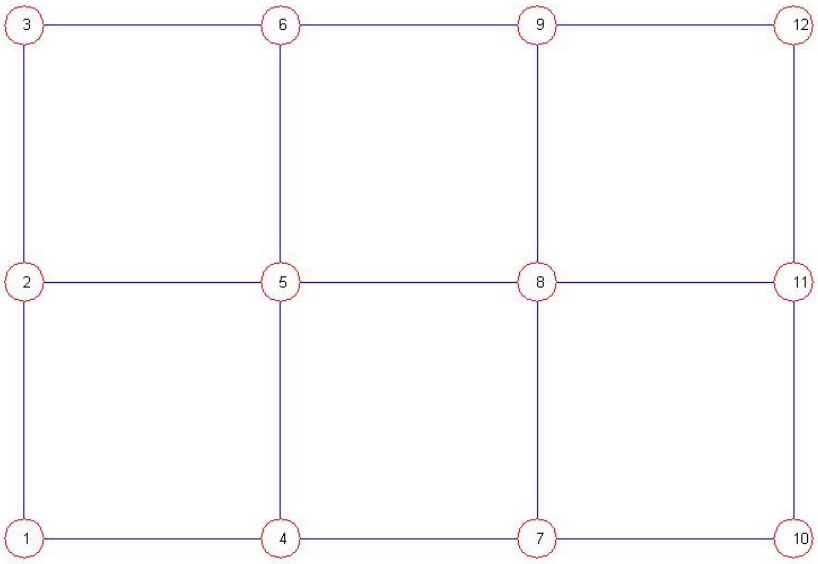


Figure 7.1: Structure of a grid graph

Similarly as in chapter 6.1.3 the *minimum enclosing rectangle* was used, but here in a differnt way. A shortest path between two vertices must run within their minimum enclosing rectangle. If there is no possible connection inside this rectangle, then there should be a connection via detour. Otherwise no path between the two vertices is available. In the following some functions are presented, where the minimal enclosing rectangle is abbreviated with *mer*.

The function *merVertices* determines all vertices of the minimal enclosing rectangle between the two vertices u and v. Additionally the height and the width are determined.

```
function [vert,h,w] = merVertices(rows,columns,u,v)

%determine height
h_u = mod(u-1,rows)+1;
h_v = mod(v-1,rows)+1;
h = abs(h_u-h_v)+1;
```

```
%determine width
if mod(u-1, rows)+1 < mod(v-1, rows)+1
    temp = u;
    u = v;
    v = temp;
end
w = (abs(v+h-1-u))/rows+1;

%vector which will be filled in the following with
%the searched vertices
vert = zeros(1, w*h);

%change the vertices, such that the vertice u is the
%left one
if v < u
    temp = v;
    v = u;
    u = temp;
end

%determine all vertices and store them
for i = 0:w-1
    for j = 0:h-1
        if mod(u-1, rows)+1 < mod(v-1, rows)+1
            vert(1+j+i*h) = u+i*rows+j;
        else
            vert(1+j+i*h) = u+i*rows-j;
        end
    end
end
end
```

A similar function is *merValue*, which gets the length of the shortest path, that is running inside the minimal enclosing rectangle for the two given vertices u and v. Basically the output here is the result of $height + width - 1$ of the minimal enclosing rectangle.

The function *merMin* finds the one vertex from each vertex set $T1$ and $T2$, which have to be connected. In general the two vertices are searched, where the length of the shortest path is minimal. This function plays an important role in the steps (3) and (4) of the algorithm.

```
function [a, b] = merMin(g, rows, columns, T1, T2)

%declaration
a = 0; %index of the first vertice in T1
b = 0; %index of the second vertice in T2
value = rows * columns; %biggest possible value

%save the the two vertices with the minimal
%shortest path
for i = 1:size(T1, 2)-1
    for j = i+1:size(T2, 2)
        if merValue(g, rows, columns, T1(i), T2(j))
           < value
           value = merValue(g, rows, columns, T1(i),
               T2(j));
           a = i;
           b = j;
        end
    end
end
end
```

In step (3) the vertice sets $T1$ and $T2$ are equal the terminal set T_k. However in step (4) $T1$ is the set of the not yet connected terminals T'_k and $T2$ is the set of vertices $V(S_k)$.

