FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Multilevel Heuristiken für das Rooted Delay-Constrained Minimum Spanning Tree Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Martin Berlakovich
Matrikelnummer 0326695

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: ao.Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl
Mitwirkung: Univ.-Ass. Dipl.-Ing. Mario Ruthmair

Wien, 01.07.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Martin Berlakovich, Gartenweg 10/4/6, 7331 Weppersdorf

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

Wien, 1. Juli 2010

_____

Martin Berlakovich

Danksagung

Nun da sich mein Studium dem Ende nähert möchte ich hiermit meinen Eltern für ihre, nicht nur finanzielle, Unterstützung während meiner Studienzeit danken. Weiterer Dank gebührt meinen Betreuern Prof. Dr. Günther Raidl und Dipl.-Ing. Mario Ruthmair für ihre Unterstützung während dem Verfassen der Diplomarbeit.

Kurzfassung

Viele kombinatorische Optimierungsprobleme sind NP-schwer und können deshalb höchstwahrscheinlich nicht effizient exakt gelöst werden. Aus diesem Grund bedient man sich oft Heuristiken um gute Näherungslösungen für derartige Probleme zu finden. In dieser Arbeit wird das Rooted Delay-Constrained Minimum Spanning Tree Problem (RDCMSTP) behandelt. Es handelt sich hierbei um ein NP-schweres Problem, wobei für einen gegebenen Graphen, dessen Kanten über Kosten und einen Delaywert verfügen, ein kostenminimaler Spannbaum gesucht wird. Jedoch besteht die Einschränkung, dass der Pfad zwischen einem beliebigen Knoten und einer definierten Wurzel nicht länger als eine gegebene Delaygrenze sein darf. Eine praktische Anwendung für dieses Problem ist eine Form von Vertrieb mit einer Lieferzeitgarantie, beispielsweise ein Paketversand mit 24 Stunden Liefergarantie.

Im Zuge dieser Arbeit werden drei Multilevel-Heuristiken für das RDCMSTP vorgestellt. Eine Multilevel-Heuristik besteht grundsätzlich aus zwei Teilen, dem Coarsening und dem Refinement. Während des Coarsenings werden Knoten und/oder Kanten systematisch zusammengefasst um so den Graph zu vereinfachen. Der Graph wird in mehreren Iterationen vereinfacht und es entstehen verschiedene Ebenen des Graphen, sogenannte Levels.

Im zweiten Schritt, dem Refinement, wird, mit Hilfe der Informationen aus dem Coarsening, sukzessiv der gesuchte Spannbaum erstellt. Während des Refinements kann der Graph, mit Hilfe von lokalem Improvement, unter Verwendung geeigneter Nachbarschaftsstrukturen untersucht werden, um so eine Verbesserung der Gesamtlösung zu erzielen.

Die Ergebnisse der entwickelten Multilevel-Verfahren ähneln jener führender bestehender Verfahren bzw. werden manchmal geringfügige Verbesserungen erzielt. Dass heißt, es ist durchaus möglich für das RDCMSTP mit Multilevel-Ansätzen gute Ergebnisse zu erzielen. Es zeigt sich jedoch auch, dass sich die Verwendung von Multilevel-Verfahren, vor allem bei sehr klassischen Ansätzen, als zusätzliche Einschränkungen bei der Lösungsfindung negativ auswirken können. Vor allem die Component-based Multilevel-Heuristik erzielt teilweise sehr gute Ergebnisse und kann durchaus mit bestehenden Verfahren konkurrieren.

Abstract

Many combinatorial optimization problems are NP-hard and can not be solved exactly in an efficient way commonly. Therefore heuristics are often applied to generate good approximations for such problems. This thesis discusses the Rooted Delay-Constrained Minimum Spanning Tree Problem (RDCMSTP) which is NP-hard. The task is to find a minimum spanning tree for a given graph where the edges have cost and delay values minimizing the sum of costs. However, an additional constraint is applied that no path from a specified root node to any other node may exceed a given delay bound. An application for the given problem is a form of distribution with a guarantee of timely delivery, for example a shipment service with a guarantee for delivery within 24 hours.

In this document three multilevel heuristics for the RDCMSTP are introduced. A multilevel heuristic basically consists of two steps, the coarsening and the refinement step. During the coarsening step vertices and/or edges are systematically merged in order to create a reduced graph. The graph is reduced in multiple iterations thus creating multiple levels of detail.

In the second step, the refinement, a solution tree is constructed using the information acquired during the coarsening. While constructing the solution the graph can be searched for local improvements using appropriate neighborhoods thus increasing the quality of the solution.

The results achieved by the multilevel heuristics are similar to leading existing heuristics and there were only slight improvements. This shows, that it is possible to achieve good results for the RDCMSTP with multilevel techniques. Additionally it is shown, that the use of multilevel heuristics, especially for classic approaches, can be seen as an additional constraint during the search for a solution which can have negative effects. Especially the Component-based Multilevel Heuristic achieves very good results and can definitely compete with existing heuristics.

Inhaltsverzeichnis

1 Einleitung

Bei vielen kombinatorischen Optimierungsproblemen handelt es sich um NP-schwere Probleme, siehe Kapitel 2.2 für die genaue Definition. Für Probleme dieser Komplexitätsklasse ist es höchstwahrscheinlich nicht möglich das Problem gleichzeitig exakt und effizient zu lösen. Um dennoch effizient an Lösungen zu gelangen, werden Heuristiken verwendet. Bei Heuristiken handelt es sich um Verfahren, die den Lösungsraum nach guten Näherungslösungen durchsuchen. Näherungslösungen sind im Allgemeinen keine optimalen Lösungen, wobei eine optimale Lösung in der Regel nicht bekannt ist. Näherungslösungen sind Lösungen mit vergleichsweise hoher Qualität, welche nahe an dem unbekannten Optimum einer Probleminstanz liegen. Generell kann man Heuristiken in zwei Kategorien unterteilen, Konstruktions- und Verbesserungsheuristiken.

Eine Konstruktionsheuristik versucht eine Lösung zu konstruieren, indem eine Lösung von Grund auf erstellt wird, beispielsweise durch sukzessives Hinzufügen von Kanten eines Graphen zu einer Teillösung. In jedem Schritt werden mögliche Kandidaten auf ihre Eigenschaften überprüft und anhand dessen eine Entscheidung getroffen. Stehen beispielsweise zwei Kanten mit entsprechenden Kosten zur Auswahl und will man eine Lösung erzeugen, welche minimal in Bezug auf die Kosten ist, besteht die Möglichkeit einfach die günstigere Kante zu verwenden. Man spricht hierbei auch von Greedy-Verfahren. Der Entscheidungsprozess während der Konstruktion einer Lösung ist stark von dem Problem und der Heuristik abhängig. Sobald eine vollständige Lösung erzeugt wurde, ist die Konstruktionsheuristik beendet.

Im Unterschied dazu versucht eine Verbesserungsheuristik eine bestehende Lösung durch geeignete Operationen zu verbessern, beispielsweise durch den Austausch zweier Kanten. Oft wird eine Verbesserungsheuristik auf eine zuvor durch eine Konstruktionsheuristik erstellte Lösung angewandt. Oft werden bei Verbesserungsheuristiken Nachbarschaften von Lösungen definiert. Die Lösungen dieser Nachbarschaften werden aus einer bestehenden Lösung durch relativ einfache Operationen, sogenannte Moves, beispielsweise den Austausch zweier Kanten, erzeugt. Die Anzahl der Lösungen, die in einer Nachbarschaft enthalten sind, man spricht hier von der Größe einer Nachbarschaft, wirkt sich meist auf den Aufwand des Durchsuchens einer Nachbarschaft aus.

Im Verlauf dieser Arbeit werden nun drei Heuristiken vorgestellt, welche auf dem Multilevel-Prinzip basieren. Multilevel-Heuristiken können für viele Probleme verwendet werden, auch bei graphbasierten Problemen wie dem RDCMSTP. Eine Multilevel-Heuristik besteht grundsätzlich aus zwei Teilen, dem Coarsening und dem Refinement. Im Coarsening wird der Graph systematisch iterativ vereinfacht. Während des Coarsenings werden Informationen zur Lösungsfindung extrahiert. Anhand dieser wird im anschließenden Refinement, in dem der Graph wieder rücktransformiert wird, eine Lösung erzeugt. Die Multilevel-Heuristik wird im Detail in Kapitel 4 erläutert.

Das generelle Multilevel Prinzip kann bei vielen Problemen angewandt werden, jedoch müssen sowohl Coarsening als auch Refinement entsprechend an das vorliegende Problem angepasst werden. Die primäre Aufgabenstellung der Diplomarbeit besteht darin eine Multilevel-Heuristik für

das Rooted Delay-Constrained Minimum Spanning Tree Problem zu entwickeln. Hierbei handelt es sich grundsätzlich um die Problemstellung des Auffindens eines minimalen Spannbaums eines Graphen in Bezug auf die Kosten der verwendeten Kanten, allerdings ist bei dem RDCMSTP ein Delay-Constraint zu beachten. Alle Kanten des Graphen weisen nicht nur Kosten auf, sondern verursachen auch eine Verzögerung, den sogenannten Delay. Weiters wird ein Knoten des Graphen als Wurzelknoten festgelegt, der beispielsweise als Sender eines Signals interpretiert werden kann. In Abbildung 1.1 ist ein Beispiel für einen solchen Graphen zu sehen. Der Knoten s ist hier der Wurzelknoten bzw. der Sender. Die Kanten werden durch eckige Klammern beschrieben, wobei der erste Wert die Kosten und der zweite den Delay darstellt.

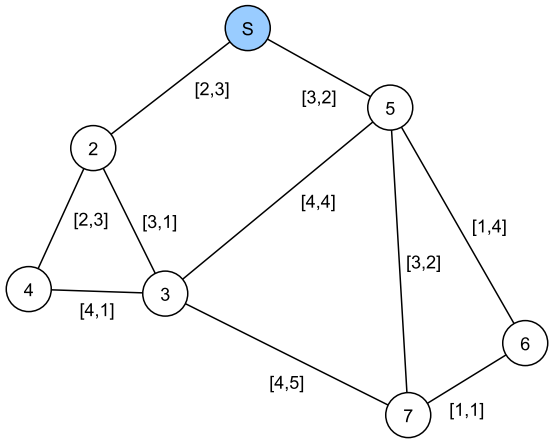

Abbildung 1.1: Ein Beispielgraph. Derartige Graphen stellen die Probleminstanzen des RDCMSTP dar.

Prinzipielle Aufgabenstellung des RDCMSTP ist das Auffinden eines kostenminimalen Spannbaums, während die Pfade zwischen dem gegebenen Wurzelknoten und den anderen Knoten des Graphen eine vorgegebene Delaygrenze B nicht überschreiten dürfen. Eine detaillierte Erklärung der Problemstellung des RDCMSTP findet sich in Kapitel 2.

Literatur zu Multilevel-Heuristiken für dieses Problem ist nach dem derzeitigen Wissensstand nicht vorhanden. Die Hauptaufgabe der Diplomarbeit bestand also darin, Überlegungen über mögliche Multilevel-Heuristiken anzustellen, diese zu implementieren und im Anschluss zu testen und zu vergleichen.

Im Zuge der Diplomarbeit wurden drei verschiedene Multilevel Ansätze verfolgt und entsprechende Heuristiken entwickelt. Diese Heuristiken wurden speziell für das RDCMSTP erstellt, welches im folgenden Kapitel genauer erklärt wird. Das anschließende Kapitel enthält eine Beschreibung der Problemstellung der Diplomarbeit. Kapitel 3 behandelt bestehende exakte als auch heuristische Verfahren zum RDCMSTP und Kapitel 4 behandelt das Multilevel Verfahren im Allgemeinen.

Die Überlegungen und Ansätze für die entwickelten Heuristiken wurden in gemeinsamer Diskussion mit den Betreuern verfeinert. Die erste Heuristik basiert auf der Überlegung des Ausnützens der Knotengrade. Die zweite Heuristik versucht durch ein weiteres Kantenattribut, welches die Effizienz der einzelnen Kanten beschreibt, eine gute Näherungslösung zu finden. Die dritte

und letzte Heuristik versucht schrittweise Kantenketten und Komponenten aufzubauen und zu verbinden. Die Arbeitsweise des Coarsening Teils der drei Heuristiken wird im Detail in den Kapiteln 5 bis 7 erklärt. In Kapitel 8 wird ein Improvement-Verfahren vorgestellt, welches während des Refinements verwendet wird. Kapitel 9 enthält die erzielten Ergebnisse im Detail sowie eine Diskussion dieser, gefolgt von einer Zusammenfassung der gewonnenen Erkenntnisse in Kapitel 10. Die Implementierung der Heuristiken erfolgte in einem von Mario Ruthmair zur Verfügung gestellten C++ Framework. Zum Erstellen der Abbildungen von Graphen wurde das Freeware Tool yED Graph Editor verwendet.

2 Rooted Delay-Constrained Minimum Spanning Tree Problem

Dieses Kapitel beinhaltet eine allgemeine Beschreibung des Rooted Delay-Constrained Minimum Spanning Tree Problems (RDCMSTP), gefolgt von einer kurzen Erläuterung zur Komplexität des Problems. Weiters werden Möglichkeiten zum Preprocessing von Probleminstanzen des RDCMSTP vorgestellt und abschließend einige Anwendungen erläutert.

2.1 Allgemeine Problembeschreibung

Es folgt eine formalere Beschreibung des RDCMSTP. Gegeben ist ein Graph $G = (V, E)$ bestehend aus einer Menge V von n Knoten, wobei genau ein Knoten aus V den Sourceknoten s darstellt, und einer Menge E von m Kanten mit zugehöriger Kostenfunktion $c : E \to \mathbf{R}_0^+$ und Delayfunktion $d : E \to \mathbf{R}^+$. Weiters ist eine Delaygrenze $B > 0$ gegeben. Ziel ist nun das Auffinden eines Baumes $T = (V, E')$ mit $E' \subseteq E$, wobei die Summe der Kosten des Baumes $c(T) = \sum_{e \in E'} c(e)$ minimiert werden soll und gleichzeitig für alle $v \in V$ gilt, dass die Summe der Delays des Pfades $d(P(s, v)) = \sum_{e \in P(s,v)} d(e) \leq B$ ist. Da es sich bei einer Lösung um einen Spannbaum, also einen Baum der alle Knoten des Graphen beinhaltet, handelt, existiert in einer gültigen Lösung immer genau ein Pfad $P(s, v)$.

Abbildung 2.1 zeigt die Auswirkungen des Delay-Constraints auf das Auffinden eines minimalen Spannbaums in einem Graphen. Gegeben ist der Graph G, siehe Abbildung 2.1 a), bestehend aus einer Menge V von Knoten. V beinhaltet den Knoten s, der die Wurzel repräsentiert, und sechs weitere Knoten. Diese sind durch eine Menge E von Kanten verbunden. Die Beschreibung der Kanten ist $[Kosten, Delay]$ zu lesen. Die Kante $(s, 2)$ verursacht also Kosten im Wert von 2 und einen Delay im Wert von 3. Weiters wird eine Delaygrenze von 5 angenommen. Es darf also kein Pfad von s zu einem beliebigen anderen Knoten einen größeren Delay als 5 aufweisen.

Ein minimaler Spannbaum, siehe Abbildung 2.1 b), kann durch den Algorithmus von Kruskal zum Auffinden des minimalen Spannbaums [17] gefunden werden. Hierzu werden sukzessive die kostengünstigsten Kanten hinzugefügt, welche keine Kreise verursachen, bis ein Spannbaum gegeben ist. Dieser Baum ist minimal in Bezug auf die Kosten, hier 12, jedoch werden beim Erstellen des Baums die Delaywerte ignoriert. Dies hat zur Folge, dass die Knoten 4, 6 und 7 einen höheren Delay als die erlaubte Grenze von 5 aufweisen. Man spricht hier von einer Verletzung des Delay-Constraints.

Im Vergleich dazu wird in c) der minimale Spannbaum präsentiert, der den Delay-Constraint nicht verletzt. Man sieht jedoch, dass dadurch deutlich teurere Kanten verwendet werden müssen, was dazu führt, dass die Gesamtkosten des Baumes nun 16 betragen. Weiters kann man aus dem Beispiel auch den Begriff der Striktheit des Delay-Constraints ersehen. Eine Delaygrenze von 5 lässt hier nur eine mögliche Lösung zu. Eine Erhöhung der Delaygrenze auf 6

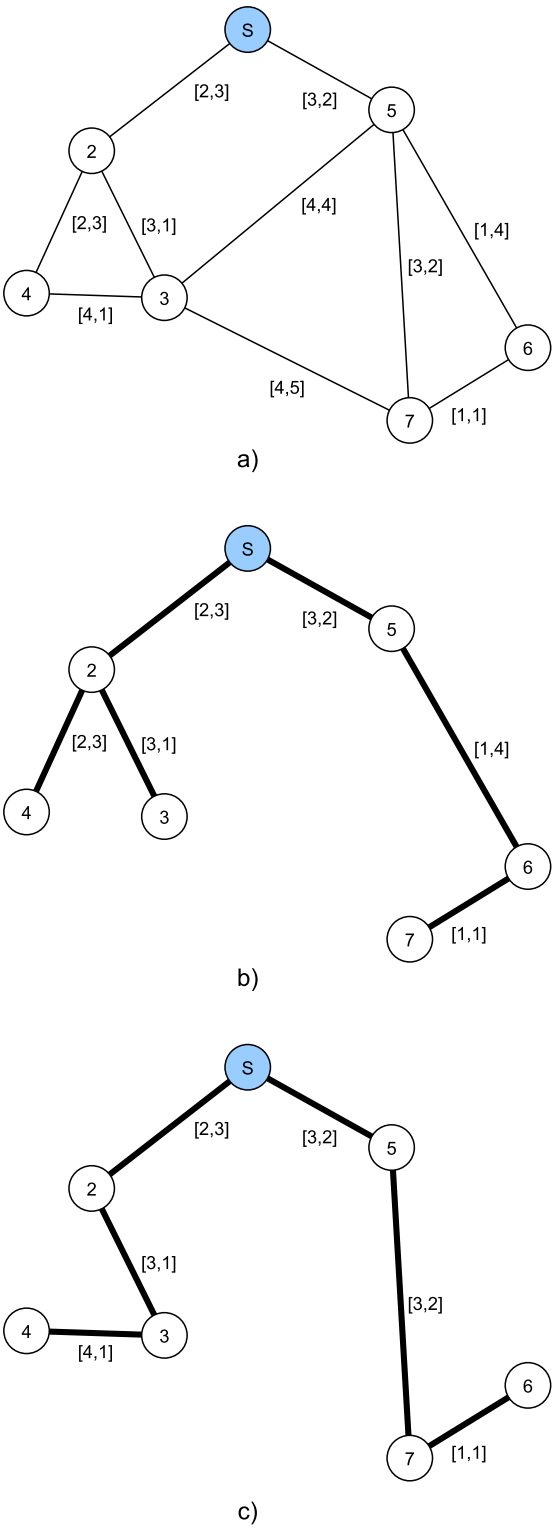

Abbildung 2.1: Diese Abbildung verdeutlicht die Auswirkungen des Delay Constraints. Der minimale Spannbaum b) des Originalgraphen a), verfügt über deutlich geringere Kosten als der durch den Delay-Constraint eingeschränkte Spannbaum c).

würde bereits weitere Spannbäume zulässig machen. Eine Verringerung auf 4 würde gar keine Lösung ermöglichen.

Im generellen Fall führt eine niedrigere Delaygrenze zu einem schwierigeren Problem, da es weniger zulässige Lösungen gibt. Allerdings kann sich dies positiv auf eine Heuristik auswirken, da weniger zulässige Lösungen gleichzeitig weniger Möglichkeiten bedeutet, was die Lösungsfindung einer Heuristik beschleunigt.

2.2 Komplexität

Während das gewöhnliche Minimum Spanning Tree Problem, beispielsweise mit dem Algorithmus von Kruskal [17], in polynomieller Zeit exakt gelöst werden kann, führt der zusätzliche Delay-Constraint des RDCMSTP zur NP-Schwierigkeit des Problems, wobei NP hier für Nondeterministic Polynomial-time steht. Dies bedeutet, dass solche Probleme mittels einer nichtdeterministischen Turingmaschine in polynomieller Zeit entschieden werden können. Bei einer nichtdeterministischen Turingmaschine geht man von uneingeschränkter Parallelisierbarkeit der Berechnungspfade aus, welche allerdings technisch nicht realisiert werden kann. Für praktische Zwecke gelten NP-schwere Probleme daher als nicht effizient lösbar. Zusätzlich bedeutet NP-schwer, dass, sofern P $\neq$ NP, kein Algorithmus zum Ermitteln einer optimalen Lösung mit polynomieller Laufzeit existiert [7]. Der Beweis erfolgt der NP-Schwierigkeit erfolgt mittels Reduktion des Problems, auf ein anderes Problem, für das die NP-Schwierigkeit bereits bewiesen wurde.

Ein Spezialfall des RDCMSTP ist das sogenannte Hop-Constrained Minimum Spanning Tree Problem (HMSTP). Bei diesem Spezialfall gilt für alle Kanten $e \in E$, $d(e) = 1$, die Delaygrenze beschreibt also die maximale Anzahl verwendbarer Kanten für einen Pfad $P(s, v)$. In [10] wird die NP-Schwierigkeit dieses Problems begründet. Da das RDCMSTP prinzipiell eine Generalisierung des HMSTP darstellt, ist auch das RDCMSTP NP-schwer, siehe [2]. Die weitere Beweisführung basiert auf der Reduktion auf das Exact Cover by 3-Sets Problem, siehe [24].

2.3 Preprocessing

Die Laufzeit eines Algorithmus für das Rooted Delay-Constrained Minimum Spanning Tree Problem hängt in der Regel von der Knoten- und Kantenanzahl ab. Eine Verringerung der Kantenanzahl durch Ausscheiden von Kanten, die nicht in einer gültigen Lösung enthalten sein können, kann daher zu Laufzeitverbesserungen führen. Im weiteren werden Preprocessing-Schritte erklärt, welche auf Instanzen des RDCMSTP angewandt werden können.