The function *merShortestPath* determines every shortest path from vertice u to v. In addition there are some criteria, such that the path which satisfy all the criteria, will be chosen. This function is used for step (5) of the algorithm. It is first checked, if the given vertices are valid, which means that the vertices exists in the graph G and that they are not equal. Then all neighbor

vertices (denoted by *near*) of u are determined. Here the above explained function *merVertices* is used to limit the area of the vertices. Before criterion one is used, it is checked whether one of the neighbors is leading to an impass.

In the first criterion the neighbor is selected, which is not a terminal. If the result is not unique, then continue with criterion two.

```
%criterion 1
temp = near;

%if a neighbor is contained in one of the terminal
%sets => set it equals 0
if size(near, 2) > 1
    for i = 1:size(T, 1)
        for j = 1:size(near, 2)
            if any(T{i} == near(1)) == 1
                near(1) = 0;
            end
            if any(T{i} == near(2)) == 1
                near(2) = 0;
            end
        end
    end
end

%remove neighbors (if needed)
if near(2) == 0
    near = near(near~=near(2));
end

if near(1) == 0
    near = near(near~=near(1));
end

%if there is one neighbor go on with the next
%vertice for the path, else go to criteria 2
if size(near, 2) == 1
```

```
    path = [path near(1)];
    continue;
else
    near = temp;
end
```

In the second criterion the not yet connected terminals of the same net are taken into account. The function *shortestPath* determines every shortest path from vertice u and v to all vertices in set T'.

```
%criterion 2
function direction = shortestPath(g, start, u, v, T)

%Output: 0 => the length of the paths are equal
%        1 => u is the chosen vertex
%        2 => v is the chosen vertex

length_u = size(edges(g), 1);
length_v = size(edges(g), 1);
direction = 0;
T = T(T~=start);

%determine all shortest paths to the not yet
%connected terminals and store the smallest one
for i=1:size(T, 2)
    path = find_path(g, u, T(i));
    if length_u > size(path, 2)
        length_u = size(path, 2);
    end
    path = find_path(g, v, T(i));
    if length_v > size(path, 2)
        length_v = size(path, 2);
    end
end

if length_u < length_v
    direction = 1;
```

```
end
if length_v < length_u
    direction = 2;
end
end
```

The vertex with the shortest path will be the chosen one. The output is 1, if u is the chosen vertex and 2 if v is the chosen vertex. Otherwise if these lengths are equal, then the output is 0, in this case continue with criterion 3.

In the last criterion the node is selected, which gives less change in the direction as explained in section 6.1.3. The vector *path* contains the until then determined shortest path.

```
%criterion 3
if size(path, 2) >= 2
    if  abs(path(end)-path(end-1)) == rows
        if path(end) < last
            path = [path near(2)];
        else
            path = [path near(1)];
        end
        continue;
    end
    if abs(path(end)-path(end-1)) == 1
        if path(end) < last
            path = [path near(1)];
        else
            path = [path near(2)];
        end
        continue;
    end
end
```

If no connection via the shortest path within the minimum enclosing rectangle is possible, then there must be a detour. The function *findDetour* determines a path, which is as small as possible, that connects two compo-

nents. Finally, the following *heuristic* was created by using the presented functions.

```
function [S] = heuristic(example_name)

%% Input
struct = importdata(example_name);
T = getfield(struct, 'S'); %terminal sets
c = getfield(struct, 'c'); %columns
r = getfield(struct, 'r'); %rows
dim = getfield(struct, 'dim'); %span

%% Initialization
graph_init;
G = graph;
grid(G,r,c); %grid graph
k = size(T,1); %number of terminal sets
S = cell(k,1); %egde sets for the Steiner trees
VS = cell(k,1); %all vertices of each Steiner tree
T_ = T; %terminals which are not used

%% Algorithm
for j=1:dim
for i=1:k
   path = [];
   if isempty(S{i}) %if edge set i is empty
       temp = T_{i};
       while isempty(path) && size(temp,2) >= 2
            [s_k,t_k] = merMin(G, r, c, temp, temp);
            path = merShortestPath(G, r, c, i, temp(
               s_k), temp(t_k), T, T_);
            if isempty(path)
                temp([t_k]) = [];
            end
       end
       if isempty(path)
            temp = T_{i};
```

```
            [s_k, t_k, path] = findDetour(G, r, c,
               T_{i}, T_{i});
        end
        if s_k ~= 0
            VS{i} = [VS{i} path];
            T_{i} = T_{i}(T_{i}~=temp(s_k));
            T_{i} = T_{i}(T_{i}~=temp(t_k));
        else
            'No solution found'
            return;
        end
    else %if terminal set i is not empty
        if ~isempty(T_{i})
        temp = T_{i};
        while isempty(path) && size(temp,2) >= 2
            [s_k, t_k] = merMin(G, r, c, temp, VS{i
               });
            path = merShortestPath(G, r, c, i, temp(
               s_k), VS{i}(t_k), T, T_);
            if isempty(path)
                temp([s_k]) = [];
            end
        end
        if isempty(path)
            temp = T_{i};
            [s_k, t_k, path] = findDetour(G, r, c,
               T_{i}, VS{i});
        end
        if s_k ~= 0
            VS{i} = [VS{i} path(1:size(path,2)-1)];
            T_{i} = T_{i}(T_{i}~=temp(s_k));
        else
            'No solution found'
            return;
        end
        end
    end
    %save used edges and delete them from the graph
```

```
    for pth=1:size(path,2)-1
         S{i}=[S{i} path(pth) path(pth+1)];
         delete(G,path(pth),path(pth+1))
    end
end
end
draw(G)
end
```

As input-file a struct is used, which contains all terminal sets, number of columns, number of rows and the number of elements from the largest terminal set. The toolbox matgraph is then used to creat a grid graph with all vertices and edges, such that in the end a visualization of the graph $\hat{G}$ can be shown (Figure 7.2). The graph $\hat{G}$ is the graph, where all used edges have been deleted.

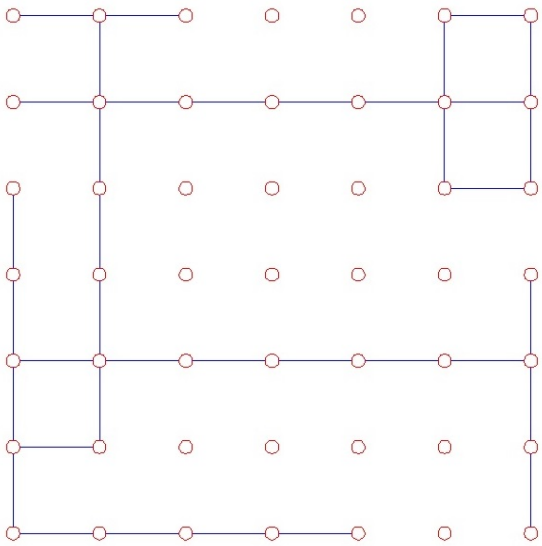


Figure 7.2: Graph with not-used edges

7.3 Results

In order to show the goodness of the heuristic, some validation tests were made. It turned out to be difficult to create a concept that provides a good overview of different results. In the following grid graphs were used as test examples, which have a quadratic structure, starting with 4×4 up to 18×18. The terminal sets of all test example were determined by random numbers. In addition, it must be said that all vertices, which are located on the boundary of the outer face, are assigned to a terminal set. This means, that all the test examples are terminal intensive and generally not easy to solve. For each size of the grid graph the average of time and iterations are given. An iteration in this context means, how often an attempt is made to connect two components. Furthermore, it is listed for how many of the randomly generated test examples a solution was found. In the table 7.1 a summary of the results is given. In order to get a good average 100 test examples were created for each size of the grid graph.