Der erste Preprocessing-Schritt sucht nach Kanten, deren Verwendung verhindert, dass ein Knoten innerhalb der deklarierten Delaygrenze zum Sourceknoten verbunden werden kann. Hierzu werden die kürzesten Pfade bezüglich der Delays zwischen den Knoten des Graphen und dem Sourceknoten ermittelt. Dies geschieht mittels des Dijkstra Algorithmus, siehe [3]. Gilt nun für beide Knoten einer Kante, dass der kürzeste Pfad zum Sourceknoten plus der Delay der Kante die Delaygrenze überschreitet, kann diese Kante nicht Teil einer gültigen Lösung sein. Ein Verwenden einer derartigen Kante hätte zur Folge, dass für einen der beteiligten Knoten, eine Verletzung des Delay-Constraints vorliegt. Da diese Kanten nicht Teil einer gültigen Lösung sein können, können sie aus dem Graph entfernt werden. Dies geschieht mittels der Formel 2.1.

$$d_{min}(s,i) + d(i,j) > B \wedge d_{min}(s,j) + d(i,j) > B \tag{2.1}$$

Für eine Kante (i,j) wird also geprüft, ob der minimale Delay zwischen den Knoten der Kanten und dem Sourceknoten $d_{min}(s,i)$ bzw. $d_{min}(s,j)$ plus der Delay der Kante, $d(i,j)$, die Delaygrenze B übersteigt. Trifft dies in beiden Fällen zu, so kann die Kante unmöglich Teil einer gültigen Lösung sein und wird entfernt.

Während der erste Preprocessing-Schritt Kanten eliminiert, welche nicht in einer gültigen Lösung enthalten sein können, ist das Ziel des zweiten Preprocessing-Schritts jene Kanten zu eliminieren, die nicht in einer optimalen Lösung enthalten sein können. Hier wird für eine Kante (i,j) geprüft, ob es günstiger ist, sowohl in Bezug auf Kosten als auch Delay, beide Knoten direkt zum Sourceknoten zu verbinden, als einen der Knoten zum Sourceknoten zu verbinden und die Kante (i,j) zu verwenden um für den anderen Knoten ebenfalls eine Verbindung zum Sourceknoten herzustellen. Dies geschieht mittels der Formel 2.2.

$$c(s,j) \leq c(i,j), d(s,j) \leq d_{min}(s,i) + d(i,j) \wedge c(s,i) \leq c(i,j), d(s,i) \leq d_{min}(s,j) + d(i,j) \tag{2.2}$$

Für eine Kante (i,j) wird geprüft, ob die Kosten der Kante zwischen dem Sourceknoten s und dem Knoten j geringer sind als die Kosten der Kante zu dem Knoten i. Ist dies der Fall, so kann der Knoten j kostengünstiger zum Sourceknoten verbunden werden als zum Knoten i. Ähnlich wird der Test für den Delay durchgeführt, wobei hier nicht der Delay der Kante (s,i) verwendet wird, sondern der minimale Delay zwischen den beiden Knoten. Ist auch hier der Delay der Kante (s,j) geringer so ist es in jedem Fall besser den Knoten j direkt zum Sourceknoten zu verbinden als den Umweg über den Knoten i zu gehen. Man beachte, dass selbst im Fall von Gleichheit der direkte Weg bevorzugt wird, da das Entfernen einer Kante nur Vorteile bringen würde. Der selbe Test wird auch für den Knoten i durchgeführt, wobei nun der Umweg über den Knoten j getestet wird. Ist es auch hier von Vorteil, dass der Knoten direkt zu s verbunden wird, ist die Kante (i,j) keinesfalls teil einer optimalen Lösung und wird eliminiert.

Die Elimination dieser Kanten beschleunigt nicht nur das Coarsening, da weniger Kanten geprüft werden müssen, sondern auch ein eventuelles Improvement während des Refinements, da weniger Kanten während des Improvements überprüft werden müssen.

2.4 Anwendungen

Das Rooted Delay-Constrained Minimum Spanning Tree Problem wird oft durch das Beispiel einer zentralen Sendestation erklärt. Es soll eine Verbindung zwischen einem Sender und mehreren Empfängern erzeugt werden. Die Wahl der Verbindungen zwischen dem Sender und den einzelnen Empfängern soll so erfolgen, dass die Kosten des gesamten Netzwerks minimal sind. Es liegt also prinzipiell ein minimales Spannbaum Problem vor. Im Unterschied zum gewöhnlichen Minimum Spanning Tree Problem ist nun allerdings ein Wurzelknoten, der Sender, ausgewählt und die Verbindungen werden nicht nur durch Kosten beschrieben, sondern auch durch eine Verzögerung, den Delay. Der Delay, den das Signal vom Sender benötigt um zu jedem einzelnen Empfänger zu gelangen, soll hier nicht größer als ein bestimmter Grenzwert sein, um so eine gewisse Quality of Service zu gewährleisten. Ziel ist es also die Verbindungen so zu wählen, dass die Kosten des Netzwerks minimal sind, jedoch unter der Einschränkung, dass der Delay keines

Empfängers größer als eine gegebene Delaygrenze ist.

Ein weiteres Beispiel ist ein Produzent von verderblicher Ware. Hier muss der Produzent seine Kunden über entsprechende Transportwege beliefern. Ziel ist natürlich wieder die Kosten des Transports minimal zu halten, mit der Einschränkung, dass der Transport eine gegebene Maximaldauer nicht überschreiten darf, damit die Ware nicht verdirbt.

Auch ein Paketversand mit Lieferzeitgarantie ist ein typisches Beispiel für das RDCMSTP. Man kann den Wurzelknoten als eine Versandzentrale sehen und die anderen Knoten als Kunden oder Zwischendepots. Die garantierte Lieferzeit stellt die Delaygrenze dar und die Transportwege die Kanten mit ihren Kosten und Delays.

Generell kann jeder Versand mit einer Form von Lieferzeitgarantie als ein RDCMSTP gesehen werden. In einem gewissen Sinn kann auch das Beispiel der zentralen Sendestation als Versand gesehen werden.

3 Bestehende Verfahren

Bei dem Rooted Delay-Constrained Minimum Spanning Tree Problem (RDCMSTP) handelt es sich um ein noch relativ unerforschtes Problem. Während für Spezialfälle wie das Hop-Constrained Minimum Spanning Tree Problem (HMSTP), hier weisen alle Kanten einen Delay von 1 auf, zahlreiche Verfahren existieren, sind in der Literatur nur wenige Verfahren zur Lösung des RDCMSTP vertreten. Exakte Ansätze für kleinere Probleminstanzen werden von Gouveia et al. in [12] behandelt. Weiters existieren zwei Heuristiken. Die sogenannte Bounded Delay Broadcast (BDB) Heuristik [23] basiert auf dem Prim Algorithmus für das Minimum Spanning Tree (MST) Problem [19]. Eine zweite Heuristik basiert auf Kruskal's MST Algorithmus [17], welcher auch namensgebend für die Kruskal-basierte Heuristik (KBH) [22] ist. Diese bestehenden Verfahren werden in diesem Kapitel erläutert.

3.1 Exakte Verfahren

Neben Heuristiken existieren auch exakte Verfahren zur Lösung des Problems, siehe [12]. Hier werden Modelle für das RDCMSTP vorgestellt, mit deren Hilfe das Problem exakt gelöst werden. Diese basieren unter anderem auf Constraint Shortest Paths, also kürzeste Pfade mit einem Constraint, in diesem Fall dem Delay-Constraint. Bei dem Constrained Shortest Path Problem handelt es sich ebenfalls um ein NP-schweres Problem, allerdings existieren pseudopolynomielle Algorithmen [4]. Im Wesentlichen wird für einen Knoten k für jeden diskreten Wert h zwischen 0 und der Delaygrenze B ein kürzester Pfad zum Sourceknoten berechnet. Während h schrittweise erhöht wird, wird geprüft, ob eine günstigere Verbindung für einen Knoten j, welcher sich im Set S_h befindet, gefunden werden kann. Ist dies der Fall, werden die neuen minimalen Kosten des Knotens j gespeichert. Weiters werden alle Kanten (i, j) geprüft, für welche die entsprechenden Bedingungen aus Zeile 7 des Algorithmus 1 gelten. Bei Kanten, für die diese Bedingungen gelten, wird der Knoten i, falls noch nicht vorhanden, zu dem Set $S_h + d(i, j)$ hinzugefügt und der Wert von $f(i, h)$ aktualisiert, wobei $f(i, h)$ hier die minimalen Kosten des Pfades des Knotens i zum Sourceknoten unter dem Constraint h beschreibt. Algorithmus 1 zeigt die Berechnung von Constrainted Shortest Paths, siehe auch [12].

Aus einer Formulierung des RDCMSTP als Pfadmodell kann eine Lagrange-Relaxierung abgeleitet werden. Diese ermöglicht eine untere Grenze für die Abschätzung des Optimums, siehe [8]. Eine Näherung der optimalen Lagrange-Multiplikatoren kann durch die sogenannte Subgradient Optimierung [14] gefunden werden. Weiters wird ein Column-generation-Ansatz vorgestellt sowie eine Formulierung basierend auf Layered Graphs, welche eine Generalisierung eines Ansatzes für das Hop-Constrained Minimum Spanning Tree Problem darstellt [11]. Die Tests dieser exakten Verfahren beschränken sich allerdings auf sehr kleine vollständige Instanzen, 20 bis 40 Knoten. Für größere vollständige Instanzen mit 100 oder mehr Knoten wären diese Ansätze nicht mehr vernünftig einsetzbar.

Algorithmus 1 : Constrained Shortest Path aus [12]

Input : Graph, Knoten k
Output : Constrained Shortest Path zum Knoten k

Setze $S_h \leftarrow \emptyset$ für $h = 1, \ldots, B$ and $S_0 \leftarrow \{0\}$
Setze $MinCost(i) \leftarrow \infty$ für $i \in V \setminus \{0\}$
forall h *mit* $h = 0, \ldots, B-1$ **do**
  forall $j \in S_h$ *mit* $f(j,h) \leq MinCost(k)$ **do**
    $MinCost(j) \leftarrow \text{Min}\{MinCost(j), f(j,h)\}$
  forall $j \in S_h$ *mit* $f(j,h) \leq MinCost(k)$ **do**
    forall $(i,j) \in E$ *mit* $d(i,j) + h \leq B$ *und* $c(i,j) + f(j,h) <$ $Min\{MinCost(i), MinCost(k), f(i, h + d(i,j))$ **do**
      if $i \notin S_{h+d(i,j)}$ **then**
        Füge i zu $S_{h+d(i,j)}$ hinzu
      $f(i, h + d(i,j))) \leftarrow costs(i,j) + f(j,h)$
      if $i = k$ **then**
        $MinCost(k) = f(i,h)$

3.2 Heuristiken

Es gibt bereits zwei Konstruktionsheuristiken für das RDCMSTP, die jedoch nicht auf dem Multilevel-Prinzip beruhen. Bei der BDB-Heuristik [23] handelt es sich um eine Heuristik für das RDCMSTP, die auf dem Prim-Algorithmus für das Minimum Spanning Tree (MST) Problem [19] basiert. Ausgehend von dem Sourceknoten s werden die verbleibenden Knoten nacheinander verbunden ohne den Delay-Constraint zu verletzen. Ist kein Anbinden von verbleibenden Knoten mehr möglich, wird eine sogenannte Delay-Relaxation durchgeführt. Hier wird für einen Knoten v ein Pfad zum Sourceknoten gesucht, der einen geringeren Delay als die aktuelle Verbindung aufweist. Es wird die maximale Delay-Relaxation gesucht. In einer zweiten Phase wird im Zuge des sogenannten Link-Replacement nach günstigeren Verbindungen gesucht. Im Prinzip handelt es sich bei Phase 2 also um eine Improvementphase.

Eine neuere Heuristik, die sogenannte Kruskal-basierte Heuristik (KBH), findet sich in [22]. Diese Heuristik ähnelt, wie der Name bereits verrät, dem Kruskal-Algorithmus zum Finden minimaler Spannbäume [17]. Die nach ihren Kosten sortierten Kanten werden, sofern der Delay-Constraint dadurch nicht verletzt wird, zur Lösung hinzugefügt. Es werden also Knoten verschmolzen, wodurch Teilbäume, auch Komponenten genannt, entstehen. Nachdem alle Kanten überprüft wurden, werden eventuelle übriggebliebene Komponenten durch eine Methode basierend auf Shortest-Delay-Paths, die kürzesten Pfade eines Knotens oder einer Komponente zum Sourceknoten basierend auf den Delays, an den Baum angebunden. So entsteht eine sehr gute Näherungslösung. Die KBH wird im späteren Verlauf der Arbeit als Vergleichsheuristik für die selbst erstellten Multilevel-Heuristiken verwendet.

3.3 Rooted Delay-Constrained Steiner Tree Problem

Bei dem Rooted Delay-Constrained Steiner Tree Problem handelt es sich um eine Generalisierung des RDCMSTP. Im Unterschied zum RDCMSTP ist hier eine Menge von Knoten gegeben, welche innerhalb einer gegebenen Delaygrenze erreicht werden müssen. Die verbleibenden Knoten des Graphen können hierzu verwendet werden, ihre Verwendung ist jedoch nicht zwingend.

Zum Rooted Delay-Constrained Steiner Tree Problem gibt es, im Unterschied zum RDCMSTP, zahlreiche aktuellere Publikationen. Es wurden Meta-Heuristiken wie zum Beispiel Greedy Randomized Adaptive Search Procedures, siehe [25] und [28], path-relinking [9] und Variable Neighborhood Descent [20] angewandt. Weiters werden in [18] exakte Verfahren, basierend auf Integer Linear Programming, vorgestellt.

4 Multilevel-Heuristik

Multilevel-Heuristiken existieren für eine Vielzahl von kombinatorischen Optimierungsproblemen. In [26] wird die generelle Multilevel-Heuristik vorgestellt. Dieser vorgestellte Algorithmus ist in Algorithmus 2 ersichtlich. Anwendungen des Multilevel-Prinzips existieren für eine Vielzahl bekannter Probleme wie das Graph Partitioning Problem [27] oder das Vehicle Routing Problem [21]. In diesem Kapitel wird die generelle Arbeitsweise einer Multilevel-Heuristik erklärt und Anwendungen des Multilevel-Prinzips auf andere Probleme vorgestellt. Anschließend wird das Multilevel-Prinzip anhand eines graphischen Beispiels verdeutlicht. Weiters werden spezifische Anfordungen des Rooted Delay-Constrained Minimum Spanning Tree Problems an eine Multilevel-Heuristik diskutiert.

Algorithmus 2 : Multilevel

Input : Probleminstanz P_0
Output : Lösung C_0

```
1
2  l = 0
3
4  while coarsening do
5  |
6  |  P_{l+1} = coarsen(P_l)
7  |  l = l + 1
8
9  C_l = initialise(P_l)
10
11 while l > 0 do
12 |
13 |  l = l − 1
14 |  C_l^0 = extend(C_{l+1}, P_l)
15 |  C_l = refine(C_l^0, P_l)
```

4.1 Allgemeine Beschreibung

Das Multilevel-Prinzip ist im generellen Fall sehr einfach. Ein Problem wird rekursiv immer weiter vereinfacht um mehrere Detailstufen zu erhalten. Man spricht hier auch von einer Approximationshierarchie, da die Vereinfachungen immer Approximationen des Originalproblems darstellen. Dieser Teil einer Multilevel-Heuristik wird als Coarsening bezeichnet.

Nach dem Coarsening wird das ursprüngliche Problem Schritt für Schritt bzw. Level für Level wieder hergestellt. Die zuvor durchgeführten Vereinfachungen werden rückgängig gemacht, man spricht hier von dem Refinement. Im Zuge des Refinements kann nach alternativen Lösungen gesucht werden. Dies geschieht meist durch Improvementverfahren wie lokaler Suche oder auch variabler Nachbarschaftssuche [13]. Sobald das ursprüngliche Problem durch kontinuierliches Refinement wiederhergestellt wurde, ist das Refinement und somit auch die Multilevel-Heuristik abgeschlossen. Eine Multilevel-Heuristik besteht also immer aus zwei Teilen, dem Coarsening und dem Refinement.

Die grundlegende Motivation für die Verwendung eines Multilevel-Verfahrens liegt darin, das Problem in mehreren Detailebenen zu bearbeiten. Im Lauf des Coarsenings wird ein Problem stetig vereinfacht, beispielsweise wird bei einem graphbasierten Problem die Knoten und Kantenanzahl reduziert. Mit zunehmender Vereinfachung des Problems wird es auch möglich Lösungsansätze zu verwenden, welche zuvor aufgrund von hohem Aufwand nicht in Frage kommen. Die Frage, die sich für ein gegebenes Problem stellt, ist, ob eine Multilevel-Variante eines Algorithmus für das gegebene Problem einen Vorteil bringt. Diese Frage wird in [1] behandelt. Weiters wird eine Anwendung des Multilevel-Prinzips auf das Graph Partioning Problem (GPP) gezeigt. Im folgenden Abschnitt wird diese Anwendung erläutert.

4.2 Anwendung des Multilevel Prinzips auf das Graph Partitioning Problem

Bei dem Graph Partitioning Problem (GPP) handelt es sich ebenfalls um ein NP-schweres Problem. Gegeben ist ein Graph $G(V, E)$ mit Gewichten auf den Knoten und/oder Kanten. Gesucht ist eine Unterteilung oder Partitionierung des Graphen in k Partitionen, wobei jede Partition eine möglichst gleiche Summe von Knotengewichten aufweisen soll. Weiters soll die Summe der Kantengewichte, welche zwischen den Partitionen verlaufen, das sogenannte *cut-weight*, minimiert werden.

Der generelle Multilevel-Ansatz für dieses Problem besteht nun darin, Paare von Knoten zusammenzufügen und den Graph so zu vereinfachen, bis die Knotenanzahl eine gegebene Grenze unterschreitet. Im Anschluss wird mittels eines entsprechenden Algorithmus eine Lösung für dieses einfachste Level erzeugt. Danach wird der Graph im Zuge des Refinements schrittweise wiederhergestellt, wobei die Lösung des vorhergehenden Levels stets als Startlösung des aktuellen Levels verwendet wird. Abbildung 4.1 zeigt den prinzipiellen Ablauf. In den oberen Abbildungen wird der Graph von links nach rechts durch Coarsening schrittweise vereinfacht. Danach wird die Partitionierung durchgeführt und der Graph in der unteren Reihe von rechts nach links wiederhergestellt.

Ein Ansatz zur Verwendung des Multilevel-Prinzips für Partitionierung stammt aus [15]. Für das Coarsening wird der sogenannte edge-contraction Algorithmus verwendet. Das Grundprinzip dieses Algorithmus liegt darin, für einen gegebenen Graphen ein maximales unabhängiges Subset *maximal independent subset* der Kanten bzw. ein maximales Matching der Knoten zu finden. Ein solches Subset besitzt die Eigenschaft, dass keine zwei Kanten zu dem gleichen Knoten inzident sind. Kann ein derartiges Subset nicht mehr erweitert werden, spricht man von einem maximalen Subset. Wurde ein solches Subset gefunden, werden die Knoten der im Subset vorhandenen Kanten verschmolzen. Das Knotengewicht des daraus resultierenden Knoten wird

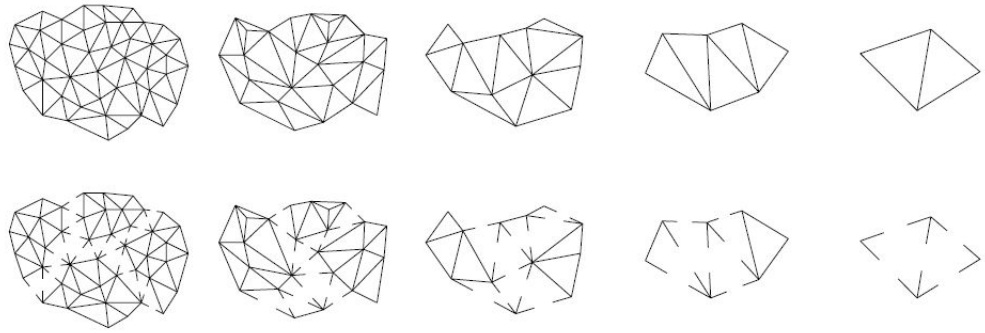

Abbildung 4.1: Ein Beispiel von Multilevel-Partitionierung aus [1].

aus den Knotengewichten der ursprünglichen gewonnen. Alle Kanten, die durch diesen Prozess nicht entfernt werden, werden in das nächste Level übernommen. Ein maximales unabhängiges Subset kann einfach gefunden werden. Jeder ungematchte Knoten wird mit einem noch nicht verwendeten Knoten, zu dem eine Kante besteht, gematcht, oder, falls ein solcher Knoten nicht verfügbar ist, mit sich selbst. [16] zeigt, dass es von Vorteil ist, hier darauf zu achten, möglichst teure Kanten zu verwenden.

Der beschriebene Vorgang wird im Zuge des Coarsenings fortgesetzt bis die Knotenanzahl des aktuellen Levels einen bestimmten Grenzwert erreicht. Für eine k-Partiton wäre k ein einfacher Grenzwert. Es ist jedoch von Vorteil wenn diese erste bzw. Initial-Partition möglichst gleiche Knotengewichte aufweist. Daher erweist es sich als nützlich den Grenzwert für den Abbruch des Coarsenings höher zu setzen um eine möglichst ausbalancierte Initial-Partition konstruieren zu können.

Während des Refinements wird die Partition des vorhergehendes Levels jeweils um das aktuelle Level erweitert und es entsteht erneut eine Initial-Partition. Im Anschluss wird versucht, diese Partition durch entsprechende Verbesserungsheuristiken zu verbessern. Es gibt eine Vielzahl von Verbesserungsheuristiken, wobei in neueren Arbeiten meist der Algorithmus von Fiduccia und Mattheyses [5] mit linearer Laufzeit verwendet wird.

4.3 Beispiel anhand eines Graphen

Im folgenden Abschnitt wird das Multilevel-Prinzip anhand eines selbst erstellten Beispiels erklärt. Gegeben ist ein Graph, welcher über einen Wurzelknoten s verfügt und dessen Kanten über Kosten verfügen. Es soll nun mittels Multilevel-Heuristik ein minimaler Spannbaum erzeugt werden. Zur einfacheren Illustration des prinzipiellen Ablaufs des Multilevel-Verfahrens wird in diesem Beispiel auf Constraints und Kantenbeschriftungen verzichtet. Es soll also lediglich ein minimaler Spannbaum erzeugt werden. Hierbei ist zu beachten, dass dieses Problem grundsätzlich effizienter gelöst werden kann und es hier nur zu Demonstrationszwecken über einen Multilevel-Ansatz gelöst werden soll.

Im Zuge des Coarsenings wird der Graph systematisch vereinfacht. Dies geschieht, indem bestimmte Knoten oder Kanten miteinander verschmolzen werden. Die Wahl der Knoten bzw. Kanten ist hier von dem gegebenen Problem und der aktuellen Heuristik abhängig. Dies wird

wiederholt, bis ein Abbruchkriterium erfüllt ist, beispielsweise bis keine weiteren Verschmelzungen mehr möglich sind. Das aktuelle Level ist somit beendet. Dieser Vorgang wird im nächsten Level wiederholt. Durch das Zusammenfügen von Knoten bzw. Kanten wird der Graph dahingehend vereinfacht, dass immer weniger Knoten und Kanten für den weiteren Verlauf des Coarsenings eine Rolle spielen und so der Aufwand für jede weitere Iteration verringert wird. Die Kehrseite ist, dass durch das Festlegen von Teilen der Lösung der durchsuchbare Lösungsraum verringert wird. Es ist also wichtig die Balance zwischen dem Grad der Vereinfachung und der Effizienz zu finden. Das Coarsening endet, wenn in dem aktuellen Level keine neuen Verschmelzungen möglich sind.

Nachdem das Coarsening durchgeführt wurde, folgt das Refinement. Hierbei wird anhand der aus dem Coarsening gewonnenen Informationen, beispielsweise welche Kanten in der Lösung enthalten sind oder welcher Knoten mit einem anderen Knoten verschmolzen wurde, die Lösung konstruiert. Dies geschieht, indem die zuvor im Coarsening durchgeführten Vereinfachungen bzw. Verschmelzungen nun Schritt für Schritt bzw. Level für Level wieder rückgängig gemacht werden, um so den ursprünglichen Graphen zu erzeugen. Im Zuge des Refinements kann versucht werden, die Lösung durch lokales Improvement zu verbessern. Hierzu werden mögliche Verbesserungen innerhalb des aktuellen Graphen, also des aktuellen Levels, gesucht. Allerdings ist hier zu beachten, dass während des Refinements die Knoten- und Kantenanzahl des Graphen zunimmt, was zu einem deutlichen Mehraufwand führt. Es ist also wichtig entsprechende Verbesserungsverfahren zu verwenden, damit die Laufzeit nicht zu hoch wird. Das Refinement endet, sobald der Originalgraph wiederhergestellt ist bzw. das Improvement in diesem untersten Level durchgeführt wurde.

Die Abbildungen 4.2 und 4.3 zeigen den Vorgang des Coarsenings und des Refinements anhand eines Beispiels. Gegeben ist ein Graph mit sieben Knoten, wovon einer den Wurzelknoten darstellt. Es soll nun ein Spannbaum gefunden werden. Im Zuge des Coarsenings werden die Knoten 2 und 3, 4 und 5 und 6 und 7 zusammengefasst. Daraus resultiert ein vereinfachter Graph, bestehend aus 4 Knoten und 5 Kanten. Im nächsten Schritt werden bereits alle verbleibenden Knoten zum Sourceknoten verbunden und das Coarsening ist somit abgeschlossen. Die erlangten Informationen bestehen in Form der verwendeten Kanten. Alternativ könnte man den Knoten auch die entsprechenden Vorgängerknoten zuweisen, um den Baum zu beschreiben.

Anschließend wird im Refinement der Baum aufgebaut. Ausgehend von der Wurzel wird der Baum bzw. Graph Schritt für Schritt bzw. Level für Level erweitert, um so die Lösung zu produzieren. Man beachte, dass in Abbildung 4.3 nur die Kanten des Baumes abgebildet werden. Andere Kanten, die zwischen den neu hinzugekommenen Knoten bzw. diesen und bereits bestehenden verlaufen, werden ebenfalls wieder berücksichtigt, um so ein lokales Improvement durchführen zu können.

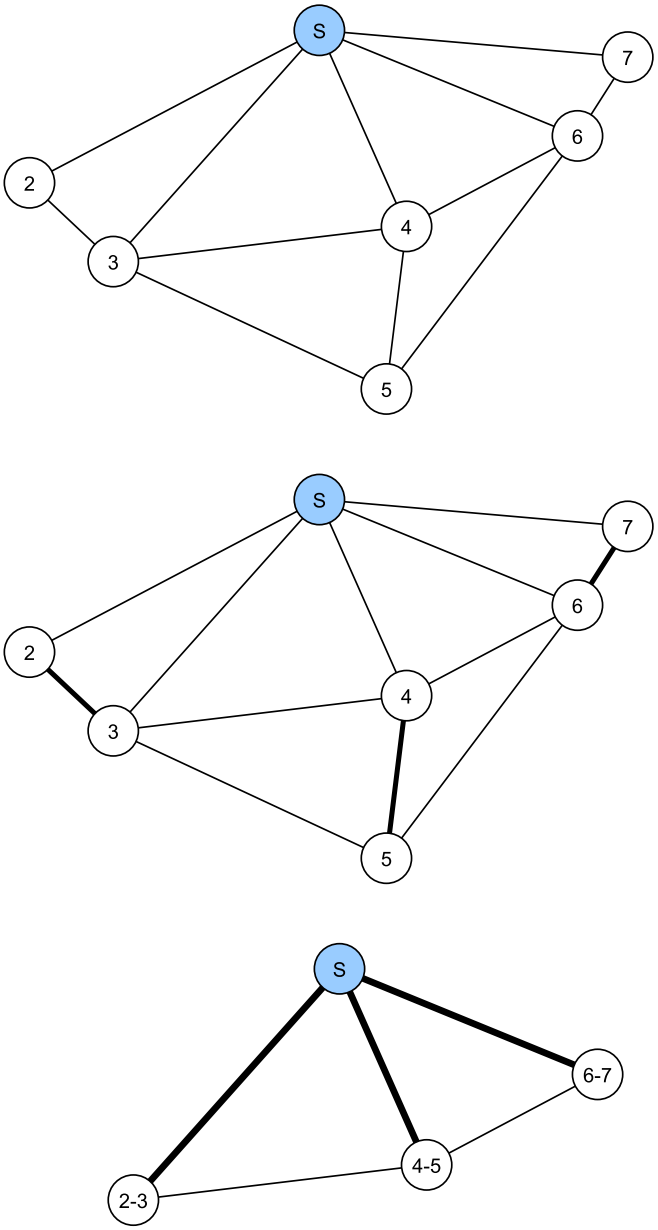

Abbildung 4.2: Das Coarsening. Der Graph wird sukzessive vereinfacht, indem Knoten zusammengefasst werden.

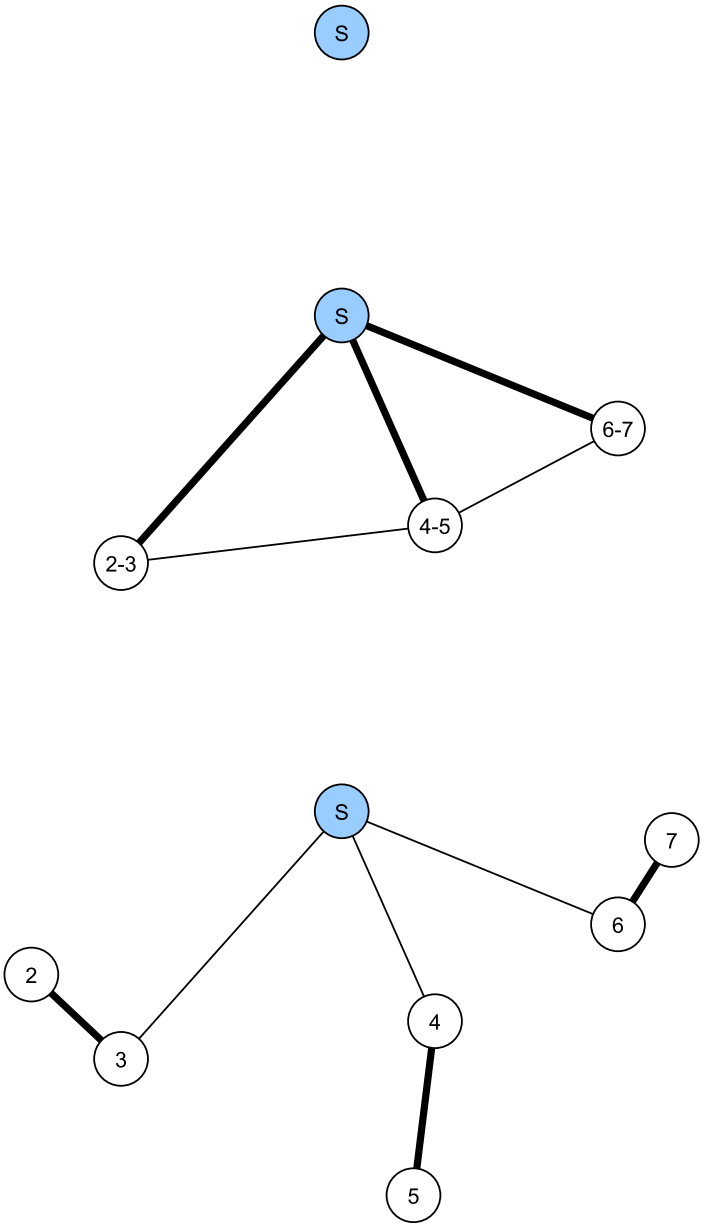

Abbildung 4.3: Das Refinement. Hier wird der Graph wiederherstellt und der Baum konstruiert.

Die Lösung ist hier immer ein Spannbaum. Dieser ist per Definition kreisfrei und besitzt $|V|-1$ Kanten. Jeder Knoten, mit Ausnahme der Wurzel, besitzt genau einen Vorgänger. Dies bedeutet, dass in dem Fall von parallelen Kanten, diese können bei dem Verschmelzen zweier Knoten entstehen, eine der Kante ausgewählt werden muss. Im Allgemeinen ist dies jene Kante, die zu einer günstigeren Lösung führt. Um Kreisfreiheit zu gewährleisten, muss während des Coarsenings darauf geachtet werden, dass niemals Kanten verwendet werden, welche Knoten verbinden die bereits direkt oder indirekt miteinander verbunden sind. Dies könnte beispielsweise erfolgen, indem man Knoten mit einer Komponenten-ID versieht, wobei 2 Knoten die miteinander verbunden sind die gleiche ID besitzen. Während des Refinements bzw. des Improvements während des Refinements, wo bereits eine Lösung vorliegt, muss lediglich darauf geachtet werden, dass jeder Knoten stets einen Pfad zum Wurzelknoten besitzt. Somit können unerwünschte Kreisbildungen verhindert werden.

4.4 Multilevel für das RDCMSTP

Das generelle Multilevel-Prinzip kann bei vielen Problemen angewandt werden, jedoch müssen sowohl Coarsening als auch Refinement entsprechend an das vorliegende Problem angepasst werden. Die Problemstellung der Diplomarbeit besteht nun darin, entsprechende Multilevel-Heuristiken für das Rooted Delay-Constrained Minimum Spanning Tree Problem zu entwickeln.

Die Eigenschaft des RDCMSTP, die auch die NP-Schwierigkeit des Problems verursacht, ist der Delay-Constraint. Eine Möglichkeit den Delay-Constraint in einer Multilevel-Heuristik zu behandeln, besteht darin, während des Coarsenings darauf zu achten, dass der Delay-Constraint auch in höheren Levels eingehalten werden kann. Das Verschmelzen zweier Knoten bzw. das Verwenden einer Kante soll also nicht verhindern, dass eine gültige Lösung gefunden wird. In Abbildung 4.4 ist ein sehr einfacher Graph mit lediglich drei Knoten abgebildet, für den ein minimaler Spannbaum zu finden ist, wobei auch hier wieder der erste Wert der Kantenbeschriftung den Kosten und der zweite dem Delay entspricht. Unter der Annahme, dass die Delaygrenze 5 beträgt, würde ein Hinzufügen der Kante $(2, 3)$ bzw. das Mergen dieser Knoten verhindern, dass eine gültige Lösung gefunden werden kann. Eine Multilevel-Heuristik für das RDCMSTP benötigt also eine Möglichkeit derartige Operationen auszuschließen. Dies kann erfolgen, indem spezielle Informationen über den Graph berechnet werden.

Die erste dieser Informationen sind die sogenannten Shortest-Delay-Paths. Die Shortest-Delay-Paths können für alle Knoten berechnet werden, indem ein Shortest-Path-Algorithmus verwendet wird. Hierzu wird der Dijkstra Algorithmus [3] verwendet, wobei hier die kürzesten Pfade bezogen auf die Delays berechnet werden. Durch die Berechnung der Shortest-Delay-Paths gewinnt man die Information, wie groß der Delay eines Knotens zum Sourceknoten mindestens sein muss. Dies ermöglicht gleichzeitig den Rückschluss, wie groß der Delay zwischen einem Knoten v und eventuellen Nachfolgeknoten maximal sein darf, um die Zulässigkeit der Lösung zu bewahren.

Abbildung 4.5 zeigt ein Beispiel eines Graphen mit Delaygrenze $B = 5$. Die Shortest-Delay-Paths wurden berechnet und sind als Funktion $d(v)$ in die Knoten eingetragen. Der Shortest-Delay-Path des Sourceknotens beträgt stets 0. $d(6) = 3$ bedeutet also, dass der kürzeste Delaypfad vom Knoten 6 zum Sourceknoten eine Länge von 3 besitzt. Dies bedeutet gleichzeitig, dass nur jene Knoten zu 6 verbunden werden dürfen, deren Delayentfernung nicht größer als 2 ist. Die Kanten $(3, 5)$ und $(4, 5)$ können in diesem Beispiel nicht verwendet werden, da sonst bei einer

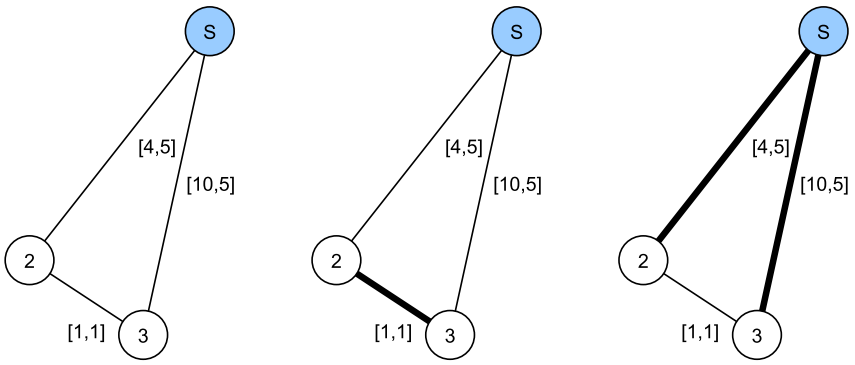

Abbildung 4.4: Durch Hinzufügen der Kante $(2, 3)$ kann bei einer Delaygrenze von 5 keine gültige Lösung mehr produziert werden.

Delaygrenze von 5 keine zulässige Lösung mehr produziert werden kann. Hierbei sei erwähnt, dass solche Kanten durch entsprechendes Preprocessing eliminiert werden können und hier nur zu Demonstrationszwecken im Graph enthalten sind.

In Abbildung 4.6 werden weitere Probleme des RDCMSTP verdeutlicht. Es genügt nicht nur die Shortest-Delay-Paths zu kennen. Eine Hinzunahme einer Kante verbindet nicht immer nur die aktuellen Knoten. Im Fall früherer Verschmelzungen ist diese Kante gleichzeitig Teil des Pfades zum Sourceknoten für die beteiligten Nachfolgerknoten. Das Hinzufügen von Kante $(2, 6)$ bzw. das Verschmelzen dieser Knoten wäre im ersten Level durchaus möglich. Wurde allerdings, wie in Abbildung 4.6 gezeigt, bereits eine Verschmelzung der Knoten 5 und 6 durchgeführt, würde das Anbinden des Knotens 6 zum Knoten 2 bei einer Delaygrenze von 5 bereits zu einer ungültigen Lösung führen.

Es ist also wichtig zu wissen, wie groß der Delay zwischen einem Knoten und seinen Nachfolgeknoten ist. Hierzu wird der Delay zwischen dem Knoten v und jenem Blattknoten mit dem höchsten Delaypfad zu v, also jenem Knoten, der am weitesten von dem Knoten v in Bezug auf den Delay entfernt ist, gespeichert. Dieses Delaymaximum oder Childdelay muss aktualisiert werden, sobald eine Verschmelzung durchgeführt wird. Es werden also pro Knoten zwei Delaywerte gespeichert, der Shortest-Delay-Path und der Childdelay. Im Fall des Knotens 6 aus Abbildung 4.6 beträgt die Länge des Shortest-Delay-Paths 3 und der Childdelay 2.

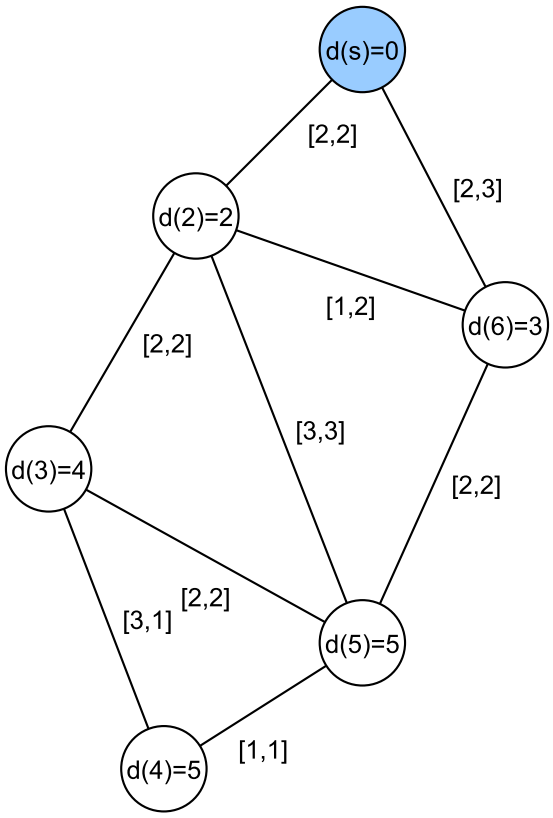

Abbildung 4.5: Ein Graph mit einer Delaygrenze B=5. Die Delaywerte $d(v)$ eines Knotens entsprechen der Länge des kürzesten Pfads zum Sourceknoten. Die Kante $(4, 5)$ kann nicht verwendet werden ohne den Delay-Constraint zu verletzen.

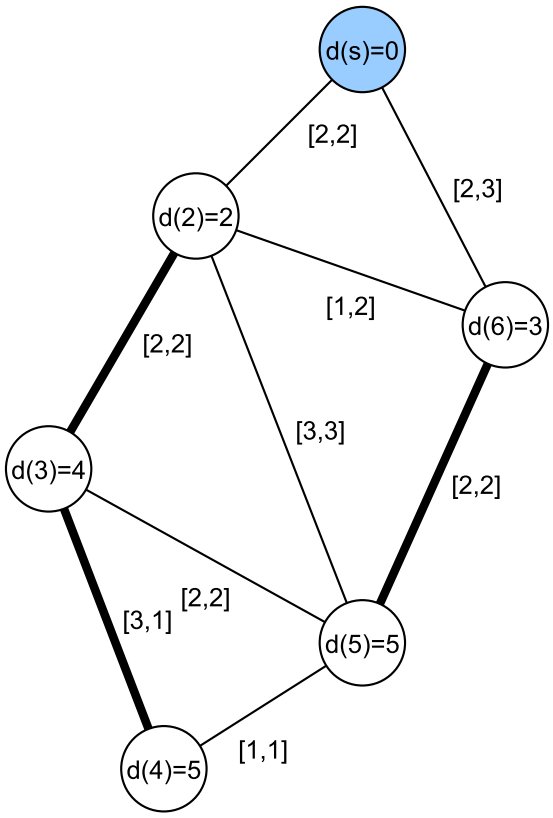

Abbildung 4.6: Ein Graph mit einer Delaygrenze B=5. Die Verwendung der Kante $(2, 6)$ ist nicht mehr zulässig, da der Childdelay von 6 bereits 2 beträgt. Eine Hinzunahme der Kante zu diesem Zeitpunkt würde eine zulässige Lösung verhindern.

Diese Informationen sind nicht nur für eine Multilevel-Heuristik nützlich. Auch andere Heuristiken, wie die Kruskal-basierte Heuristik [22], erfordern diese Informationen. Jedoch ist es aufgrund des Prinzips der Multilevel-Heuristik, der schrittweisen Verarbeitung des Graphen, oft notwendig, diese Informationen zu aktualisieren. Die grundsätzliche Vorgehensweise einer Multilevel-Heuristik für das RDCMSTP ist also den Graph systematisch zu vereinfachen und dabei die Delayinformation der einzelnen Knoten aktuell zu halten. Wie dies im speziellen geschieht, hängt natürlich von der Strategie der jeweiligen Multilevel-Heuristik ab.

Ein weiteres Problem von Multilevel-Heuristiken für das RDCMSTP ergibt sich aus der Vorgangsweise des Multilevel-Prinzips. Durch das Verschmelzen von Knoten bzw. durch das Verwenden der entsprechenden Kante soll sich die gesamte Knoten- und Kantenanzahl verringern, um so den Graph während des Coarsenings zu vereinfachen. Dies bedeutet aber gleichzeitig, dass im Zuge des Coarsenings zuvor durchgeführte Verschmelzungen während des weiteren Coarsenings nicht mehr rückgängig gemacht werden können. Mit anderen Worten wirken sich Entscheidungen in den ersten Levels des Coarsenings auf den gesamten Vorgang aus. Abbildung 4.7 zeigt ein Beispiel einer Kantenauswahl, die zu einer Lösung führt, welche schlechter als das Optimum ist. Dies ist ein Problem, welches für jede Heuristik besteht, auch für eine Multilevel-Heuristik. Oft können solche Fehlentscheidungen im Zuge des Refinements nicht mehr durch lokales Improvement behoben werden. Der kritische Teil einer Multilevel-Heuristik besteht also im Coarsening.

Es gibt kein Patentrezept für die Coarsening-Strategie. Man kann lediglich versuchen, durch entsprechende Überlegungen möglichst gutes Coarsening zu betreiben oder aber genug Raum für lokales Improvement zu lassen. In den folgenden Kapiteln werden drei Coarsening-Strategien vorgestellt, die verschiedene Ansätze und Überlegungen verfolgen, um möglichst gute Ergebnisse zu erhalten.