Size ($n \times n$)	Average Time (sec)	Success Rate (%)	Average Iterations
4×4	0.0469	100	4.86
6×6	0.2561	97	7.42
8×8	0.4904	65	13.25
10×10	2.7760	34	16.67
12×12	6.7298	24	24.41
14×14	10.9823	16	29.33
16×16	21.3625	8	34.73
18×18	42.1222	4	41.25

Table 7.1: Results of the random examples

The results in the table above show, that with the size of the graph, also the computations time is growing (Figure 7.3). With a larger grid graph, not only the number of vertices, but also the number of terminals is growing. For

this reason, it is clear that the number of iterations will grow too, because there are more components, which have to be connected. The success rate reveals, that this heuristic reliably gives a solution for small problems, but for larger problems finding a solution is getting unprobably.

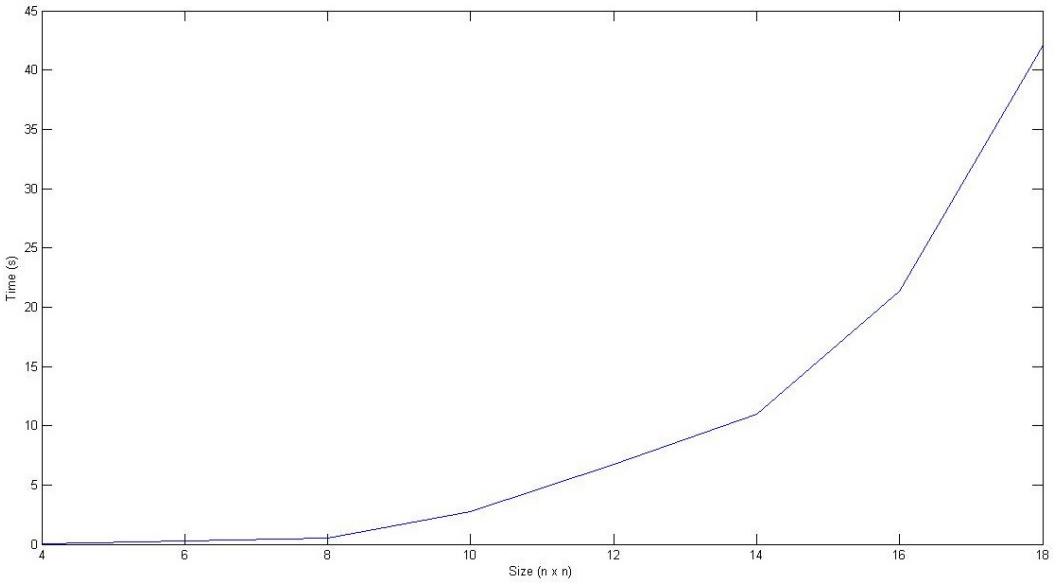


Figure 7.3: Computing Time

In figure 7.4 a solution of an easy example was created by the implemented heuristic. This example can be found in BEAVER [Coh88]. Given that all Steiner trees have a length, which are as minimal as possible, this solution is obviously the optimal solution for the minimal length of the Steiner tree packing problem.