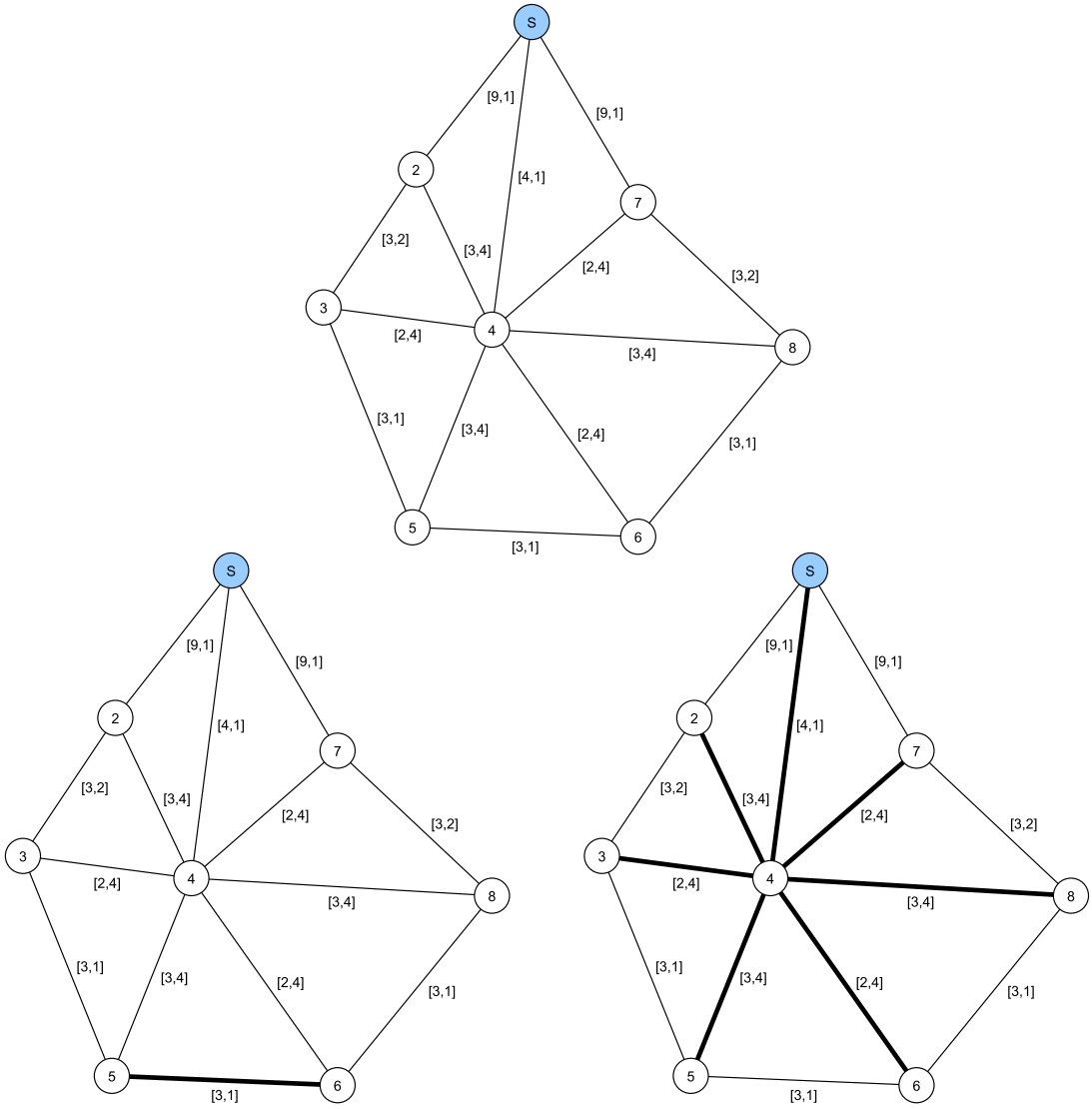

Abbildung 4.7: Ein Graph mit einer Delaygrenze B=5. Bereits das Hinzufügen der Kante $(5, 6)$ (links) verhindert die Optimallösung (rechts).

5 Knotengrad-basierte Multilevel Heuristik

Bei der ersten im Zuge der Diplomarbeit entwickelten Multilevel-Heuristik für das Rooted Delay-Constrained Minimum Spanning Tree Problem handelt es sich um die sogenannte Degree-based Multilevel-Heuristik. Bei dieser Heuristik werden in jedem Level des Coarsenings die Knoten nach ihrem Knotengrad, also der Anzahl der adjazenten Kanten, sortiert. Danach wird eine Anzahl von Knoten mit dem höchsten Knotengrad ausgewählt und zu sogenannten Superknoten befördert. Diese Superknoten sind nun Verbindungspunkte für alle anderen Knoten des Graphen. Für die Nicht-Superknoten wird nun versucht sie möglichst kostengünstig an einen der verfügbaren Superknoten anzubinden, ohne dabei den Delay-Constraint zu verletzen. Kann ein Nicht-Superknoten in einem Level nicht an einen Superknoten angebunden werden, tritt ein Reparaturvorgang in Kraft, der auf den Shortest-Delay-Paths basiert.

Nach Abschluss des Coarsenings wurde jedem Knoten ein Vorgänger zugewiesen. Es ist also möglich, den Lösungsbaum unmittelbar zu erstellen. Allerdings kann im Zuge des Refinements lokales Improvement, durch eine Nachbarschaft basierend auf Kantenaustausch, stattfinden. Im Folgenden wird die grundsätzliche Überlegung hinter der Degree-based Multilevel-Heuristik erläutert. Im Anschluss wird die Implementierung des Coarsening-Schritts vorgestellt. Refinement und das zugehörige Improvementverfahren finden sich in Kapitel 8 und die Testergebnisse in Kapitel 9.

Weiters wird eine einheitliche Notation eingeführt, welche für die Algorithmen in diesem und den nachfolgenden Kapiteln verwendet wird.

$pred(v)$ ... der dem Knoten v zugewiesene Vorgänger

$sdp(v)$ ... der Delay des Shortest-Delay-Paths zwischen dem Sourceknoten s und dem Knoten v

$sdp\_pred(v)$ ... der Vorgänger des Knotens v im Shortest-Delay-Path

$child\_d(v)$ ... der Delay des Pfades mit dem größten Delay zwischen dem Knoten v und einem direkten oder indirekten Nachfolger

$source\_d(v)$ ... der Delay des Pfades zwischen dem Sourceknoten s und dem Knoten v im aktuellen Lösungsbaum

$degree(v)$ ... der Knotengrad des Knoten v

$score(v)$ ... der Rankingscore des Knoten v, siehe Kapitel 6

$v\_status(v)$ ... gibt an ob es sich bei dem Knoten v um einen Major- oder Minorknoten handelt, siehe Kapitel 7

$component(v)$ ... die Komponente in der sich der Knoten v befindet

$componentdelay(v)$ ... der Delay des Pfades zwischen dem Knoten v und dem Majorknoten der Komponente von v

$score((u, v))$ ... der Rankingscore der Kante (u, v), siehe Kapitel 6

5.1 Grundlegende Überlegungen

Die erste Überlegung zu einer Multilevel-Heuristik für das Rooted Delay-Constraint Minimum Spanning Tree Problem bestand darin Level für Level Unterbäume aufzubauen und diese anschließend im Verlauf des Coarsenings zu verbinden. Es sollen also Knoten ausgewählt werden, die für dieses Level als Anschlussknoten fungieren. Diese Knoten werden im weiteren Verlauf der Arbeit Superknoten genannt. Durch das Verbinden der übrigen Knoten zu diesen Superknoten entstehen Bäume, wobei der Superknoten hier die Wurzel dieser Unterbäume darstellt. Diese Unterbäume werden im nächsthöheren Level durch ihre Wurzelknoten, also die Superknoten des vorhergehenden Levels, repräsentiert. Abbildung 5.1 zeigt diesen Vorgang, wobei hier auf Kantenbeschriftungen verzichtet wird. Die Knoten s, 3, 5, 8, 12 und 13 werden als Superknoten ausgewählt. Alle verbleibenden Knoten werden zum nächstgelegenen Superknoten verbunden. Im nächsten Level bleibt der reduzierte Graph übrig, bestehend aus den Superknoten des vorhergehenden Levels. Man beachte, dass der Sourceknoten immer ein Superknoten ist.

Weiterhin ist zu beachten, dass es möglich sein muss, diese Unterbäume im weiteren Verlauf des Coarsenings zum Sourceknoten verbinden zu können. Der maximale Delay bzw. der Childdelay eines Unterbaumes wird als Knotenattribut des zugehörigen Superknotens festgehalten. Es ist stets zu gewährleisten, dass der Shortest-Delay-Path des Superknotens plus der dazugehörige Childdelay die Delaygrenze B nicht überschreitet.

5.2 Ermitteln der Superknoten

Die Frage, die sich stellt, ist nun, wie viele und vor allem welche Knoten als Superknoten eines Levels ausgewählt werden sollen. Sowohl die Anzahl als auch die Auswahl der Knoten spielen eine wichtige Rolle im Verlauf des Coarsenings.

5.2.1 Anzahl der Superknoten

Die Anzahl der verwendeten Superknoten bestimmt prinzipiell die Anzahl der notwendigen Coarsening-Schritte. Werden beispielsweise sehr wenige Superknoten ausgewählt, wird die Knotenanzahl des Graphen sehr rasch reduziert. Dies hat jedoch den Nachteil, dass weniger Möglichkeiten in Betracht gezogen werden, da sämtliche Kanten, die zwischen zwei Nicht-Superknoten verlaufen, nicht berücksichtigt werden. Ein weiteres Problem kann bei einer relativ strikt gewählten Delaygrenze entstehen, da so nur wenige gültige Lösungen existieren. Stehen jene Knoten, die für eine gültige Lösung als Superknoten benötigt, werden nicht zur Auswahl, muss eine alternative Auswahl getroffen werden, siehe Kapitel 5.3.2.

Allerdings entstehen auch durch die Wahl sehr vieler Superknoten Probleme. Eine hohe Anzahl

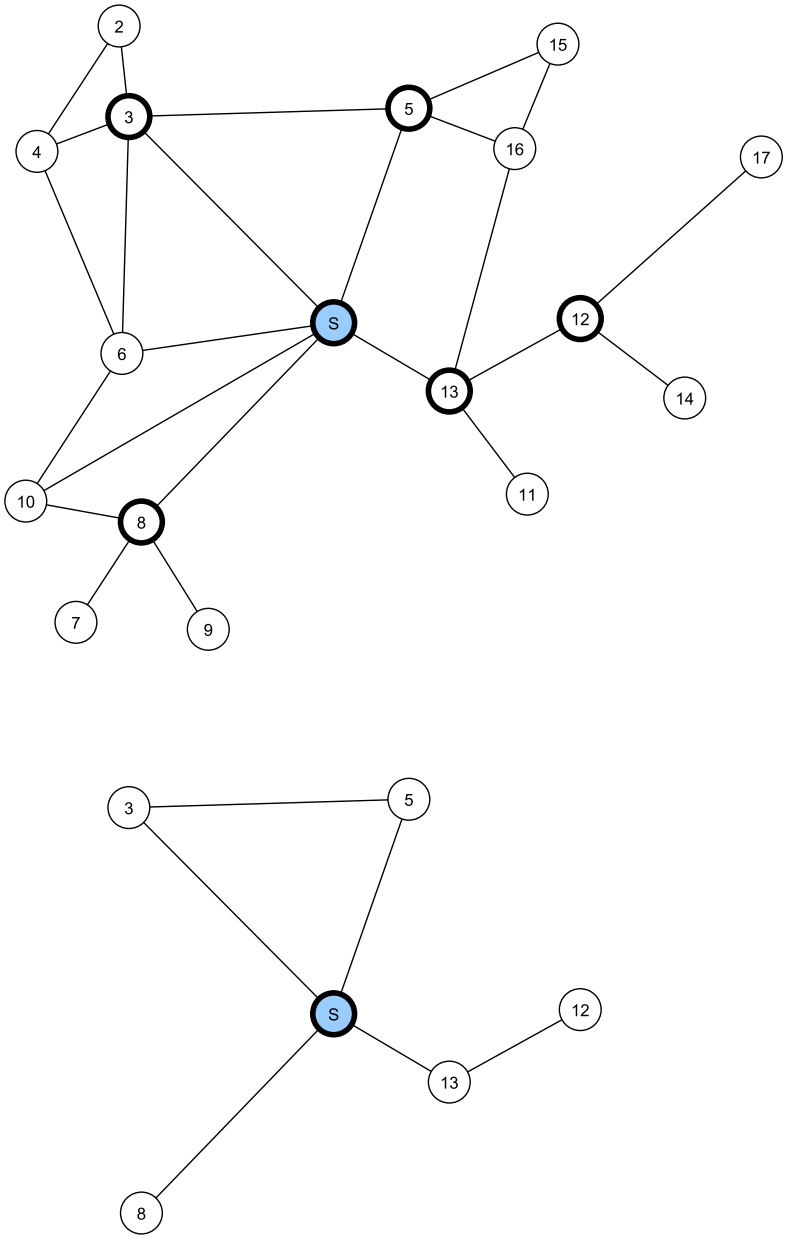

Abbildung 5.1: Die Superknoten werden durch die dicke Umrandung dargestellt. Alle übrigen Knoten werden zu diesen verbunden und sind im nächsten Level des Coarsening-Schritts nicht mehr von Interesse.

an Superknoten führt zu einem vergleichsweise langsamen Coarsening. Da die niedrigeren Levels sehr aufwändig sind, da viele Kanten berücksichtigt werden müssen, wirkt sich ein zu langsames Coarsening schlaghaltig auf die Effizienz der gesamten Heuristik aus. Es ist also wichtig, eine geeignete Anzahl an Superknoten pro Level zu finden, wobei diese stark von der Probleminstanz und den gegebenen Umständen abhängig ist. Auf das Problem der Superknotenanzahl gibt es keine eindeutige Antwort, somit wird dem Benutzer der Heuristik überlassen, diese durch einen Parameter festzulegen bzw. damit zu experimentieren. Bei diesem Parameter handelt es sich um einen Prozentwert, der angibt, wieviele der Knoten in jedem Level als Superknoten ausgewählt werden.

5.2.2 Wahl der Superknoten

Ähnlich schwierig gestaltet sich die Auswahl geeigneter Superknoten. Da nur Kanten, die zwischen Superknoten und Nicht-Superknoten verlaufen, im weiteren Verlauf des Coarsenings berücksichtigt werden, ist die Wahl der Superknoten entscheidend. Eventuelle Fehlentscheidungen bei der Superknotenwahl können sich darin äußern, dass Kanten, die sowohl niedrige Kosten als auch niedrigen Delay aufweisen, nicht verwendet werden. Auch kann es bei nicht-vollständigen Graphen vorkommen, dass die Superknoten derart gewählt werden, dass in jedem Coarsening-Schritt sehr wenige Kanten überprüft werden. Dies geschieht, wenn Knoten mit einem sehr niedrigen Knotengrad ausgewählt werden.

Die Überlegung zur Wahl der Superknoten dieser Heuristik besteht darin, Knoten mit möglichst hohem Knotengrad auszuwählen. Dies hat mehrere Vorteile. Zum einen bringt die Verwendung von Knoten mit hohem Knotengrad im Allgemeinen mehr Möglichkeiten für den aktuellen Coarsening-Schritt. Es stehen in Summe mehr Kanten zur Verfügung, die im Verlauf des Coarsenings verwendet werden können. Im Fall einer Kante, die zwischen einem Super- und einem Nicht-Superknoten verläuft, erhöht sich die Anzahl der Verbindungsmöglichkeiten für den Nicht-Superknoten in dem aktuellen Level. Im Fall einer Kante, die zwischen zwei Superknoten verläuft, steht diese im nächsthöheren Level zur Verfügung. Dies erhöht im Allgemeinen die Qualität der im Zuge des Coarsenings produzierten Lösung.

Ein weiterer Vorteil der Auswahl von Knoten mit hohem Knotengrad liegt in der Struktur der entstehenden Lösung. Im allgemeinen Fall ist es besser einen Baum zu konstruieren, der eine eher geringe Tiefe, also Pfade bestehend aus eher weniger Kanten, besitzt. Kürzere Pfade bedeuten oft auch kürzeren Delay. Dies wiederrum hält mehr Möglichkeiten für höhere Level offen bzw. bietet mehr Möglichkeiten für lokales Improvement in der Refinementphase.

Natürlich gibt es Ausnahmen, in denen sich diese Überlegung negativ auswirkt. Im praktischen Fall ist es jedoch oft so, dass sich der Delay euklidisch verhält. Dies bedeutet, dass eine direkte Verbindung zweier Knoten einen geringeren Delay aufweist als ein Umweg über einen Drittknoten. Ein kürzerer Delay bringt indirekt eine Verbesserung durch mehr Möglichkeiten im späteren Verlauf der Heuristik, beispielsweise während eines lokalen Improvements im Refinement.

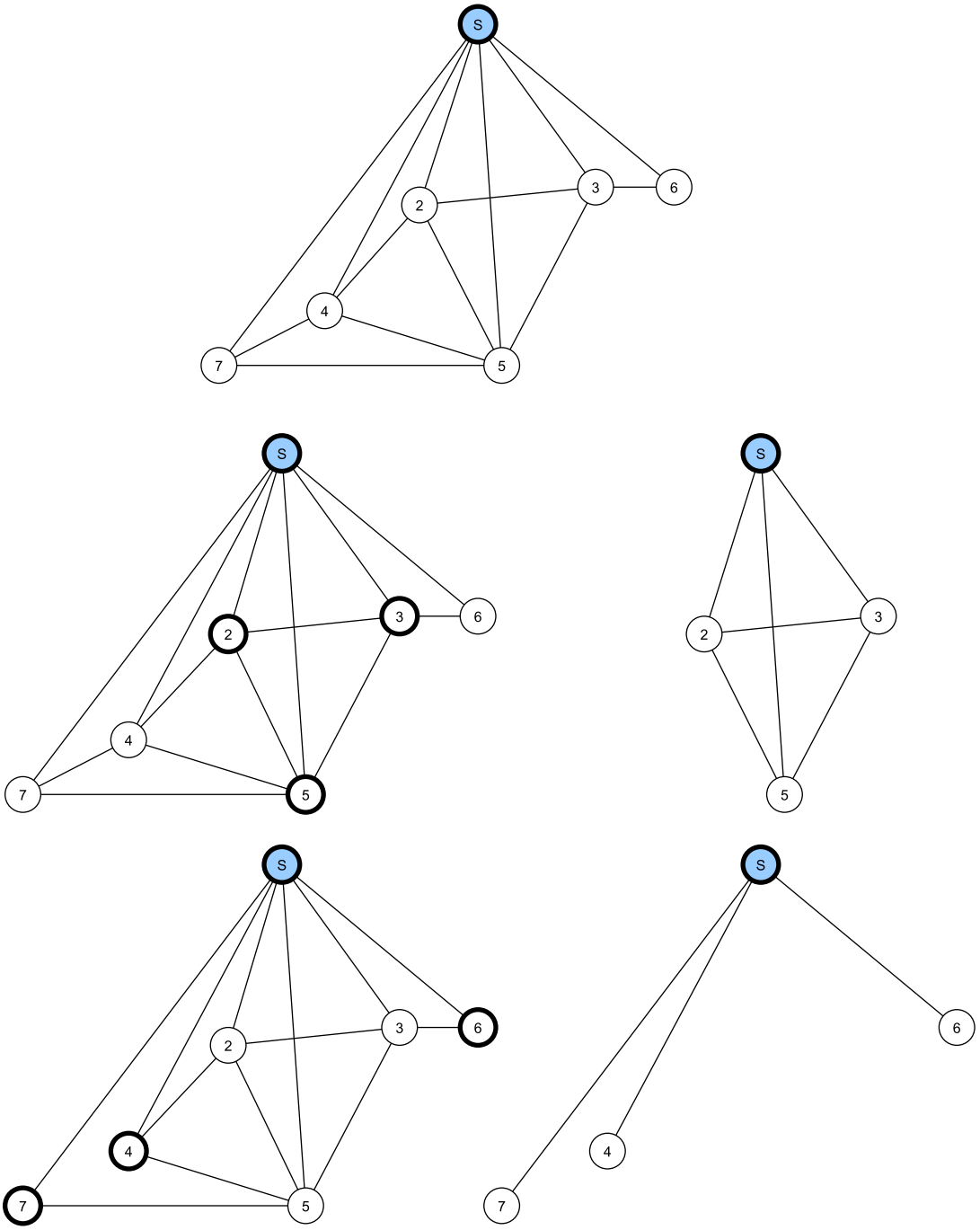

Abbildung 5.2: Hier wird der Unterschied zweier Selektionen von Superknoten verdeutlicht.

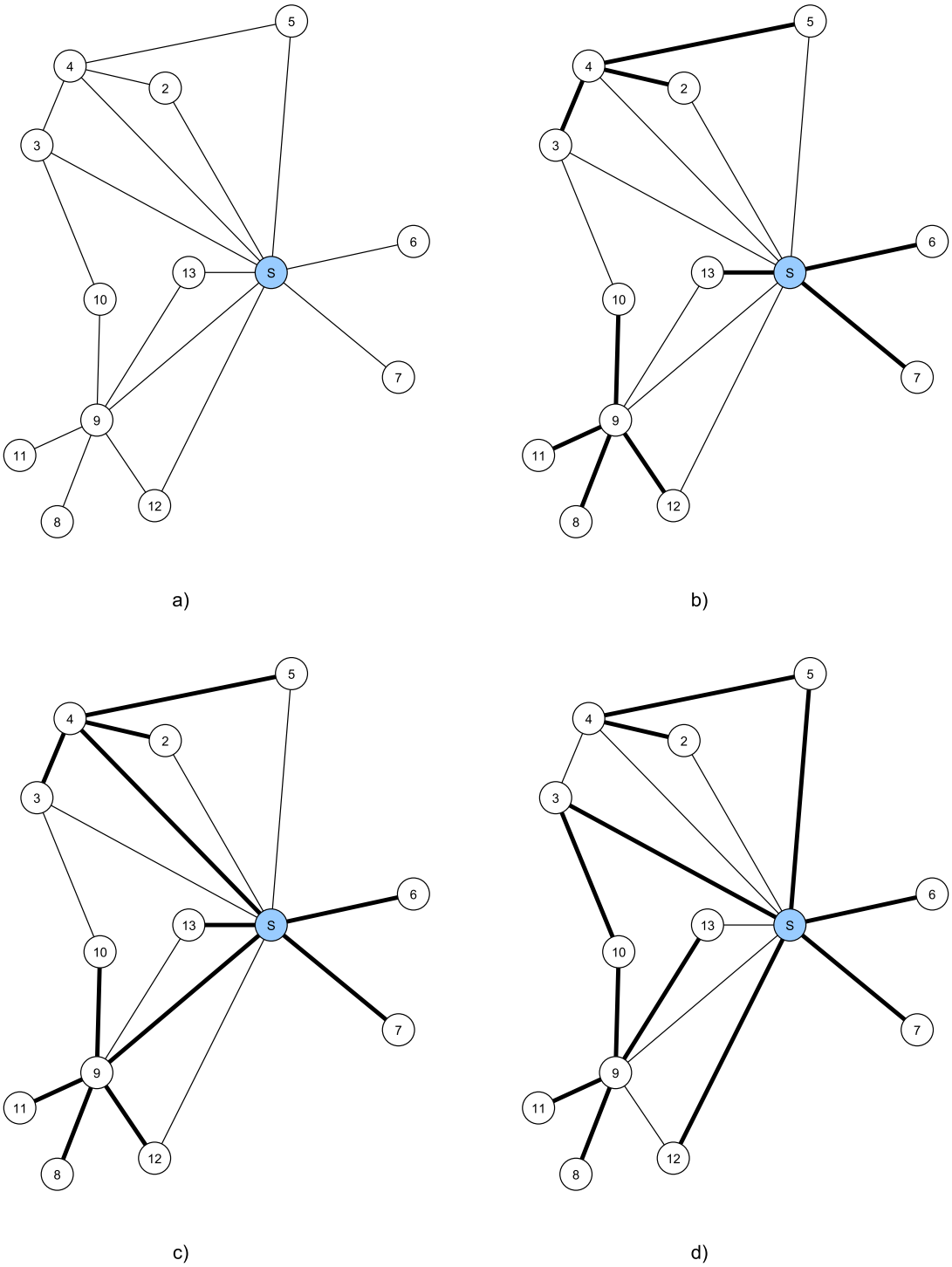

Abbildung 5.3: Aus dem Originalgraph a) wird durch Auswahl der Superknoten in b) im weiteren Verlauf des Coarsenings der Baum c) erzeugt. Dieser wird in den meisten Fällen besser als ein Baum wie in d) ersichtlich sein.

Es ist natürlich möglich, im Fall von vollständigen Graphen sogar sehr wahrscheinlich, dass zwei Knoten den gleichen Knotengrad besitzen. Da die Anzahl der auszuwählenden Superknoten durch die Anzahl der Knoten und den entsprechenden Parameter begrenzt ist, ist bekannt, wieviele Superknoten für das aktuelle Level benötigt werden. Nachdem die Knoten nach ihrem Knotengrad absteigend sortiert wurden, wird die benötigte Anzahl an Superknoten aus der gesamten Knotenmenge des aktuellen Levels entnommen. Im Fall von mehreren Knoten mit gleichem Knotengrad wird eine zufällige Auswahl dieser Knoten als Superknoten verwendet. Hierzu wird der Waterman Algorithmus, siehe [6], verwendet.

5.2.3 Algorithmus zur Wahl der Superknoten

Der vollständige Algorithmus zur Knotenauswahl findet sich in Algorithmus 3. Man beachte, dass dieser Algorithmus auch von der in Kapitel 6 vorgestellten Multilevel-Heuristik verwendet wird, weshalb es einen Parameter *Auswahlkriterium* gibt. Für die Degree-based Multilevel-Heuristik wird der Parameter *Auswahlkriterium* auf *degree* gesetzt. Vorab wird ermittelt, wieviele Superknoten gesucht werden und die vorhandenen Knoten nach ihrem Knotengrad sortiert. Hierzu wird eine sogenannte Priority-Queue eingesetzt, wobei im Pseudocode eine Liste verwendet wird um die Lesbarkeit zu erhöhen. Dabei handelt es sich um eine Datenstruktur, in der die Elemente nach einer Priorität sortiert werden, wobei die Priorität hier durch den Knotengrad gegeben ist. Die Knoten werden nun absteigend aus dieser Priority-Queue entfernt. Solange es sich um Knoten mit gleichem Knotengrad handelt, werden diese in einer Kandidatenliste gespeichert. Sobald ein Knoten mit niedrigerem Knotengrad erreicht wurde, wird die Kandidatenliste überprüft. Übersteigt die Anzahl der Kandidaten die Anzahl von noch gesuchten Superknoten nicht, werden die entsprechenden Knoten schlicht als Superknoten übernommen.

Tritt allerdings der andere Fall ein, so sind mehr Superknotenkandidaten als gesuchte Superknoten vorhanden. Es muss also eine Auswahl aus den bestehenden Kandidaten getroffen werden. Hier kommt der Waterman Algorithmus [6] zur Anwendung. Dieser garantiert, dass alle Kandidaten die exakt gleiche Wahrscheinlichkeit haben, in die Superknotenliste aufgenommen zu werden. Die Worst-Case-Laufzeit der Knotenauswahl beträgt $\mathcal{O}(|V| \cdot \log(|V|))$, wobei $|V|$ die Knotenanzahl angibt. Der Aufwand begründet sich durch den Sortiervorgang in der verwendeten Datenstruktur.

5.3 Coarseningphase

Nach Auswahl der Superknoten werden die noch vorhandenen Nicht-Superknoten zu diesen verbunden. Hierzu werden die Kanten des aktuellen Levels durchlaufen und jene Kanten, die zwischen Superknoten und Nicht-Superknoten verlaufen, auf ihre Möglichkeit in Bezug auf den Delay geprüft. Im weiteren Verlauf wird die jeweils kostengünstigste Möglichkeit der Anbindung eines Nicht-Superknotens an einen entsprechenden Superknoten genutzt. Auf eventuelle Problemknoten, die nicht zu einem Superknoten verbunden werden können, wird eine auf Shortest-Delay-Paths basierende Reparaturmethode angewandt. Im Laufe des Coarsenings wird jedem Knoten außer dem Sourceknoten ein Vorgänger zugewiesen. Prinzipiell entspricht dies bereits einer gültigen Lösung, die später im Refinementschritt verbessert werden kann.

Algorithmus 3 : Knotenauswahl

Input : Knotenliste $L\_aktiveKnoten$, $Auswahlkriterium$
Output : Knotenliste $L\_Superknoten$

```
1  Knotenliste L_aktiveKnoten, L_Kandidaten, L_Superknoten
2  act = L_aktiveKnoten.first()
3  gesuchte_Superknotenanzahl = |L_aktiveKnoten| * Superrate
4  if Auswahlkriterium == degree then
5      Sortiere L_aktiveKnoten absteigend nach ihrem Knotengrad
6  else if Auswahlkriterium == score then
7      Sortiere L_aktiveKnoten absteigend nach ihrem Score
8  L_aktiveKnoten = L_aktiveKnoten \ {s}
9  L_Superknoten = L_Superknoten ∪ {s}
10 while |L_Superknoten| < gesuchte_Superknotenanzahl do
11     if Auswahlkriterium == degree then
12         max = degree(act)
13         while degree(act) == max do
14             L_Kandidaten = L_Kandidaten ∪ {act}
15             act = L_aktiveKnoten.next()
16     else if Auswahlkriterium == score then
17         max = score(act)
18         while score(act) == max do
19             L_Kandidaten = L_Kandidaten ∪ {act}
20             act = L_aktiveKnoten.next()
21     if |L_Superknoten| + |L_Kandidaten| ≤ gesuchte_Superknotenanzahl then
22         L_Superknoten = L_Superknoten ∪ L_Kandidaten
23         L_Kandidaten.clear()
24     else
25         // Waterman Algorithmus
26         Menge neu ausgewählter Superknoten S = ∅
27         t = 0
28         m = gesuchte_Superknotenanzahl − |L_Superknoten|
29         forall v ∈ L_Kandidaten do
30             if |S| < m then
31                 S ← S ∪ {v}
32             else
33                 Mit Wahrscheinlichkeit P = m/(t+1) ermittle zufälliges Element z ∈ S
34                 S ← S \ {z} ∪ {v}
35             t = t + 1
36         L_Superknoten = L_Superknoten ∪ {S}
```

5.3.1 Anbinden der Knoten

Nachdem die Superknoten ermittelt wurden, wird für die verbleibenden Nicht-Superknoten ein Vorgängerknoten ermittelt, um so die Baumstruktur entstehen zu lassen. Dies geschieht, indem alle Kanten des aktuellen Levels, die zwischen einem Superknoten und einem Nicht-Superknoten verlaufen, überprüft werden. Hierbei muss beachtet werden, dass keine Zuweisung eines Vorgängerknotens erfolgen darf, welche die Zulässigkeit der Lösung verhindert. Die bereits ermittelten kürzesten Wege in Bezug auf die Delays zwischen den Knoten und dem Sourceknoten, die Shortest-Delay-Paths, sind zu jedem Zeitpunkt auch für die Superknoten vorhanden. In höheren Levels wird es auch vorkommen, dass der anzubindende Nicht-Superknoten in früheren Levels als Superknoten fungierte und somit bereits einen Unterbaum beinhaltet. Daher muss auch der Childdelay des Nicht-Superknotens berücksichtigt werden. Als dritte Größe muss natürlich auch der Delay der in Frage kommenden Kante berücksichtigt werden. Die Kante (u, v) zwischen einem Superknoten u und einem Nicht-Superknoten v kann also nur akzeptiert werden, wenn gilt:

$$sdp(u) + d((u, v)) + child\_d(v) \leq B \tag{5.1}$$

Abbildung 5.4 zeigt den Ablauf des Verbindens bzw. Anbindens der Knoten. Die Knoten s, 2, 6 und 8 wurden als Superknoten ausgewählt. Die strichlierten Linien zeigen jene Kanten, die zwischen Super- und Nicht-Superknoten verlaufen und somit überprüft werden müssen. Nun werden die kostengünstigsten Kanten ausgewählt, die keine Verletzung des Delay-Constraints mit sich bringen. Die Kante $(7, 8)$ wäre beispielsweise günstiger als die Kante $(6, 7)$, kann aber aufgrund der Delaygrenze B=5 nicht verwendet werden. Im nächsten Schritt sind nur noch die Superknoten des vorhergehenden und die dazwischen verlaufenden Kanten übrig. Zusätzlich sind die Knoten nun mit entsprechenden Childdelays (d) behaftet.

Durch die Einschränkung, dass nur Kanten zwischen Superknoten und Nicht-Superknoten berücksichtigt werden, werden viele Möglichkeiten außer Acht gelassen. Mögliche Fehler können jedoch eventuell im Zuge des Improvements während des Refinements behoben werden. Ein weiteres Problem ist, dass der Shortest-Delay-Path eines Knotens nicht zwangsläufig aus Superknoten des momentanen Levels besteht. Dieses Problem wird im Abschnitt 5.3.2 erläutert.

5.3.2 Gewährleisten einer gültigen Lösung

Die Shortest-Delay-Paths und damit auch die Bedingung aus Formel 5.1 stellen sicher, dass es eine Möglichkeit gibt einen Knoten zum Sourceknoten zu verbinden. Es ist jedoch nicht gegeben, dass diese Shortest-Delay-Paths über Superknoten des momentanen Levels verlaufen. Speziell bei einem sehr strengen Delay-Constraint, der nur wenige gültige Lösungen zulässt, kann es vorkommen, dass es keine gültige Verbindung zu einem bestehenden Superknoten gibt.

Für derartige Problemknoten muss eine entsprechende Anpassung während des Coarsenings vorgenommen werden um eine gültige Lösung zu garantieren. Da bereits ein möglicher Pfad, eben der Shortest-Delay-Path, bekannt ist, kann dieser als eine Art Ersatzlösung verwendet werden. Hierzu wird ausgehend von dem Problemknoten der Shortest-Delay-Path verfolgt. Der unmittelbar nachfolgende Knoten wird rückwirkend zum Superknoten erklärt und der Problemknoten bekommt diesen als Vorgänger zugewiesen. Der Childdelay des neuen Superknotens wird berechnet und er wird wie auch die anderen Superknoten in das nächste Level übernommen. Es wird also prinzipiell eine rückwirkende Anpassung vorgenommen.

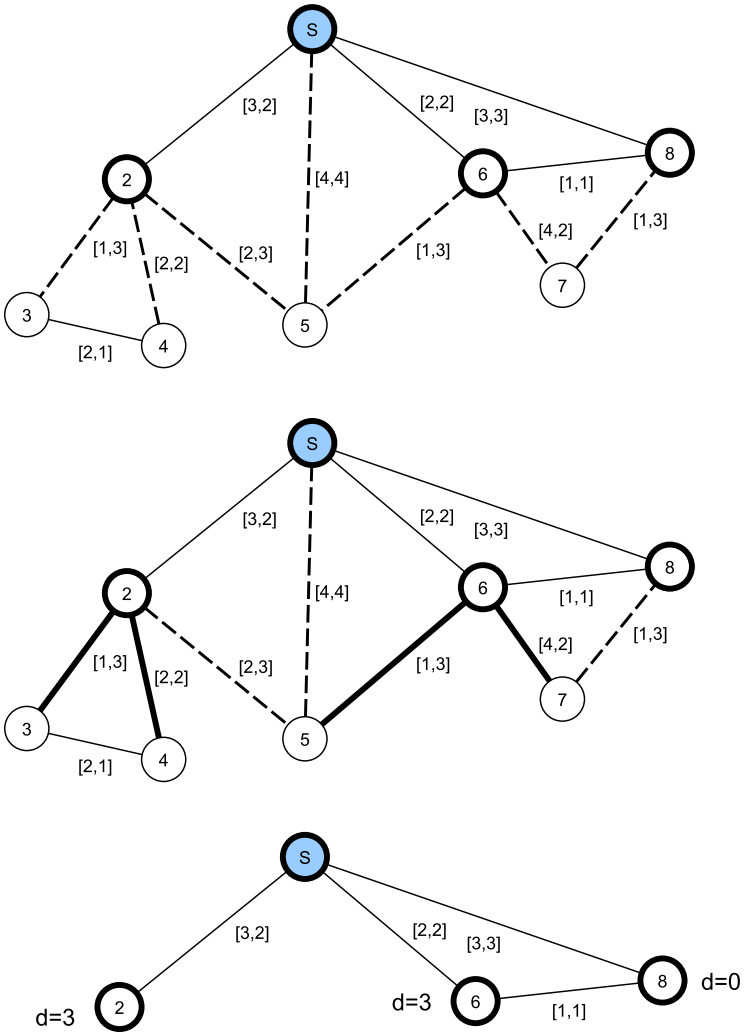

Abbildung 5.4: Ein Graph mit Delaygrenze B=5. Die strichlierten Linien zeigen die überprüften Kanten. Die kostengünstigsten möglichen Kanten werden ausgewählt. Im nächsten Schritt sind nur noch die Superknoten des vorherigen Schritts und die dazwischen verlaufenden Kanten von Interesse. Die Childdelays d der Knoten werden hier ebenfalls abgebildet.

In Abbildung 5.5 und 5.6 sind die möglichen Szenarien illustriert, die zu derartigen Problemknoten führen können. Das erste Szenario wird in 5.5 gezeigt. Hier kann der Knoten 4 an keinen der ursprünglich gewählten Superknoten angebunden werden. Es wird also der Shortest-Delay-Path ausgehend von dem Knoten 4 verfolgt. Der unmittelbar folgende Knoten ist der Knoten 5. Nun wird Knoten 5 rückwirkend zum Superknoten befördert und wird gleichzeitig der Vorgängerknoten von Knoten 4. Somit gehört die Kante $(4, 5)$ zur Teillösung. Gleichzeitig wird durch den Superknotenstatus von 5 die momentane Anbindung dieses Knotens entfernt. Die Verbindung zum Knoten 3 wird gelöst und die Kante $(3, 5)$ gehört nicht mehr zur Teillösung.

Dieses Problem entsteht in den ersten Levels des Graphen eher selten, da die Knoten noch nicht mit allzu hohen Childdelays behaftet sind. Im Laufe des Anbindens der Knoten werden die Superknoten der früheren Levels jedoch wachsende Childdelays aufweisen. Im Allgemeinen ist es schwieriger Knoten mit hohen Childdelays entsprechend zu verbinden. Im Zuge des Coarsenings werden jedoch jene Kanten den Vorzug bekommen, die kostengünstig sind. In Bezug auf den Delay wird lediglich die Möglichkeit der weiteren Anbindung ohne Delay-Constraint Verletzung vorausgesetzt. Dies kann schnell dazu führen, dass der Delay schon früh ausgenutzt wird.

Abbildung 5.6 setzt den Coarseningvorgang der Abbildung 5.5 fort. In 5.5 wurde die Kante $(6, 7)$ verwendet, was zu einem hohen Childdelay für den Knoten 6 führt. Dieser kann nun nicht mehr über die bestehenden Superknoten angebunden werden, also wird auch hier der Shortest-Delay-Path verfolgt. Dieser verläuft jedoch über den Knoten 8, womit dieser zuvor schon abgehandelte Knoten rückwirkend zum Superknoten erklärt wird. Auch hier wird der Knoten 8 zum Vorgänger von Knoten 6 erklärt. In diesem speziellen Fall ändert sich die Kantenmenge der Teillösung nicht, die Struktur für das weitere Coarsening jedoch sehr wohl. Im Weiteren wird der Childdelay des Knotens 8 ermittelt. Dieser ergibt sich aus dem Childdelay des Knotens 6, der weiterhin als Vorgänger anderer Knoten fungieren kann und dem Delay der Kante. Es ist jedoch zu beachten, dass der Childdelay des Knotens 6 neu ermittelt werden muss, um sicher zu gehen, dass dieser auch korrekt ist. Ähnlich gestaltet sich die Problematik mit Knoten 5. Da hier keine gültige Kante zu einem Superknoten vorliegt, wird der Knoten 3 rückwirkend zum Superknoten.

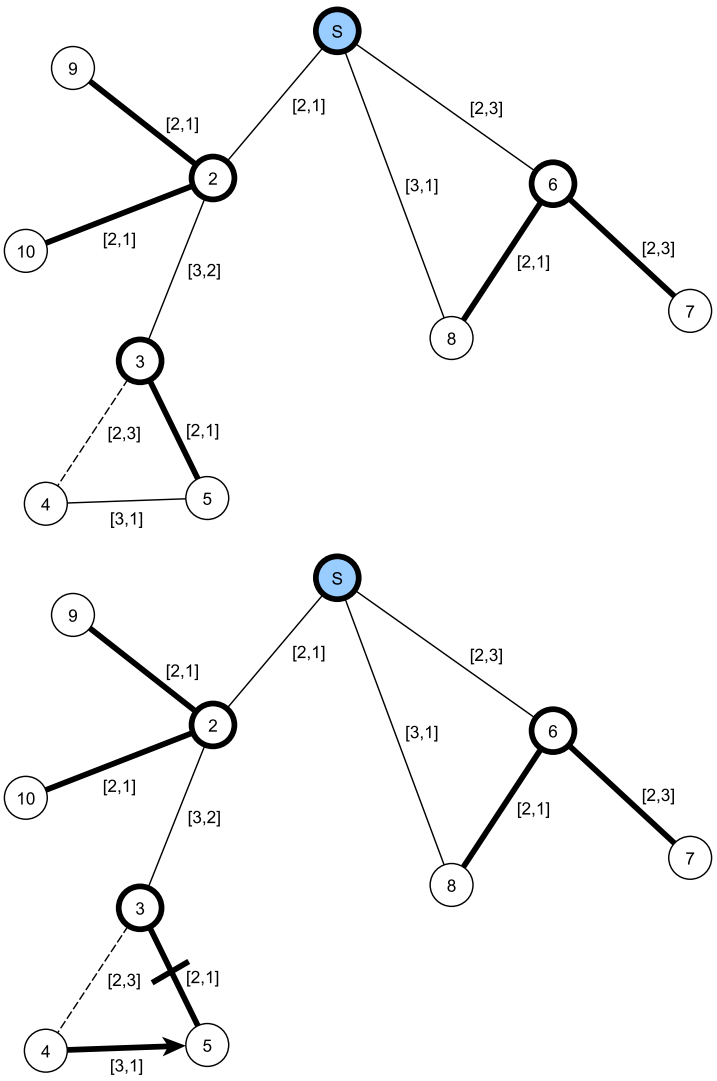

Abbildung 5.5: Ein Graph mit Delaygrenze $B = 5$. Der Knoten 4 kann zu keinem Superknoten verbunden werden. Der auf dem Shortest-Delay-Path nächstliegende Knoten wird zu einem Superknoten befördert.

5.3.3 Der Coarsening Algorithmus

Algorithmus 4 : Degree-based Multilevel Heuristik - Coarsening

Input : Graph
Output : Vorgänger für alle Knoten (gültige Lösung)

Kantenliste $L\_aktiveKanten$, $L\_inaktiveKanten$
Knotenliste $L\_aktiveKnoten$, $L\_Superknoten$
$L\_aktiveKnoten = V$
$L\_aktiveKanten = E$
Sortiere $L\_aktiveKanten$ aufsteigend nach Kosten, sekundär nach Delay.
forall $v \in V$ **do**
    $pred(v) = v$
$level = 0$
while $\exists v \in V \setminus \{s\} \mid pred(v) = v$ **do**
    $level = level + 1$
    $L\_Superknoten = Knotenauswahl(L\_aktiveKnoten, degree)$
    $L\_aktiveKnoten = L\_aktiveKnoten \setminus L\_Superknoten$
    forall $(u, v) \in L\_aktiveKanten$ **do**
        if $v \in L\_aktiveKnoten \wedge u \in L\_Superknoten$ **then**
            if *sdp(u) + d((u,v)) + child_d(v)* $\leq B$ **then**
                $pred(v) = u$
                if *child_d(u) < child_d(v) + d((u,v))* **then**
                    $child\_d(u) = child\_d(v) + d((u, v))$
                $L\_aktiveKnoten = L\_aktiveKnoten \setminus \{v\}$
                $degree(u) = degree(u) - 1$
        else if $u \notin L\_Superknoten \wedge v \notin L\_Superknoten$ **then**
            $L\_aktiveKanten = L\_aktiveKanten \setminus \{(u, v)\}$
            $L\_inaktiveKanten = L\_inaktiveKanten \cup \{(u, v)\}$
    forall $v \in L\_aktiveKnoten$ **do**
        $u = sdp\_pred(v)$
        $L\_Superknoten = L\_Superknoten \cup \{u\}$
        $L\_aktiveKnoten = L\_aktiveKnoten \setminus \{u\}$
        $pred(v) = u$
        if *child_d(u) < child_d(v) + d((u,v))* **then**
            $child\_d(u) = child\_d(v) + d(u, v)$
        $oldpred = pred(u)$
        $pred(u) = u$
        if $child\_d(oldpred) == child\_d(u) + d((u, oldpred))$ **then**
            $child\_d(oldpred) = \max\{child\_d(w) + d((oldpred, w) \mid pred(w) = oldpred\}$
        forall $(u, w) \mid w \in L\_Superknoten \wedge (u, w) \in L\_inaktiveKanten$ **do**
            $L\_aktiveKanten = L\_aktiveKanten \cup \{(u, w)\}$
            $L\_inaktiveKanten = L\_inaktiveKanten \setminus \{(u, w)\}$
        $L\_aktiveKnoten = L\_aktiveKnoten \setminus \{v\}$
    $L\_aktiveKnoten = L\_Superknoten$

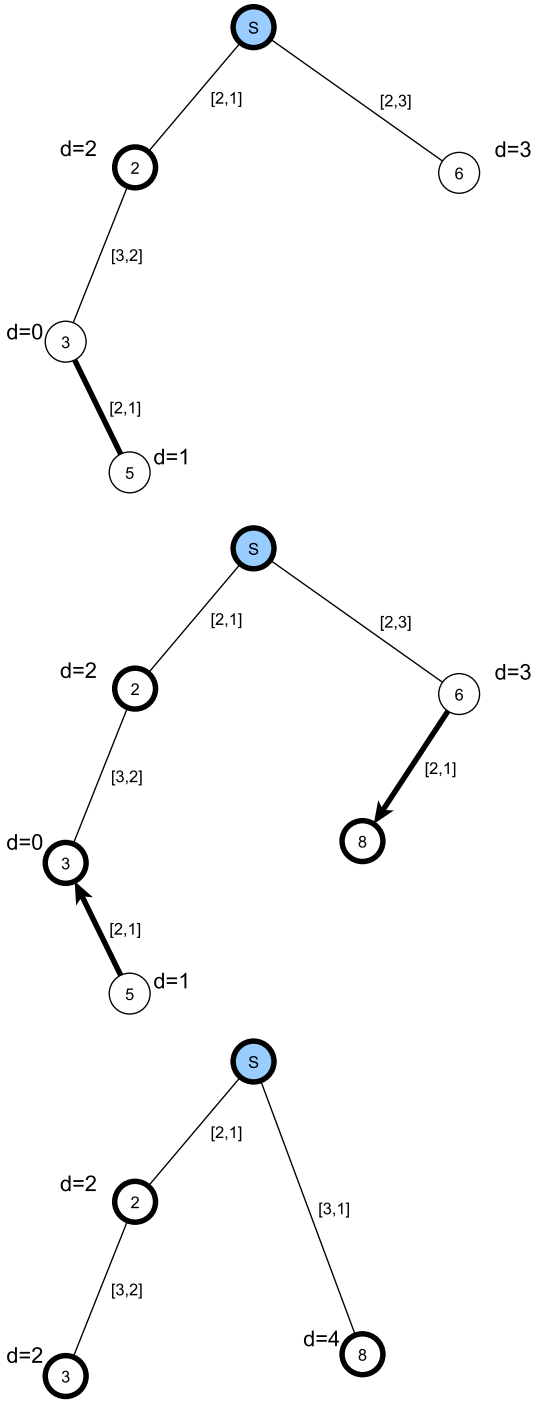

Abbildung 5.6: Der weitere Verlauf des Coarsenings. Der Knoten 6 kann nicht verbunden werden. Der Knoten 8 wird nachträglich zum Superknoten gemacht.