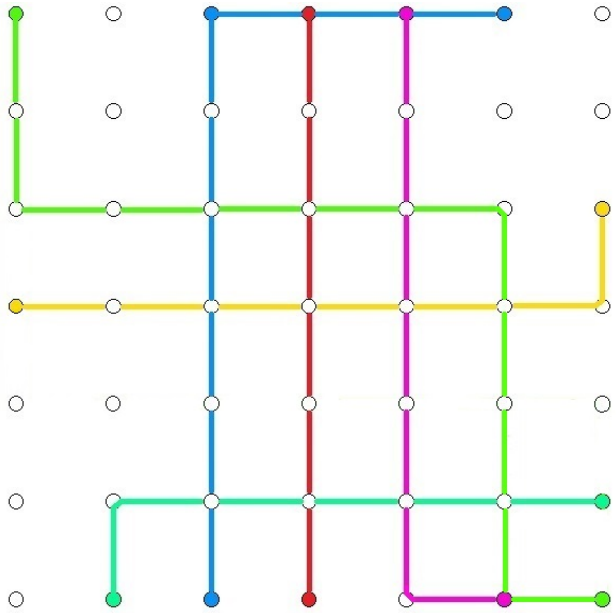
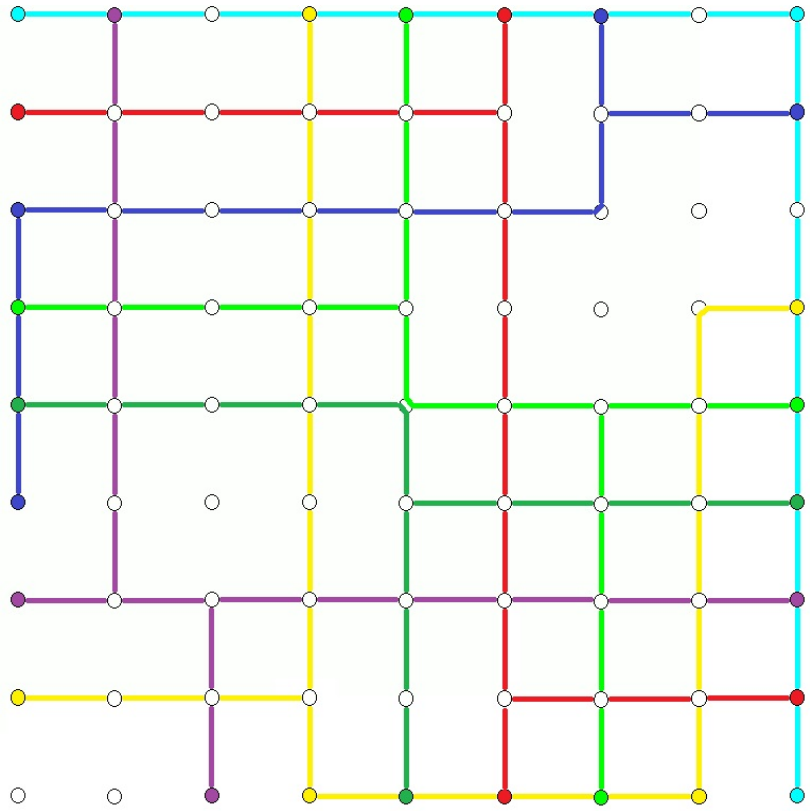
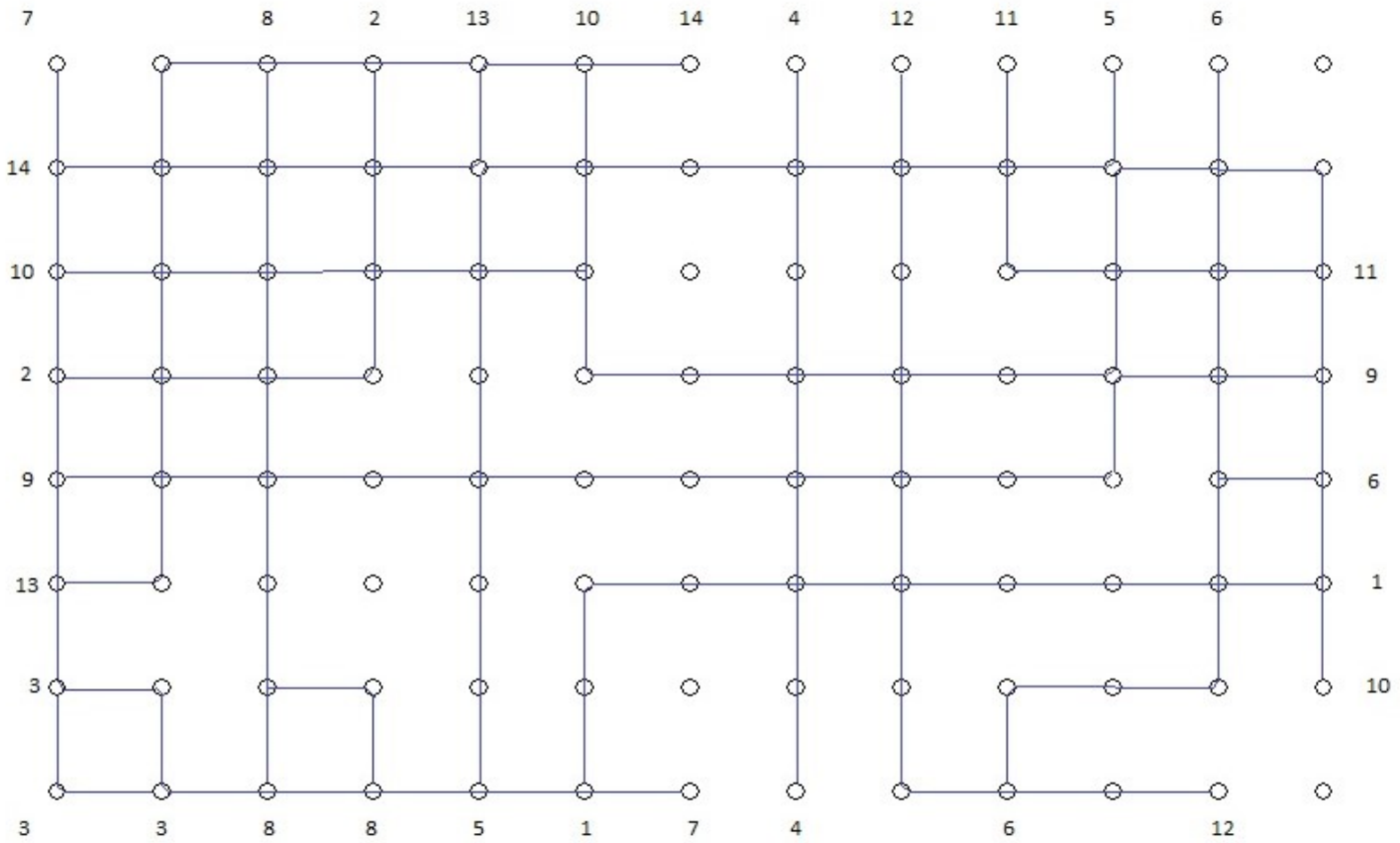