Algorithmus 4 beschreibt den Coarsening Teil der Degree-based Multilevel-Heuristik. Die Lösung wird durch eine Zuweisung von Vorgängerknoten zu jedem Knoten repräsentiert. Eine Lösung gilt als vollständig, sobald alle Knoten mit Ausnahme des Sourceknoten einen Vorgänger zugewiesen bekommen haben. Die Kanten zwischen den Knoten und ihren Vorgängern entsprechen jenen Kanten, die der Spannbaum beinhaltet. Anfänglich besteht jeder Vorgänger eines Knotens aus dem Knoten selbst und alle Knoten sind als aktiv klassifiziert. Auch alle Kanten des Graphen sind initial aktiv. Weiters werden die Kanten aufsteigend nach ihren Kosten und sekundär nach ihren Delays sortiert.

Solange keine vollständige Lösung konstruiert wurde, wird die Lösung kontinuierlich erweitert, indem eine while-Schleife durchlaufen wird. Beginnend mit jedem Schleifendurchlauf wird, wie in Algorithmus 3 beschrieben, eine entsprechende Anzahl von Superknoten ausgewählt. Diese Knoten fungieren nun als Superknoten und werden aus den aktiven Knoten entfernt. Die aktiven Knoten beinhalten also alle Knoten, für die in diesem Durchlauf ein Vorgänger ermittelt werden soll.

Nun werden die aktiven Kanten durchlaufen, wobei dies bei Beginn des Algorithmus alle Kanten umfasst. Besteht ein Endknoten einer Kante aus einem Superknoten und der andere Endknoten aus einem aktiven Knoten, so wird diese Kante überprüft. Zuerst wird die Möglichkeit des Hinzufügens der Kante ohne eine Verletzung des Delay-Constraints geprüft. Dies geschieht durch die Formel 5.1, die im Kapitel 5.3.1 beschrieben wird. Der entsprechende Superknoten wird dem aktiven Knoten als Vorgänger zugewiesen. Durch das vorhergehende Sortieren der Kanten nach ihren Kosten handelt es sich hierbei um die Kante zwischen dem aktiven Knoten und einem Superknoten, welche die niedrigsten Kosten bei gültigem Delay aufweist. Da sekundär nach den Delays sortiert wurde kann es auch keine Kante mit gleichen Kosten und geringerem Delay geben. Der aktive Knoten wird inaktiv gesetzt und ist somit für den weiteren Verlauf des Coarsenings vorerst nicht mehr von Interesse, sofern nicht der in Kapitel 5.3.2 diskutierte Spezialfall eintritt.

Alle aktiven Kanten, die nicht zwischen zwei Superknoten verlaufen, werden für den weiteren Verlauf des Coarsenings inaktiv gesetzt. Man beachte hier, dass die inaktiven Kanten in einer Liste gespeichert werden. Dies ist später für das Improvement nützlich, da die Kanten in der umgekehrten Reihenfolge deaktiviert werden, in der sie später für das Refinement bzw. das Improvement benötigt werden. Weiters wird der Knotengrad der beteiligten Superknoten reduziert, um für zukünftige Superknotenwahlen aktuelle Knotengrade zur Verfügung zu haben.

Im Anschluss wird geprüft, ob zu allen aktiven Knoten ein Vorgänger gefunden wurde. Für alle Knoten v, für die dies nicht der Fall ist, wird der beschriebene Vorgang zur Gewährleistung einer vollständigen Lösung verwendet. Der Vorgänger u dieses Knoten v im Shortest-Delay-Path wird nun auch als Vorgänger von v in der Lösungsrepräsentation eingesetzt. Die Kante (u, v) gehört nun also zur Teillösung. Dieser Vorgängerknoten u wird weiters als Superknoten eingesetzt und wird sich selbst als Vorgänger zugewiesen. Weiters werden alle Kanten, die für den weiteren Verlauf des Coarsenings interessant sein könnten, also jene, die zwischen u und anderen Superknoten verlaufen, aktiviert. Dies geschieht, indem geprüft wird, ob Kanten existieren, die zwischen dem neuen Superknoten und einem aktiven Knoten verlaufen. In dem Fall, dass eine solche Kante existiert und bereits inaktiv gesetzt wurde, wird diese wieder aktiviert.

Die Childdelays der Superknoten werden stets aktualisiert. Wird einem aktiven Knoten ein neuer Vorgänger zugewiesen, so wird geprüft, ob der Childdelay des Vorgängers geringer als der des

aktiven Knotens plus der Kante, die zwischen diesen verläuft, ist. Ist dies der Fall, so wird der Childdelay des Vorgängers auf diesen Wert gesetzt. Im dem Fall, dass ein Knoten nachträglich zum Superknoten gemacht wird, müssen auch hier die Childdelays aktualisiert werden. Während das Aktualisieren des Childdelays des neuen Superknotens analog zu einer Zuweisung eines neuen Vorgängers ist, muss der Childdelay des alten Vorgängers des neuen Superknotens ebenfalls betrachtet werden. Da dieser alte Vorgänger nun einen Nachfolger weniger hat ist es möglich, dass sich der Childdelay verringert. Entspricht der Childdelay des neuen Superknotens plus dem Delay der Kante zwischen dem neuen Superknoten und dem alten Vorgänger jenem des Childdelays des alten Vorgängers, so muss geprüft werden ob sich dieser verringert hat. Eine obere Grenze für das Aktualisieren der Childdelays des alten Vorgängers ist durch $\mathcal{O}(|V|)$ gegeben. Dies ist zwar eine sehr grobe obere Grenze, erleichtert jedoch die Laufzeitberechnung des gesamten Algorithmus.

Im Weiteren werden alle aktiven Knoten dieses Levels auf inaktiv gesetzt und die Superknoten nehmen den Platz der aktiven Knoten für den nächsten Schritt des Coarsenings ein.

Um die Laufzeit des Coarsening-Teils der Degree-based Multilevel Heuristik zu berechnen, werden zuerst einige Abschätzungen getätigt. Die Anzahl der Knoten in jedem Level l wird als $|V_l|$ bezeichnet und kann durch $|V| \cdot s^{l-1}$ abgeschätzt werden, wobei s für die Superrate steht. Für diese gilt $0 < s < 1$ und somit auch $s^2 < s$. Die Anzahl der Superknoten kann durch $|V_l| \cdot s$ und die Anzahl der aktiven Knoten durch $|V_l| \cdot (1 - s)$ beschrieben werden. Weiters ist für die Anzahl der aktiven Kanten $|E_l|$ mit $|V_l|^2$ eine obere Grenze gegeben.

Für jedes Level l werden die Superknoten ermittelt, was in $\mathcal{O}(|V_l| \cdot \log(|V_l|))$ geschieht. Weiters werden die aktiven Kanten durchlaufen, wobei der Aufwand $|E_l|$ beträgt. Zuletzt muss noch das Handling der Problemknoten betrachtet werden. Die Anzahl der Problemknoten kann durch die Anzahl der aktiven Knoten abgeschätzt werden und beträgt somit $|V_l| \cdot (1 - s)$. Für jeden Problemknoten besteht die Möglichkeit, dass eine Aktualisierung des Childdelays eines Knoten vorgenommen werden muss, was in $\mathcal{O}(|V|)$ geschieht. Somit kann der Aufwand für die Problemknoten mit $|V_l| \cdot (1 - s) \cdot |V|$ abgeschätzt werden. In Formel 5.2 findet sich die gesamte Aufwandsabschätzung für ein einzelnes Level l.

$$\mathcal{O}(|V_l| \cdot \log(|V_l|) + |E_l| + |V_l| \cdot (1 - s) \cdot |V|) \tag{5.2}$$

Sowohl die Auswahl der Superknoten, als auch das Durchlaufen der aktiven Kanten wird durch den Aufwand der Problemknoten dominiert. Es ist also nur der dritte Teil von 5.2 für die Laufzeitberechnung des gesamten Coarsening Algorithmus interessant. Als nächstes wird die Anzahl der Level betrachtet. Eine untere Grenze für die Anzahl der Level ergibt sich durch $\log_{1/s}(|V|)$. Allerdings kann sich durch die Problemknoten die Anzahl der Superknoten in einem Level erhöhen und somit auch die Anzahl der Level. Es ist schwierig eine gute obere Grenze zu finden, allerdings bietet es sich für die Aufwandsberechnung an, einfach von einer unendlichen Anzahl an Level auszugehen, woraus sich Formel 5.3 ergibt.

$$\sum_{l=1}^{\infty} |V_l| \cdot (1 - s) \cdot |V| \tag{5.3}$$

Durch Ersetzen von $|V_l|$ durch $|V| \cdot s^{l-1}$ ergibt sich Formel 5.4.

$$\sum_{l=1}^{\infty} |V| \cdot s^{l-1} \cdot (1 - s) \cdot |V| \tag{5.4}$$

Nun kann der Ausdruck $\left|V^2\right| \cdot (1-s)$ aus der Summe gezogen werden wodurch sich Formel 5.5 ergibt.

$$\left|V^2\right| \cdot (1-s) \cdot \sum_{l=1}^{\infty} s^{l-1} \tag{5.5}$$

Abschließend kann die Summe berechnet werden wodurch sich der Ausdruck in Formel 5.6 ergibt.

$$\left|V^2\right| \cdot (1-s) \cdot \frac{1}{1-s} \tag{5.6}$$

Nach Kürzen von $(1-s)$ und $\frac{1}{1-s}$ bleibt $\mathcal{O}(\left|V^2\right|)$ übrig. Zusätzlich muss jedoch beachtet werden, dass zu Beginn eine Sortierung der Kanten durchgeführt wird, was in $\mathcal{O}(|E| \cdot \log(|E|))$ geschieht. Somit ergibt sich als Worst-Case Abschätzung für den Aufwand für den Coarsening Algorithmus der Degree-based Multilevel-Heuristik $\mathcal{O}(|E| \cdot \log(|E|) + \left|V^2\right|)$.

Der Coarsening-Teil dieser Multilevel-Heuristik garantiert eine zulässige Näherungslösung. Diese Lösung kann direkt erzeugt werden oder im Zuge des Refinements durch lokales Improvement verbessert werden. Ein Verfahren zum lokalen Improvement wird in Kapitel 8 vorgestellt.

6 Ranking-basierte Multilevel Heuristik

Die grundlegende Überlegung für die zweite entwickelte Multilevel-Heuristik war eine alternative Auswahl der Kanten und Knoten zu verwenden. Anders als bei der Degree-based Multilevel-Heuristik werden die Kanten nicht nur nach ihren Kosten, sondern nach ihrer generellen Güte beurteilt. Dies bedeutet, dass Kanten bevorzugt werden, die sowohl geringe Kosten als auch einen geringen Delay aufweisen. Weiters sollen jene Knoten als Superknoten verwendet werden, die nicht lediglich einen hohen Knotengrad sondern möglichst gute Kantenverbindungen aufweisen. Hierzu wird ein neues Kantenattribut verwendet, welches die Qualität einer Kante in Bezug auf sowohl Kosten, als auch Delay im Vergleich zu anderen Kanten beschreibt. Dieses Attribut basiert auf einer Rangliste der Kanten im Bezug auf ihre Kosten und ihren Delay.

Ziel dieser Ranking-based Multilevel-Heuristik ist es also durch bevorzugtes Verwenden von Kanten, die keine sehr hohen Delays verursachen, viele Möglichkeiten offen zu halten. Im Unterschied zur Degree-based Multilevel Heuristik wird der Delay also nicht nur als ein Constraint gesehen, sondern auch als Entscheidungskriterium in den Coarseningvorgang eingebaut. Im Allgemeinen wird von dem Coarsening der Ranking-based Multilevel-Heuristik eine Lösung mit mehr Raum für Verbesserungen erzeugt. Im Anschluss wird, im Zuge des Refinements, versucht, die Lösung zu verbessern, wobei hier die tatsächlichen Kosten als Entscheidungskriterium verwendet werden.

6.1 Grundlegende Überlegungen

Die Degree-based Multilevel-Heuristik wählt aus den möglichen Kanten zum Verbinden eines Knotens zu einem Superknoten stets jene Kante, welche die geringsten Kosten aufweist. Diese Vorgehensweise erzeugt für das momentane Level und die gegebenen Superknoten eine optimale Teillösung. Man spricht von einer optimalen Teillösung, wenn es sich um einen Teilbereich einer Lösung, hier des Spannbaumes, handelt, welcher optimal gelöst wurde. Es gibt also keine kostengünstigere Möglichkeit, die gegebenen Knoten zu den gegebenen Superknoten zu verbinden. Die Optimalität dieser Teillösung ist jedoch auf speziell dieses Level, also die gegebenen Superknoten und die zu verbindenden Knoten beschränkt. Diese lokale Optimalität entspricht aber im Allgemeinen nicht gleichzeitig der Teillösung eines globalen Optimums.

Abbildung 6.1 zeigt die Problematik anhand eines Beispiels. Werden lediglich die Kosten der Kanten in Betracht gezogen, so wird Kante $(s, 2)$, Kante $(s, 3)$, Kante $(4, 5)$ und Kante $(4, 6)$ verwendet. Das Problem besteht in der Kante $(4, 6)$. Hier beträgt der Delay 3, wodurch nur noch ein Pfad mit Delaywert 2 zum Anbinden des Knotens 4 an den Sourceknoten verwendet werden kann. Dieser existiert in Form der Kante $(s, 2)$ und der Kante $(2, 4)$, welche allerdings sehr teuer ist. Die günstigere Kante $(3, 5)$ kann nun nicht mehr verwendet werden ohne den Delay-Constraint zu verletzen, und es entsteht ein Lösungsbaum mit Kosten 11. Eine bessere Lösung würde hier entstehen, wenn der Knoten 6 an den Knoten 5 angebunden wird. Die Kante $(5, 6)$ verursacht zwar höhere Kosten, aber ist vom Delay her wesentlich günstiger. Dadurch

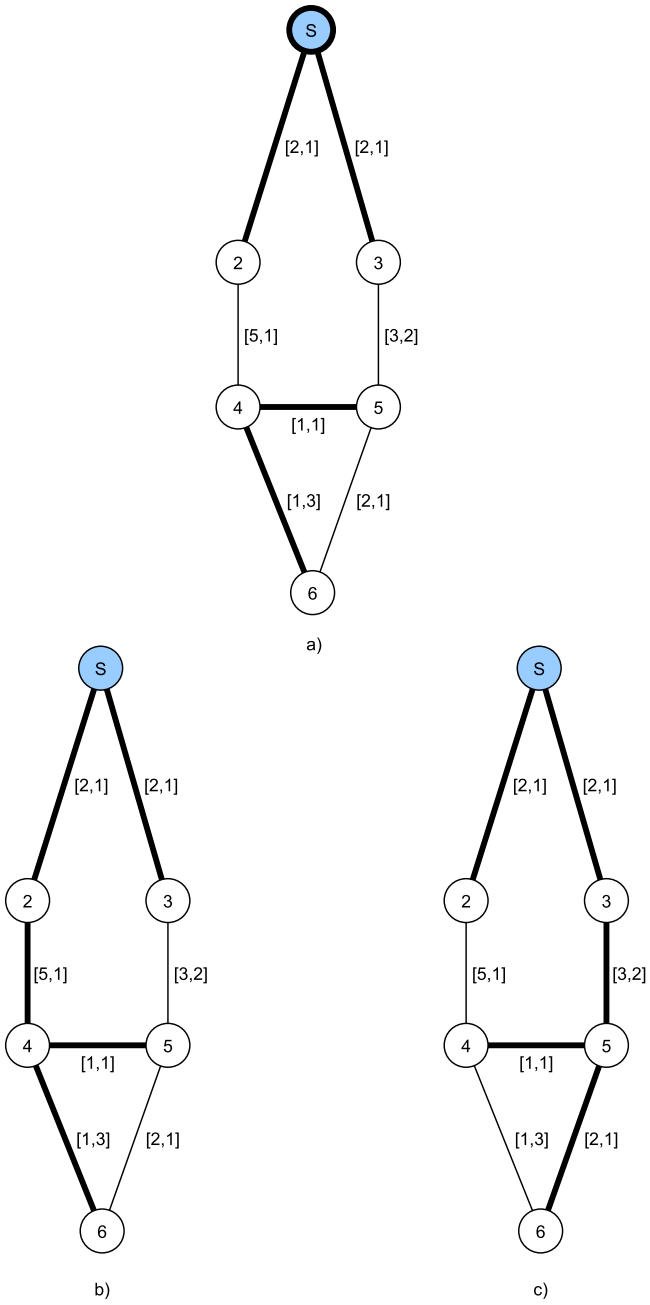

Abbildung 6.1: Ein Graph mit Delaygrenze B=5. Wird während der Kantenentscheidung in a) nur auf die Kosten geachtet, so entsteht Lösung b). Diese ist schlechter als das Optimum in c).

wird ermöglicht eine bessere Lösung zu konstruieren, die lediglich Kosten von 10 aufweist.

Es kann also durchaus von Vorteil sein, eine teurere aber kürzere, also im Bezug auf den Delay günstigere, Kante zu verwenden. Eine billigere Kante zu verwenden führt also nicht notgedrungen zu einem kostengünstigeren Baum. Das andere Extrem besteht darin, lediglich die Kanten mit dem kürzesten Delay zu verwenden und einen minimalen Spannbaum in Bezug auf den Delay zu konstruieren. Allerdings werden in dieser Lösung die Kosten vollends ignoriert, was im Allgemeinen auch keine gute Startlösung für lokales Improvement darstellt.

Ziel der Ranking-based Multilevel-Heuristik ist es nun einen Mittelweg zu beschreiten. Bei der Kantenentscheidung sollen sowohl Kosten als auch Delay berücksichtigt werden. Es wird also nach einem Maß für die generelle Qualität der Kanten gesucht.

6.2 Ranking Score

Das Qualitätsmaß für eine Kante soll sowohl Kosten als auch Delay miteinbeziehen und die Kanten miteinander vergleichbar machen. Im Verlauf des Coarsenings soll dann anhand dieser neuen Kanteneigenschaft entschieden werden, welche Kanten verwendet werden. Kritisch ist nun, wie man Kosten und Delay in eine Kanteneigenschaft so verpackt, dass diese wirklich aussagekräftig ist.

Kosten und Delay sind im praktischen Fall in der Regel nicht direkt miteinander vergleichbar. Denkt man zurück an das Beispiel der zentralen Sendestation, so entsprechen Kosten den tatsächlichen Währungskosten der Verbindung, während der Delay in einem Zeitmaß angegeben wird. Auch in abstrakteren Beispielen sind Kosten und Delay nicht notwendigerweise direkt miteinander vergleichbar, somit ist ein direktes in Beziehung Setzen der beiden Größen auch für ein Qualitätsmaß der Kanten nicht geeignet. Auch eine Normalisierung der Wertebereiche kann hier problematisch werden, sobald die Werte für Kosten und Delay in Beziehung gesetzt werden.

Für die Ranking-based Multilevel-Heuristik werden Ranglisten verwendet um die Kanten miteinander vergleichen zu können. Die Qualität einer Kante, in Relation zu den anderen Kanten, wird ermittelt, indem ihre Position in der Rangliste der Kosten sowie in jener der Delays verwendet wird. So wird eine Punktzahl, der Score oder Ranking Score, berechnet, welcher als neues Kantenattribut und Entscheidungskriterium verwendet wird. Die Berechnung dieser Scores erfolgt gemäß der Formel 6.1.

$$score(u,v) = \left(1 - \frac{r^c_{(u,v)} - 1}{|E|}\right) * \left(1 - \frac{r^d_{(u,v)} - 1}{|E|}\right) \tag{6.1}$$

Wobei hier $r^c_{(u,v)}$ für den Rang der Kante (u,v) in der Kostenrangliste und $r^d_{(u,v)}$ für den Rang der Kante (u,v) in der Delayrangliste steht. Diese Ränge entsprechen den Positionen der Kanten in einer nach Kosten bzw. Delay sortierten Rangliste. Für eine Kante (a,b), welche die niedrigsten Kosten aufweist, gilt also $r^c_{(a,b)} = 1$. Die Subtraktion von 1 und die Division durch die Anzahl der Kanten dient zur Normalisierung. Je höher der Rang einer Kante, desto größer ist also der Wert, der von 1 abgezogen wird, und desto geringer der Wert der jeweiligen Klammer. Durch Multiplikation der beiden Klammern ergibt sich also ein höherer Wert, wenn die Kante sowohl vergleichsweise niedrige Kosten als auch niedrigen Delay aufweist. Dieser Score repräsentiert

also die generelle Qualität der Kante. Je größer der Score der Kante (u, v) desto besser ist die Kante (u, v), im Allgemeinen im Vergleich zu anderen Kanten des Graphen.

Um den Ranking Score jeder Kante zu ermitteln, müssen die Kantenlisten zwei mal sortiert werden um die Ranglisten für die Kosten und den Delay zu ermitteln. Anschließend werden die Ranking Scores als zusätzliches Kantenattribut gespeichert, auf das während des Coarsenings zurückgegriffen wird. Der Ranking Score ersetzt gewissermaßen die Kosten, während der Delay weiterhin wie bisher verwendet wird um eine Gültigkeit der Lösung zu garantieren.

Der Ranking Score wird im Verlauf des Coarsenings der Ranking-based Multilevel-Heuristik allerdings nicht nur zur Kantenentscheidung verwendet. Auch die Wahl der Superknoten erfolgt mittels Ranking Score. Hierzu werden für alle Knoten die Ranking Scores der adjazenten Kanten aufsummiert.

$$score(u) = \sum_{v \in V | (u,v) \in E} score(u, v) \tag{6.2}$$

Die Scores werden für Knoten gemäß der Formel 6.2 berechnet. Diese Scores dienen anschließend, ähnlich wie die Knotengrade bei der Degree-based Multilevel-Heuristik, als Entscheidungskriterium. Jene Knoten mit dem höchsten Score werden als Superknoten verwendet, wobei der Score jedes Level aktualisiert wird, abhängig von den im aktuellen Graphen vorhandenen Kanten. Als Datenstruktur wird wie auch bei der Degree-based Multilevel Heuristik eine Priority-Queue verwendet, wobei hier die Priorität durch die Scores gegeben ist.

6.3 Coarseningphase

Die Coarseningphase ähnelt im Wesentlichen jener der Degree-based Multilevel-Heuristik. Als Vorverarbeitungsschritt werden die Ranking Scores für die Kanten und in weiterer Folge auch für die Knoten berechnet. Im Anschluss wird wieder Schritt für Schritt eine Lösung aufgebaut, wobei während des Coarsenings immer auf die Gültigkeit dieser geachtet wird.

6.3.1 Berechnung der Ranking Scores

Beginnend werden die Ranking Scores für die Kanten berechnet. Hierzu werden die Kanten nach ihren Kosten sortiert. Nun wird das Ranking erstellt, wobei, wenn zwei Kanten die selben Kosten aufweisen, beide auch den gleichen Rang erhalten. Darauffolgende Ränge werden ausgelassen. Bei zwei Kanten mit Kosten 1 und einer Kante mit Kosten 2 wären die Ränge also 1, 1 und 3. Es folgt die Berechnung der Kosten-Scores gemäß dem ersten Teil der Formel 6.1, um zu gewährleisten, dass bessere Ränge auch höhere Scores ergeben. Dies ermöglicht später das einfache Aufsummieren für die Knoten-Scores. Die Kosten-Scores werden als Kantenattribut gespeichert. Im Anschluss werden die Kanten nach dem Delay sortiert und der selbe Vorgang für den Delay angewandt. Die ermittelten Delay-Scores werden mit den Kosten-Scores multipliziert, um die schlussendlichen Ranking-Scores zu erhalten. Weiters wird der Score der beteiligten Knoten um diesen Ranking Score erhöht, um die Grundlage für die Superknotenauswahl zu schaffen. Dieser Vorgang wird in Algorithmus 5 gezeigt.

Algorithmus 5 : calcRankingScores

Input : Graph
Output : Ranking Scores für alle Knoten und Kanten

```
1  Sortiere E aufsteigend nach Delay
2  aktueller_Delay = 0
3  aktueller_Rang = 0
4  aktuelle_Position = 0
5  forall (u, v) ∈ E do
6      aktuelle_Position = aktuelle_Position + 1
7      if d((u, v)) ≠ aktueller_Delay then
8          score((u, v)) = 1 − (aktuelle_Position/ |E|)
9          aktueller_Rang = aktuelle_Position
10         aktueller_Delay = d((u, v))
11     else
12         score((u, v)) = 1 − (aktueller_Rang/ |E|)
13 Sortiere E aufsteigend nach Kosten
14 aktuelle_Kosten = 0
15 aktueller_Rang = 0
16 aktuelle_Position = 0
17 forall (u, v) ∈ E do
18     aktuelle_Position = aktuelle_Position + 1
19     if c((u, v)) ≠ aktuelle_Kosten then
20         score((u, v)) = score((u, v)) ∗ (1 − (aktuelle_Position/ |E|))
21         aktueller_Rang = aktuelle_Position
22         aktuelle_Kosten = c((u, v))
23     else
24         score((u, v)) = score((u, v)) ∗ (1 − (aktueller_Rang/ |E|))
25     score(u) = score(u) + score((u, v))
26     score(v) = score(v) + score((u, v))
```

6.3.2 Coarsening Algorithmus

Das Coarsening der Ranking-based Multilevel-Heuristik wird in Algorithmus 6 beschrieben. Nachdem die Ranking-Scores für die Knoten und Kanten berechnet wurden, werden die Kanten absteigend nach ihren Scores sortiert. Die Knoten mit den höchsten Scores, also jene mit den meisten und besten adjazenten Kanten, werden in jedem Level als Superknoten ausgewählt. Anschließend werden die aktiven Kanten durchlaufen und jene aktiven Kanten, die zwischen Superknoten und Nicht-Superknoten verlaufen, überprüft. Wird der Delay-Constraint nicht verletzt, so wird diese Kante verwendet. Nicht mehr benötigte Kanten werden inaktiv gesetzt und die Scores der Superknoten aktualisiert, indem die Scores der beteiligten Superknoten um jenen der Kante reduziert werden.

Auch hier werden wie zuvor für die Degree-based Multilevel-Heuristik jene Knoten, die nicht zu bestehenden Superknoten verbunden werden konnten, mittels Shortest-Delay-Path verbunden. Die entsprechenden Vorgänger dieser Knoten werden nachträglich zu Superknoten und

werden im nächsten Level behandelt. Im Allgemeinen wird es für die Ranking-based Multilevel-Heuristik weniger solcher Problemknoten geben, da diese vorsichtiger in Bezug auf den Delay vorgeht. Nach Abschluss des Coarsenings liegt eine vollständige Lösung vor. Diese wurde im Vergleich zur Degree-based Multilevel-Heuristik wesentlich vorsichtiger erstellt, was mehr Raum für ein Improvement während des Refinement-Schritts lässt.

Die Worst-Case Laufzeit der Coarsening Phase ist im Wesentlichen ident mit jener der Degree-based Multilevel-Heuristik, $\mathcal{O}(|E| \cdot \log(|E|) + |V|^2)$. Der Unterschied zwischen beiden Algorithmen beschränkt sich darauf, dass im Coarsening-Schritt der Ranking-based Multilevel-Heuristik mehr Kantensortierungen durchgeführt werden, was jedoch einen konstanten Faktor darstellt und somit vernachlässigbar ist.

Algorithmus 6 : Coarsening: Ranking-based Multilevel-Heuristik

Input : Graph
Output : Vorgänger für alle Knoten, indirekte Lösung

Kantenliste $L\_aktiveKanten$, $L\_inaktiveKanten$
Knotenliste $L\_aktiveKnoten$, $L\_Superknoten$
$L\_aktiveKnoten = V$
$L\_aktiveKanten = E$
calcRankingsScores(Graph)
forall $v \in V$ **do**
    $pred(v) = v$
Sortiere $L\_aktiveKanten$ absteigend nach Ranking Scores
$level = 0$
while $\exists v \in V \setminus \{s\} \mid pred(v) = v$ **do**
    $level = level + 1$
    $L\_Superknoten = Knotenauswahl(L\_aktiveKnoten, scores)$
    $L\_aktiveKnoten = L\_aktiveKnoten \setminus L\_Superknoten$
    forall $(u, v) \in E$ **do**
        if $v \in L\_aktiveKnoten \wedge u \in L\_Superknoten$ **then**
            if *sdp(u) + d((u,v)) + child_d(v)* $\leq B$ **then**
                $pred(v) = u$
                if *child_d(u) < child_d(v) + d((u,v))* **then**
                    $child\_d(u) = child\_d(v) + d((u, v))$
                $L\_aktiveKnoten = L\_aktiveKnoten \setminus \{v\}$
                $score(u) = score(u) - score(u, v)$
        else if $u \notin L\_Superknoten \wedge v \notin L\_Superknoten$ **then**
            $L\_aktiveKanten = L\_aktiveKanten \setminus \{(u, v)\}$
            $L\_inaktiveKanten = L\_inaktiveKanten \cup \{(u, v)\}$
    forall $v \in L\_aktiveKnoten$ **do**
        $u = sdp\_pred(v)$
        $L\_Superknoten = L\_Superknoten \cup \{u\}$
        $L\_aktiveKnoten = L\_aktiveKnoten \setminus \{u\}$
        $pred(v) = u$
        if *child_d(u) < child_d(v) + d((u,v))* **then**
            $child\_d(u) = child\_d(v) + d(u, v)$
        $oldpred = pred(u)$
        $pred(u) = u$
        if $child\_d(oldpred) = child\_d(u) + d((u, oldpred))$ **then**
            $child\_d(oldpred) = \max\{child\_d(w) + d((oldpred, w) \mid pred(w) = oldpred\}$
        forall $(u, w) \mid w \in L\_Superknoten \wedge (u, w) \in L\_inaktiveKanten$ **do**
            $L\_aktiveKanten = L\_aktiveKanten \cup \{(u, w)\}$
            $L\_inaktiveKanten = L\_inaktiveKanten \setminus \{(u, w)\}$
        $L\_aktiveKnoten = L\_aktiveKnoten \setminus \{v\}$
    $L\_aktiveKnoten = L\_Superknoten$

7 Component-basierte Multilevel Heuristik

Die dritte Multilevel-Heuristik basiert anders als die beiden zuvor behandelten nicht auf dem Verbinden von Knoten zu zuvor festgelegten Superknoten. Vielmehr wird ähnlich der Kruskal-basierten Heuristik, KBH, versucht, durch Verschmelzen von Kanten systematisch Komponenten zu erstellen, welche die betreffenden Knoten verbinden. Anders als bei der KBH ist jedoch nicht das Ziel durch das Verschmelzen direkt einen Lösungsbaum zu erstellen. Vielmehr wird versucht, den Verschmelzungsvorgang zu steuern um Komponenten zu erzeugen, die im weiteren Verlauf der Heuristik möglichst gut verwendet werden können.

Jede Komponente besitzt stets einen Majorknoten, welcher die Eigenschaft besitzt, dass von diesem Knoten aus der kürzeste Delaypfad zwischen der Komponente und dem Sourceknoten ausgeht. In gewisser Weise ähnelt dieser Majorknoten also den Superknoten aus den Degree- und Ranking-based Multilevel-Heuristiken. Im Unterschied zu diesen werden aber die anderen Knoten der Komponente nicht deaktiviert, sondern sind fortan als Minorknoten aktiv. Diese Minorknoten einer Komponente dienen weiterhin als Anschluss für andere Komponenten zur Verschmelzung.

Während des Verschmelzungsvorgangs bzw. dem Erstellen der Komponenten werden die Kanten durchlaufen, wobei diese in drei Arten unterschieden werden, jene die zwischen zwei Majorknoten verlaufen, jene die zwischen Major- und Minorknoten verlaufen und jene die zwischen zwei Minorknoten verlaufen. Für diese drei Arten von Kanten gelten unterschiedliche Bedingungen für eine Verschmelzung bzw. Hinzunahme zur Lösung.

7.1 Grundlegende Überlegungen

Die grundlegende Vorgangsweise der KBH liegt darin die Kanten nach ihren Kosten zu sortieren und diese dann, sofern der Delay-Constraint nicht verletzt wird, zur Verschmelzung der Knoten zu Komponenten, und im Anschluss zu einer vollständigen Lösung, zu verwenden. Dabei wird die Kantenliste lediglich einmal durchlaufen und das Verschmelzen erfolgt unbegrenzt mit Ausnahme der Berücksichtigung des Delay-Constraints.

Die Überlegung der Component-based Multilevel-Heuristik, CBMH, basiert nun prinzipiell auf dieser Vorgangsweise. Allerdings soll der Verschmelzungsprozess stärker eingeschränkt werden, um eine gleichmäßigere Lösungsfindung zu ermöglichen. Prinzipiell kann eine derartige Beschränkung auch als zusätzlicher Constraint gesehen werden.

Die Hauptfrage, die sich stellt, ist nun wie der Verschmelzungs- bzw. Coarseningvorgang eingeschränkt werden soll, damit eine bessere Lösung erzeugt werden kann. Eine zu starke Beschränkung des Coarsenings kann zu Problemen führen, wenn dadurch gute Verbindungen nicht

genutzt werden. Beispielsweise führt eine Einschränkung, dass jeder Knoten nur einmal pro Level an einem Verschmelzungsschritt teilnehmen darf, dazu, dass Knoten, anstatt durch günstige Kanten an einen bereits verwendete Knoten, durch teure Kanten an noch nicht verwendete angebunden werden. Abbildung 7.1 verdeutlicht dieses Problem.

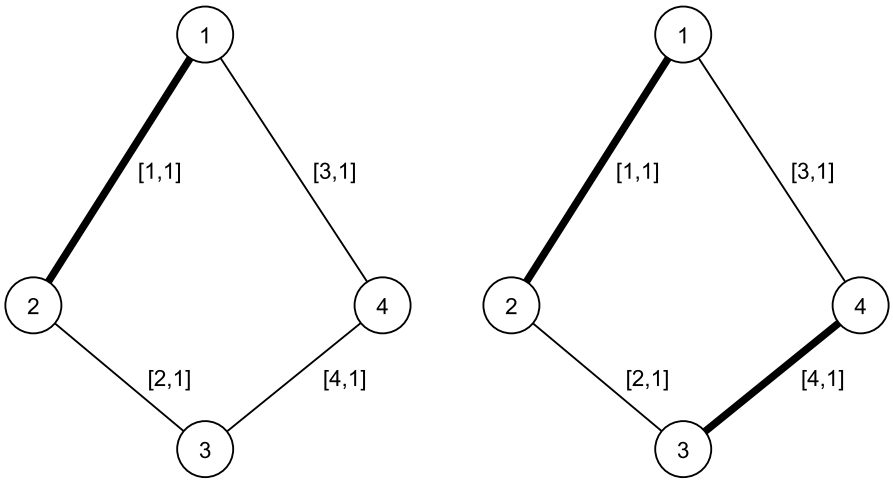

Abbildung 7.1: Durch die Einschränkung, dass jeder Knoten nur einmal pro Level für einen Mergevorgang benutzt werden kann, werden gute Verbindungen ignoriert.

Im Lauf der Diplomarbeit wurde sehr viel mit unterschiedlichen Einschränkungen für das Coarsening experimentiert. Das in den zuvor vorgestellten Heuristiken verwendete Modell von vordefinierten Superknoten, die den Coarseningvorgang steuern, ist ein Beispiel dafür. Die ursprüngliche Idee der CBMH war keine vordefinierten Superknoten zu verwenden, sondern die Superknoten während des Coarsenings entstehen zu lassen. Es ist also kein Zufall, dass eindeutige Parallelen zwischen Majorknoten und Superknoten zu erkennen sind. Beide sind der Wurzelknoten für die Teillösung, die als Baumform vorliegt. Allerdings werden die Knoten innerhalb dieser Teillösung oder Komponente nun nicht als inaktiv betrachtet, sondern stellen als Minorknoten immer noch Anschlussmöglichkeiten dar, wie es auch bei der KBH der Fall ist.

Ohne zusätzliche Einschränkungen entspricht ein Durchlaufen der Kanten und Verwenden jener, bei denen keine Verletzung des Delay-Constraints auftritt, der Vorgangsweise der KBH. Die CBMH unterscheidet nun allerdings drei verschiedene Kantentypen. Als Major-Major Kanten werden jene Kanten bezeichnet, die zwischen zwei Majorknoten verlaufen. Analog dazu werden Major-Minor Kanten und Minor-Minor Kanten definiert. Diese drei Kantentypen unterliegen in der CBMH unterschiedlichen Constraints.

Alle Knoten werden als Majorknoten initialisiert, da sie prinzipiell den Majorknoten einer Komponente, die nur diesen Knoten umfasst, entsprechen. Die zuvor nach ihren Kosten aufsteigend sortierten Kanten werden im Zuge der CBMH durchlaufen. Für Major-Major Kanten gilt neben der Einschränkung, dass der Delay-Constraint nicht verletzt werden darf, auch, dass nur ein Knoten der Kante als Majorknoten aus dem Verwenden der Kante bzw. dem Verschmelzen der Komponenten hervorgehen kann. Für die Kante (u, v) muss also entschieden werden, welcher der beiden Knoten der neue Majorknoten der Komponente ist und welcher Knoten von nun an nur noch den Status eines Minorknoten hat. Für diese Entscheidung werden sowohl die Shortest-Delay-Paths als auch die Childdelays der Knoten herangezogen. Gilt $sdp(u) + child\_d(v) \leq sdp(v) + child\_d(u)$ so wird der Knoten u zum Majorknoten der neuen Komponente und v zum Minorknoten. Man beachte, dass die Bevorzugung des Knotens u bei gleichem Wert darauf beruht, dass die Entscheidung anhand der Informationen, die zu dem Zeitpunkt vorliegen, keine Rolle spielt. Diese Überprüfung wird deshalb durchgeführt, da der Knoten, der daraus als besser hervorgeht, im Allgemeinen leichter, in Bezug auf den Delay, zum Sourceknoten verbunden werden kann. Grund hierfür ist, dass entweder der Shortest-Delay-Path wesentlich geringer ist oder die Komponente einen geringeren Childdelay aufweist. Grundsätzlich wird eine Major-Major Kante jedoch immer verwendet, insofern keine Verletzung des Delay-Constraints vorliegt.

Für Major-Minor Kanten gelten jedoch mehr Einschränkungen. Im weiteren wird die Komponente des Minorknotens mit X bezeichnet und die Komponente des Majorknotens mit Y. Wie zuvor darf auch hier der Delay-Constraint nicht verletzt werden. Offensichtlich müssen die Knoten auch in verschiedenen Komponenten liegen, da sonst Kreise entstehen würden. Die markanteste Einschränkung ist jedoch, dass eine Major-Minor Kante nur dann verwendet werden darf, wenn der Shortest-Delay-Path des Majorknotens von Y nicht geringer ist als jener des Majorknotens von X. Die Begründung für diese Einschränkung liegt darin, dass es zwar möglich und in Bezug auf die Kosten günstig wäre, die Kante zu verwenden, allerdings liegt Y prinzipiell näher, in Bezug auf den Delay, als X am Sourceknoten. Dies bedeutet gleichzeitig, dass der Majorknoten, eventuell auch die Minorknoten, von Y eine günstigere Delayverbindung zum Sourceknoten aufweisen. Durch das Verwenden der Kante würde diese günstige Verbindung verschlechtert, da jede andere Komponente, die zukünftig zu einem Knoten in Y verbunden werden soll, den Umweg über X gehen muss. Man sieht hier bereits, dass die CBMH nicht nur eine Lösung erzeugen, sondern gleichzeitig Möglichkeiten offen halten will. Eine Illustation findet sich in Abbildung 7.2. Die Kanten $(4, 5)$ und $(5, 6)$ werden als erste geprüft und der Knoten 5 geht als Majorknoten der Komponente hervor. Ein Verwenden der Kante $(3, 4)$ wäre der nächste Schritt, jedoch wird dadurch die Möglichkeit, den Knoten 3 als günstige Verbindung einzusetzen, verbaut, was in diesem Beispiel zu einer Lösung mit Kosten 15 führt. Wird diese Kante nicht verwendet, wie es in der CBMH geschieht, so bleibt der Knoten 3 als günstige Anbindungsmöglichkeit bestehen und es entsteht eine Lösung mit Kosten 13. Ist der Shortest-Delay-Path des Minorknotens geringer, so wird der Majorknoten zu diesem verbunden und wird ebenfalls zu einem Minorknoten.

Der dritte Kantentyp, die Minor-Minor Kanten, werden während des Erstellens der Komponenten ignoriert. Es wäre natürlich denkbar, dass das Verwenden einer Minor-Minor Kante einen Vorteil bringt. Allerdings wird das Verwenden einer solchen Kante in der Regel hohe Delays innerhalb der Komponenten verursachen und weitere Möglichkeiten stark verbauen. Ein weiterer Grund, warum diese Kanten ignoriert werden, ist, dass sie im Zuge des Improvements leicht erfasst werden können.

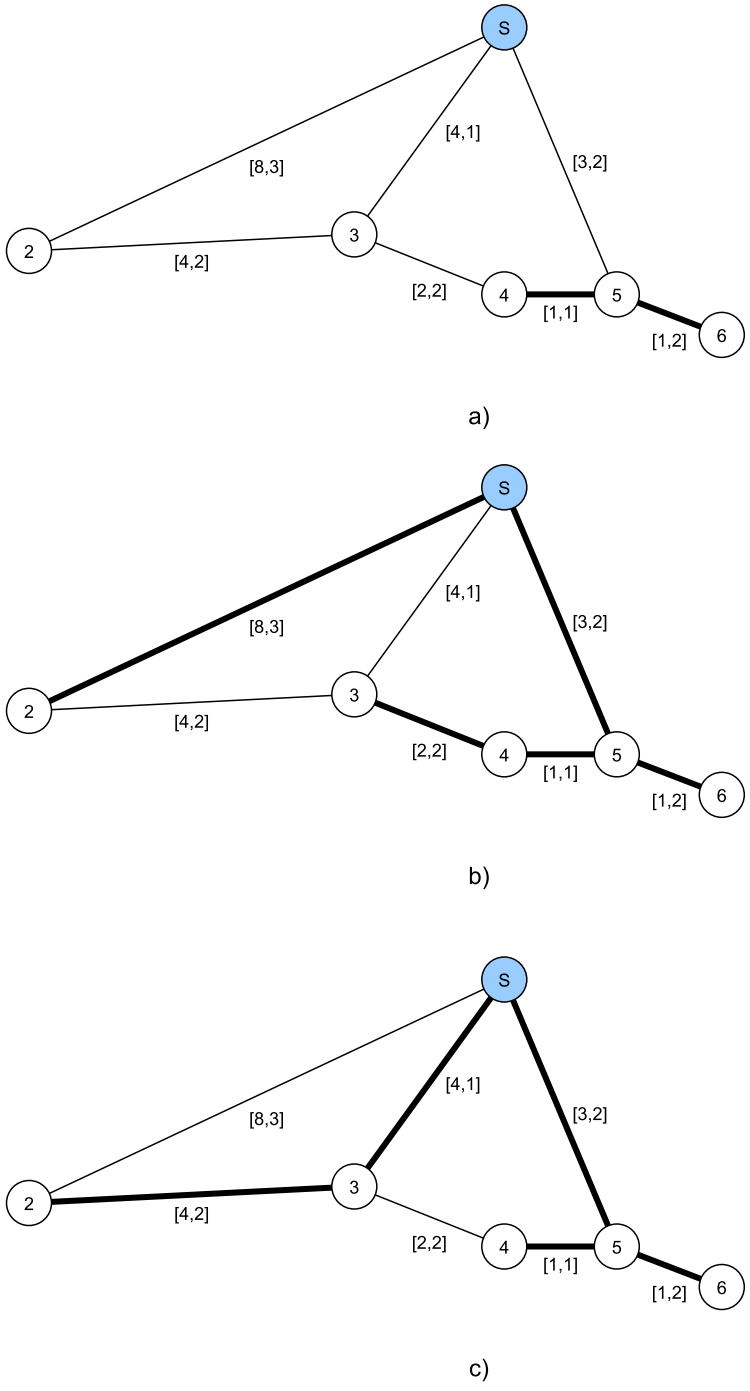

Abbildung 7.2: Der Graph in a) zeigt die Ausgangssituation. Das Verwenden der Kante $(3, 4)$ in b) verhindert eine günstige Anbindung des Knotens 2, wodurch ein Graph mit Kosten 15 entsteht. In c) wird dies wie auch bei der CBMH vermieden, wodurch ein Graph mit Kosten 13 entsteht.