Figure 7.4: Sample switchbox routing

In section 6.2 some difficult test examples from the literature were discussed. This heuristic was unable to find a possible solution to these examples. Thus, the cutting plane algorithm with its solution of the linear program plays an important role in finding solutions of the given problem.

7.4 More Examples




7
8
2
13
10
14
4
12
11
5
6
14
10
11
2
9
9
6
13
1
3
10
3
3
8
8
5
1
7
4
6
12

Chapter 8

Conclusion

In this thesis the problem of packing Steiner trees is considered. It is given an extensive background knowledge of this problem, for example definitions of the graph theory, linear programming and of the underlying Steiner tree problem. Furthermore, there are given some related problems from the literature and the main application of the Steiner tree packing problem, the VLSI-Design, is studied. An important subproblem that occurs in the VLSI-Design, can be solved by using the Steiner tree packing problem. Some detailed routing models, which can be derived from there, are also presented. In the thesis the switchbox routing model was discussed in more detail.

A survey of different integer programming models, which solve the Steiner tree packing problem, is given. It focuses on computational aspects of the problem. The undirected cut formulation is used in the cutting plane algorithm to solve the switchbox routing problem in the knock-knee model. For this the inter program had to be relaxed, such that each variable belong to the interval $[0, 1]$, which is called an linear programming relaxation. The cutting plane algorithm uses the computed solutions of the linear program and tries to find a feasible solution with a heuristic. It came out, that this approach finds either an optimal solition, or at least a lower bound of the problem.

This approach has been shown to be quite promising, but there are still open problems, that need to be solved like seperation methods, where no exact algorithm is provided. With the results, which were presented in this thesis, it can be hoped that the approach can also be used for solving prob-

lems of practical examples. In order to solve large scale problem instances as they occur in practice, the algorithm must be improved. For example, by reducing the computation time with a smaller number of variables in the linear program. This can possibly reached by reducing the area for each terminal set in the minimal enclosing area.

The heuristic of the presented approach has been implemented in Matlab, but with a symplification, that made the heuristic independent of a linear program. This heuristic turned out to be significantly worse than the cutting plane algorithm. Consequently, it can be said, that the linear program plays a very important role of finding solutions for the Steiner tree packing problem in the approach. This algorithm behaves much faster, but for a growing size of the graph, the success rate is running pretty fast towards zero.

Bibliography

[Aaz08] Aazami, Ashkan: Hardness results and approximation algorithms for some problems on graphs. (2008)

[APD03] Althaus, Ernst ; Polzin, Tobias ; Daneshmand, Siavash Vahdati: Improving Linear Programming Approaches for the Steiner Tree Problem. In: *Experimental and Efficient Algorithms: Lecture Notes in Computer Science, Volume 2647, pp 1-14* (2003)

[BB84] Brady, M.L. ; Brown, D.J.: *VLSI routing: Four layers suffice.* Advances in Computing Research, Volume 2, pp 245-258, (1984)

[BB00] Bienstock, D. ; Bley, A.: Capacitated Network Design with Multicast Commodities. In: *International Conference Telecommunication Systems, pp 575-599* (2000)

[Bix91] Bixby, Robert E.: Implementing The Simplex Method: The Initial Basis. In: *Rice University: Department of Mathematical Sciences* (1991)

[BM08] Bondy, J.A. ; Murty, U.S.R.: *Graph Theory.* Graduate Texts in Mathematics, Volume 244, (2008)

[BP83] Burstein, M. ; Pelavin, R.: Hierachical wire routing. In: *IEEE Transactions on Computer-Aided-Design, pp 223-234* (1983)

[Cho94] Chopra, Sunil: Comparison of formulations and a heuristic for packing Steiner trees in a graph. In: *Annals of Operations Research 50, pp 143-171* (1994)

[Coh88] Cohoon, J. P.: BEAVER: computational geometry-based tool for switchbox routing. In: *IEEE Transactions on Computer-Aided-Design CAD-7, pp 684-697* (1988)

[Edm65] Edmonds, J.: Minimum partition of a matroid into independent subsets. In: *Journal of Research National Bureau of Standards Sections B, pp 67-72* (1965)

[FKK03] Frank, A. ; Kiraly, T. ; Kriessel, M.: On decomposing a hypergraph into k connected sub-hypergraphs. In: *Discrete Applied Methematics, Volume 131, pp 373-383* (2003)

[GJ77] Garey, M.R. ; Johnson, D.S.: The rectilinear Steiner tree problem is *NP*-complete. In: *SIAM Journal on Applied Mathematics 32, pp 835-859* (1977)

[GMW93] Grötschel, M. ; Martin, A. ; Weismantel, R.: Packing Steiner Trees: seperation algorithms. In: *Konrad-Zuse-Zentrum für Informationstechnik Berlin, Preprint SC 93-2* (1993)

[GMW96a] Grötschel, M. ; Martin, A. ; Weismantel, R.: Packing Steiner Trees: A Cutting Plane Algorithm and Computational Results. In: *Mathematical Programming, Volume 72, pp 125-145* (1996)

[GMW96b] Grötschel, M. ; Martin, A. ; Weismantel, R.: Packing Steiner Trees: Further Facets. In: *Europ. J. Combinatorics, Volume 17, pp 39-52* (1996)

[GMW97] Grötschel, M. ; Martin, A. ; Weismantel, R.: The Steiner Tree Packing Problem in VLSI-Design. In: *Mathematical Programming, Volume 78, pp 265-281* (1997)

[Gol10] Goldreich, Oded: *P, NP, and NP-Completeness: The Basics of Computational Complexity.* Cambridge University Press, (2010)

[Hak71] Hakimi, S.L.: Steiner's problem in graphs and its implications. In: *Networks 1, pp 113-133* (1971)

[HRW92] Hwang, Frank K. ; Richards, Dana S. ; Winter, Pawel: *The Steiner Tree Problem.* Annals of Descrete Mathematics, Volume 53, 1992

[JLPP02] Jeong, Gue-woong ; Lee, Kyungsik ; Park, Sungsoo ; Park, Kyungchul: A branch-and-price algorithm for the Steiner tree packing problem. In: *Computer & Operations Research 29, pp 221-241* (2002)

[JMS03] Jain, K. ; Mahdian, M. ; Salvatipour, M.R.: Packing Steiner trees. In: *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, pp 266-274* (2003)

[Kar10] Karp, R.M.: Reducibility among combinatorial problems. In: *50 Years of Integer Programming 1958-2008, pp 219-241* (2010)

[KH12] Koch, Thorsten ; Hoàng, Nam-Dũ.: Steiner tree packing revisited. In: *Mathematical Methods of Operations Research, Volume 76, pp 95-123* (2012)

[KPS90] Korte, B. ; Prömel, H.J. ; Schrijver, A.: Steiner trees in VLSI-layout. In: *Paths, Flows, and VLSI-Layout, pp 185-214* (1990)

[Kv80] Kramer, M.R. ; van Leeuwen, J.: The complexity of wire-routing and finding minimum area layouts for arbitrary VLSI circuits. In: *Advances in Computing Research 2, pp 270-281* (1980)

[Lau07] Lau, Lap C.: An Approximate Max-Steiner-Tree-Packing Min-Steiner-Cut Theorem. In: *Combinatorica, Volume 27, pp 71-90* (2007)

[Len90] Lengauer, T.: *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, (1990)

[Lev71] Levin, A.Y.: Algorithm for shortest connection of a group of graph vertices. In: *Sov. Math. Dokl., Volume 12, pp 1477-1481* (1971)

[LL93] Lengauer, T. ; Lügering, M.: Integer programming formulations of global routing and placement problems. In: *Research Report, University of Paderborn* (1993)

[LS91] Liao, Kuo-Feng ; Sarrafzadeh, Majid: Vertex-disjoint trees and boundary single-layer routing. In: *Graph-Theoretic Concepts in Computer Science: Lecture Notes in Computer Science, Volume 484, pp 99-108* (1991)

[Luk85] Luk, W. K.: A greedy switch-box router. In: *Integration, Volume 3, pp 129-149* (1985)

[Mel61] Melzak, Z.A.: On the problem of Steiner. In: *Canadian Mathematical Bulletin, Volume 4, pp 143-148* (1961)

[MG07] Matoušek, Jǐr ; Grtner, Bernd: Integer Programming and LP Relaxation. In: *Understanding and Using Linear Programming Universitext, pp 29-40* (2007)

[MP93] Middendorf, M. ; Pfeiffer, F.: On the complexity of the disjoint paths problem. In: *Combinatorica, Volume 13, pp 185-107* (1993)

[NS08] Naves, G. ; Sebö, A.: Multifow feasibility: an annotated tableau. In: *Research Trends in Combinatorial Optimization, pp 261-283* (2008)

[RRSS93] Reed, B. ; Robertson, N. ; Schrijver, A. ; Seymour, P.D.: Finding disjoint trees in graphs on surfaces. (1993)

[RS90] Robertson, N. ; Seymour, P.D.: An outline of a disjoint paths algorithm. In: *Path, Flows and Vlsi-Layout* (1990)

[RT87] Raghavan, P. ; Thompson, C.D.: Randomized rounding: A technique for provably good algorithms and algorithmic proofs. In: *Combinatorica, Volume 7, pp 365-374* (1987)

[Ruo13] Ruohonen, Keijo: *Graph Theory*. Tampere University of Technology, (2013)

[Sar87] Sarrafzadeh, M.: Channel-routing problem in the knock-knee mode is *NP*-complete. In: *IEEE Transactions on Computer-Aided-Design CAD-6, pp 503-506* (1987)

[Sch08] Scheinerman, Edward R.: MATGRAPH: A Matlab Toolbox For Graph Theory. In: *Department of Applied Mathematics Statistics Vice Dean for Education, Whiting School of Engineering Johns Hopkins University* (2008)

[Str10] Streck, Stefanie: The Fermat problem. In: *Pacific Lutheran University* (2010)

[Van08] Vanderbei, Robert J.: *Linear Programming.* International Series in Operations Research & Management Science, Volume 114, (2008)

[Won84] Wong, R.T.: A dual approach for the steiner tree on a directed graph. In: *Mathematical Programming 28, pp 271-287* (1984)