Da im Allgemeinen nicht garantiert werden kann, dass so eine vollständige Lösung erzeugt wird, besteht die CBMH aus einem zweiten Teil. Hier werden ähnlich wie bei der KBH die verbleibenden Komponenten mittels Shortest-Delay-Paths zu einer vollständigen Lösung verbunden. Hierbei wird für übriggebliebene Komponenten der Shortest-Delay-Path des Majorknotens der Komponente verfolgt. Trifft man entlang des Shortest-Delay-Paths auf einen Knoten v, so wird geprüft, ob ein direktes Hinzufügen der Kante zwischen dem Majorknoten und v zu einer Delayverletzung führt. Ist dies nicht der Fall, wird die Kante hinzugefügt und die Komponente ist verbunden. Liegt jedoch eine Verletzung vor, wird die Kante zwar ebenfalls hinzugefügt, allerdings wird v der neue Majorknoten der Komponente und die Kante zum bisherigen Vorgänger von v wird entfernt. Dies geschieht bis keine Komponenten außer der Sourcekomponente übrig sind. Abbildung 7.3 zeigt diesen Vorgang. Die Komponente mit Majorknoten 5 soll verbunden werden. Das Hinzufügen der Kante $(4, 5)$ führt zu einer Verletzung des Delay-Constraints, daher wird 4 der neue Majorknoten der Komponente und die Kante $(3, 4)$ wird entfernt. Nun wird die Komponente mit Majorknoten 4 angebunden. Das Hinzufügen der Kante $(2, 4)$ verletzt den Delay-Constraint nicht. Die Komponente ist erfolgreich angebunden und eine vollständige Lösung wurde konstruiert.

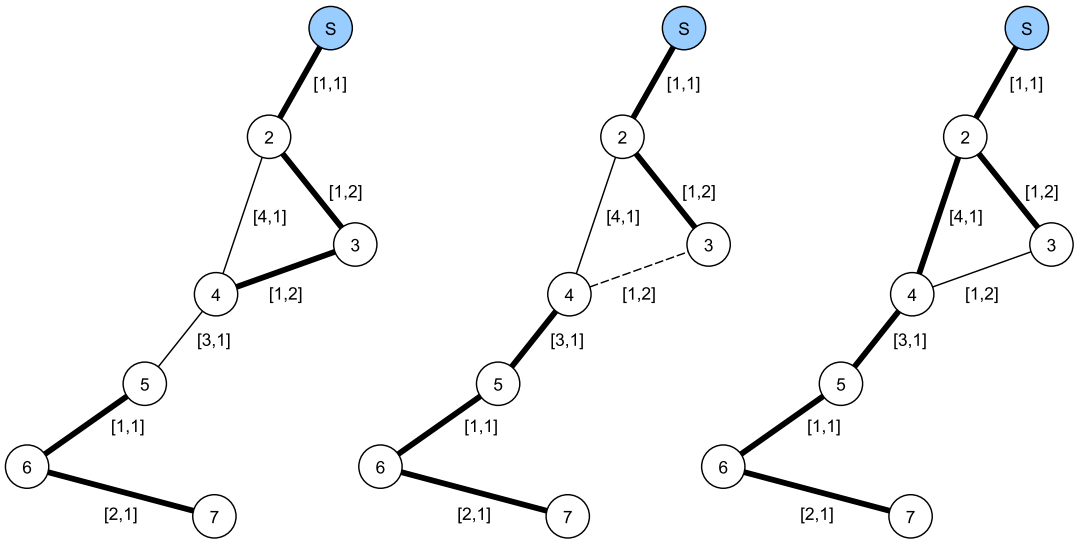

Abbildung 7.3: Komponente 5 kann nicht zum Sourceknoten verbunden werden. Der Shortest-Delay-Path wird verfolgt, um eine vollständige Lösung zu konstruieren.

Es bleibt nun die Frage, ob es sich bei der CBMH noch um ein Multilevel-Verfahren handelt. Die Kanten werden nur einmal durchlaufen, eine klassische Levelstruktur ist nicht vorhanden und man könnte das Verfahren eigentlich als Constrained KBH bezeichnen. Wenn man allerdings

ein wenig lockerer an den Begriff Multilevel herangeht, kann man sehr wohl eine Levelstruktur erkennen. Wenn man sich den Graphen im Raum vorstellt, so kann man sich die Knoten als Körper und die Kosten und Delays der Kanten, den Shortest-Delay-Path und die Childdelays der Knoten als Kräfte vorstellen, die auf diese Körper wirken. Anfangs ziehen Knoten mit attraktiverem Delay (Shortest-Delay-Path) Knoten in der Nähe an und es entstehen Komponenten. Diese Komponenten haben eine klare, Zwei-Level-Struktur, den Majorknoten, den man als Planet interpretieren kann, und seine Minorknoten, welche in dem Vergleich die Monde darstellen. Die Komponenten wollen alles in der Nähe (kostengünstige Kanten) in einen Orbit aufnehmen und kommen sich dabei in die Quere. Knoten die zuvor noch Majorknoten waren werden nun zu Monden, haben aber gleichzeitig noch ihre eigenen Monde, die jetzt als Satelliten interpretiert werden können. Allerdings wird im Fall einer Major-Minor Kante der Majorknoten nicht zum Mond des Minorknoten sondern zum Mond des Majorknoten der Komponente. Dies passiert aber erst bei Kanten mit höheren Kosten, also auf das Beispiel umgelegt in einem weiteren Orbit. In jeder Komponente, schlussendlich auch in der Sourcekomponente, die nach Abschluss der CBMH eine vollständige Lösung repräsentiert, liegen die Knoten also in unterschiedlichen Orbits. Jede Komponente hat also ihre eigene Levelstruktur basierend auf den Kosten der Kanten. Verringert man die Kostengrenze, man könnte dies als Anziehungskraft der Masse sehen, so werden Komponenten frei, die selbst wieder eine Levelstruktur besitzen. Allerdings ist dies zugegebenermaßen alles andere als eine klassische Multilevelstruktur und ob es sich bei der CBMH wirklich um eine Multilevel-Heuristik handelt oder eine Constrained-KBH ist Interpretationssache.

7.2 Der Algorithmus

Die Component-based Multilevel-Heuristik besteht im wesentlichen aus zwei Teilen. Zuerst wird der Graph zu Komponenten zusammengefasst. Im Anschluss werden diese Komponenten miteinander verbunden, um eine vollständige Lösung zu konstruieren.

7.2.1 Erstellen der Komponenten

Algorithmus 7 zeigt den Algorithmus zum Erstellen der Komponenten. Beginnend werden die Kanten nach ihren Kosten sortiert. Weiters wird der Knotenstatus aller Knoten auf Majorknoten gesetzt. Der Knotenstatus eines Knotens beschreibt, ob es sich bei dem Knoten um einen Major- oder Minorknoten handelt. Die Komponente eines Knotens gibt den Majorknoten der Komponente an, in welcher sich der Knoten befindet. Da jeder Knoten als Majorknoten initialisiert wird, entspricht folglich jeder Knoten einer Komponente.

Im Verschmelzungsprozess werden die aktiven Kanten durchlaufen bis entweder alle Kanten überprüft wurden oder nur noch eine Komponente, die Sourcekomponente, übrig ist. Abhängig vom Knotenstatus der beteiligten Knoten jeder Kante wird diese gesondert behandelt, wobei hier in zwei Fälle unterschieden wird. Handelt es sich bei den beteiligten Knoten um zwei Majorknoten, so wird geprüft, welcher der Knoten den geringeren Wert aus Shortest-Delay-Path des Knotens plus Childdelay des anderen Knoten aufweist. Weiters wird geprüft, ob ein Verwenden der Kante zu einer Delay-Constraint Verletzung führt. Ist dies nicht der Fall, so werden die beteiligten Komponenten verschmolzen und die Kante zur Lösung hinzugefügt. Der dominante Knoten, jener mit dem geringeren Wert aus dem Check in Zeile 14, bleibt Majorknoten, während der andere Knoten zum Minorknoten wird. Weiters wird der Majorknoten zum Vorgänger dieses neuen Minorknoten.

Der zweite Fall tritt ein, wenn die Kante einen Major- und einen Minorknoten verbindet. Um Kreisbildungen zu vermeiden wird auch geprüft, ob die Kante zwei Knoten aus verschiedenen Komponenten verbindet. Ist dies der Fall, wird geprüft, ob eine Delay-Constraint Verletzung vorliegt. Die letzte Bedingung besteht darin, dass der Shortest-Delay-Path des Majorknotens der Komponente des Minorknotens geringer ist als der Shortest-Delay-Path des Majorknotens. Werden alle Bedingungen erfüllt, so wird die Kante zur Lösung hinzugefügt und der Minorknoten der Kante wird zum Vorgänger des Majorknotens der Kante, welcher nun auch den Status eines Minorknotens erhält.

Im Anschluss an jede Verschmelzung erfolgt eine Aktualisierung der Knoteninformation der beteiligten Komponenten, siehe Algorithmus 8. Der Childdelay des Majorknotens der entstandenen Komponente wird aktualisiert, siehe Algorithmus 9. Weiters muss für alle Knoten der Komponente des neuen Minorknotens die Komponentenzugehörigkeit aktualisiert werden. Diese Knoten werden der Komponente des Majorknotens zugewiesen.

7.2.2 Gewährleisten einer vollständigen Lösung

Im Allgemeinen ist nicht garantiert, dass das Erstellen der Komponenten zu genau einer Komponente, der Sourcekomponente, führt. Alle Komponenten, die keine Verbindung zum Sourceknoten aufweisen, müssen zu diesem verbunden werden, siehe Algorithmus 10. Hierzu wird, ähnlich wie bei den zuvor vorgestellten Heuristiken, der Shortest-Delay-Path verwendet. Ausgehend vom Majorknoten der zu verbindenden Komponenten wird der Shortest-Delay-Path verfolgt. Sobald der Vorgänger im Shortest-Delay-Path in einer anderen Komponente liegt, wird die Kante zu diesem Vorgängerknoten hinzugefügt. Führt dies zu keiner Delay-Constraint Verletzung, ist die Komponente erfolgreich verbunden. Ist dies nicht der Fall, so wird der Vorgänger im Shortest-Delay-Path zum neuen Majorknoten der Komponente und die Kante zwischen diesem und dessen Vorgänger wird aus der Lösung entfernt. Anschließend wird der Shortest-Delay-Path weiter verfolgt bis eine Anbindung der aktuellen Komponente erfolgt ist. Dieser Vorgang liefert eine vollständige Lösung.

7.2.3 Laufzeit des Algorihmus

Die Laufzeit der gesamten Heuristik ergibt sich wie folgt. Wie bei den Heuristiken zuvor führt das Sortieren der Kanten zu einem Aufwand von $\mathcal{O}(|E| \cdot \log(|E|))$. Im Zuge des Erstellens der Komponenten werden alle Kanten durchlaufen, allerdings wird maximal $|V| - 1$ mal eine Verschmelzung durchgeführt. Im Zuge einer Verschmelzung entsteht durch das Aktualisieren der Knoteninformation ein Aufwand der $\mathcal{O}(|V|)$ beträgt. Man beachte, dass nur dann $|V| - 1$ Verschmelzungen durchgeführt werden, wenn schon während des Erstellens der Komponenten eine vollständige Lösung erzeugt wird. Bleiben Komponenten unverbunden, so werden $|V| - 1$ minus die Anzahl der übrigen Komponenten $|K| - 1$ Verschmelzungen durchgeführt. $|K| - 1$ entspricht allerdings genau der Anzahl an Komponenten die im Zuge des Gewährleistens einer vollständigen Lösung verbunden werden müssen. Das Verbinden einer Komponente erfolgt in $\mathcal{O}(|V|)$, wodurch sich für die Gesamtlaufzeit der Heuristik $\mathcal{O}(|E| \cdot \log(|E|) + |V|^2)$ ergibt.

Algorithmus 7 : Erstelle Komponenten

Input : Graph
Output : Komponenten

```
1  Knotenliste L_aktiveKnoten
2  Kantenliste L_aktiveKanten, L_solution
3  L_aktiveKanten = E
4  Sortiere L_aktiveKanten aufsteigend nach Kosten, sekundär nach Delay
5  act = L_aktiveKanten.first()
6  forall v ∈ V do
7      v_status(v) = major
8      component(v) = v
9      pred(v) = v
10 while act ≠ L_aktiveKanten.end() ∧ |Komponenten| > 1 do
11     (u, v) = act
12     if v_status(u) == major ∧ v_status(v) == major then
13         if sdp(u) + child_d(v) > sdp(v)+child_d(u) then
14             swap(u, v)
15         if sdp(u) + d((u,v)) + child_d(v) ≤ B then
16             pred(v) = u
17             L_solution = L_solution ∪ {(u, v)}
18             v_status(v) = minor
19             AktualisiereKnoteninformation(Graph, u, v)
20     else if v_status(u) ≠ v_status(v) ∧ component(u) ≠ component(v) then
21         if v_status(u) == major ∧ v_status(v) == minor then
22             swap(u, v)
23         if sdp(component(u)) + componentdelay(u) + d((u, v)) + child_d(v) ≤ B
           ∧ sdp(component(u)) < sdp(v) then
24             pred(v) = u
25             L_solution = L_solution ∪ {(u, v)}
26             v_status(v) = minor
27             AktualisiereKnoteninformation(Graph, u, v)
28     act = L_aktiveKanten.next()
```

Algorithmus 8 : AktualisiereKnoteninformation

Input : Graph, Knoten u, Knoten v
Output : aktualisierte Knoteninformationen für die Knoten der beteiligten Komponenten

```
1  AktualisiereChilddelay(u)
2  forall Knoten j | component(j)==u do
3      component(j) = v
```

Algorithmus 9 : AktualisiereChilddelay

Input : Graph, Knoten u
Output : aktualisierte Childdelays für die Knoten der Komponenten von u

```
1 parentvertex = pred(u)
2 delaypath = child_d(u) + d((u, parentvertex))
3 while child_d(parentvertex) < delaypath do
4     child_d(parentvertex) = delaypath
5     delaypath = delaypath + d((parentvertex, pred(parentvertex)))
6     parentvertex = pred(parentvertex)
```

Algorithmus 10 : Gewährleisten einer vollständigen Lösung

Input : Graph, Teillösung $L\_solution$
Output : Lösung

```
1  while |Komponenten| > 1 do
2      Wähle Komponente deren Majorknoten k ≠ s
3      act = k
4      pathpred = sdp_pred(act)
5      connected = false
6      while connected == false do
7          if sdp(component(pathpred)) + componentdelay(pathpred) +
           d((act,pathpred)) + child_d(act) < B then
8              L_solution = L_solution ∪ {(act, pathpred)}
9              connected = true
10             AktualisiereChilddelays(act)
11         else
12             L_solution =
               L_solution ∪ {(act, pathpred)} \ {(pathpred, pred(pathpred))}
13             if child_d(pathpred) < child_d(act) + d((act,pathpred)) then
14                 child_d(pred) = child_d(act) + d((act, pred))
15             act = pathpred
16             pathpred = sdp_pred(pathpred)
```

8 Refinement

Dieses Kapitel befasst sich mit dem Refinement. Weiters wird ein Improvementverfahren, welches während des Refinements angewandt werden kann vorgestellt.

8.1 Refinementphase

Nach Abschluss des Coarsenings ist bereits eine Lösung vorhanden. Im Fall von Degree-based Multilevel-Heuristik (DBMH) und Ranking-based Multilevel-Heuristik (RBMH) sind die Vorgänger für jeden Knoten bekannt. Durch Hinzufügen der Kanten zwischen allen Knoten und deren Vorgängern kann einfach ein Lösungsbaum erzeugt werden. Im Fall der Component-based Multilevel-Heuristik (CBMH) wurden die Kanten bereits hinzugefügt. Für DBMH und RBMH kann, alternativ zum einfachen Hinzufügen der Kanten, auch nach Verbesserungen während des Refinements gesucht werden. Während der Graph schrittweise wiederhergestellt und gleichzeitig ein Spannbaum erstellt wird, kann in jeder Stufe nach Verbesserungen gesucht werden. Hierzu werden alternative Anbindungen der Knoten auf die Auswirkungen auf die Kosten und ihre Gültigkeit im Bezug auf den Delay überprüft. In diesem Kapitel wird eine geeignete Nachbarschaft und ein lokales Suchverfahren vorgestellt.

Anders als bei DBMH und RBMH entsprechen bei der CBMH die eigentlichen Level nicht gleichzeitig den Ebenen der Lösung. Der Algorithmus der in diesem Kapitel vorgestellt wird, kann jedoch auch für die CBMH verwendet werden. Hierbei werden die Ebenen des Lösungsbaumes als Level übernommen.

Das entwickelte Verfahren nutzt durch das Coarsening gewonnene Information in Form von Childdelays und Baumstruktur. Weiters wird nun der tatsächliche Delay zwischen einem Knoten und dem Sourceknoten, Sourcedelay genannt, ermittelt und verwendet.

8.1.1 Nachbarschaft

Die Nachbarschaft ist vergleichsweise einfach. Sie ist definiert durch das Ersetzen des Vorgängers eines Knoten durch einen anderen Vorgänger. Dieses Ersetzen entspricht dem Austausch von Kanten. Die Kante, die zwischen einem Knoten und seinem momentanen Vorgänger verläuft, wird einer Kante, die zwischen besagtem Knoten und einem alternativen Vorgänger verläuft, gegenübergestellt. Hierbei ist zu beachten, dass die Größe der Nachbarschaft von der Anzahl der Kanten in dem momentanen Graphen abhängt. Während des Refinements wird der Graph, ausgehend vom Sourceknoten, Level für Level wiederhergestellt. Beginnend besteht der aktuelle Graph also lediglich aus dem Sourceknoten und keinen Kanten. Während der Graph schrittweise wiederhergestellt wird, werden stets die Nachfolgeknoten der Blattknoten des aktuellen Graphen aktiviert. Es wird also ein Level hinzugefügt. Weiters werden alle Kanten, die zwischen diesen Knoten verlaufen, aktiviert. Die Größe der Nachbarschaft in einem Level entspricht also $\mathcal{O}(|aktiveKanten|)$.

8.1.2 Suchverfahren

Im Zuge des Improvements während des Refinement-Schritts werden alle Knoten des aktuellen Graphen als potentielle Vorgänger für andere Knoten angesehen. Der Unterschied zum Coarsening besteht nun darin, dass der Delay zwischen den Knoten und dem Sourceknoten nicht mehr nur als minimaler Delay in Form des Shortest-Delay-Paths, sondern als tatsächlicher Delaypfad zum Sourceknoten, dem Sourcedelay, angegeben wird. Da die Childdelays der Knoten aus dem Coarsening bekannt sind und gleichzeitig der Sourcedelay während des Refinements leicht ermittelt werden kann, ist das Potential für Verbindungen bekannt. Mit anderen Worten, man kann berechnen, ob eine alternative Anbindung eines Knotens mit höherem Delay möglich ist. Für alle Kanten (u, v) muss die Formel 8.1 gelten, wobei u dem Vorgänger von v entspricht.

$$source\_d(u) + d((u, v)) + child\_d(v) \leq B \tag{8.1}$$

Die aktiven Kanten des Graphen werden durchlaufen und geprüft. Wird eine Kante gefunden, die einen Knoten kostengünstiger an einen Vorgänger anbinden kann und wird die Formel 8.1 für diese Kante erfüllt, so wird diese alternative Kante verwendet. Man beachte, dass sich dadurch sowohl der Childdelay des Knotens u und jener seiner Vorgänger verändern kann. Weiters kann sich der Sourcedelay des Knoten v und dessen Nachfolger verändern. Auf diese Aktualisierungen wird später genauer eingegangen. Da diese Aktualisierungen jedoch in Bezug auf die Laufzeit relativ teuer sind, werden die Kanten stets nach ihren Kosten sortiert, um zu vermeiden, dass ein Knoten innerhalb eines Durchlaufs mehrere Neuzuweisungen erhält.

Abbildung 8.1 illustriert den Vorgang des Refinements und des Improvements. Bei den grau unterlegten Knoten handelt es sich, ebenso wie bei den strichlierten Kanten, um deaktivierte Elemente des Originalgraphen. Dieser wird schrittweise ausgehend vom Sourceknoten wiederhergestellt. Der unmittelbare Nachfolger des Sourceknotens ist hier der Knoten 2, welcher gemeinsam mit der Kante $(s, 2)$ hinzugefügt wird. Im nächsten Schritt werden die Nachfolger der neu aktivierten Knoten, in diesem Fall nur Knoten 2, aktiviert. Es handelt sich dabei um die Knoten 3 und 4. Zusätzlich werden auch jene Kanten wieder aktiviert, die zwischen diesen Nachfolgerknoten und zwischen einem Nachfolgeknoten und einem anderen aktivierten Knoten verlaufen.

Im Zuge des Improvements werden diese neu aktivierten Kanten nun überprüft um mögliche Verbesserungen für den Graph zu finden. Während die Kante $(s, 3)$ im Vergleich zu der Kante $(2, 3)$ keine Kostenverbesserung mit sich bringt, ist die Kante $(3, 4)$ gegenüber der Kante $(2, 4)$ in Bezug auf die Kosten günstiger. Nun wird mittels der Formel 8.1 geprüft, ob die Kante verwendet werden kann ohne den Delay-Constraint zu verletzen. Der Sourcedelay des Knotens 3 beträgt 2, während der Childelay des Knotens 4, aus dem Coarsening bekannt, 1 beträgt. Ein Verwenden einer Kante mit Delay 2 ist also möglich und gleichzeitig günstiger. Der Vorgänger des Knotens 4 wird also auf 3 geändert bzw. die Kante $(2, 4)$ durch die Kante $(3, 4)$ ersetzt. Nun müssen die Sourcedelays und Childdelays der beteiligten Knoten aktualisiert werden, bevor die nächste Kante überprüft oder zum nächsten Schritt des Refinements übergegangen werden kann.

8.1.3 Aktualisieren der Delays

Zur Überprüfung, ob das Verwenden einer Kante möglich ist, ohne den Delay-Constraint zu verletzen, ist es erforderlich, dass sowohl die Sourcedelays als auch die Childdelays der beteiligten Knoten korrekt sind. Ein Austausch eines Vorgängers bzw. ein Ersetzen der Kante zwischen

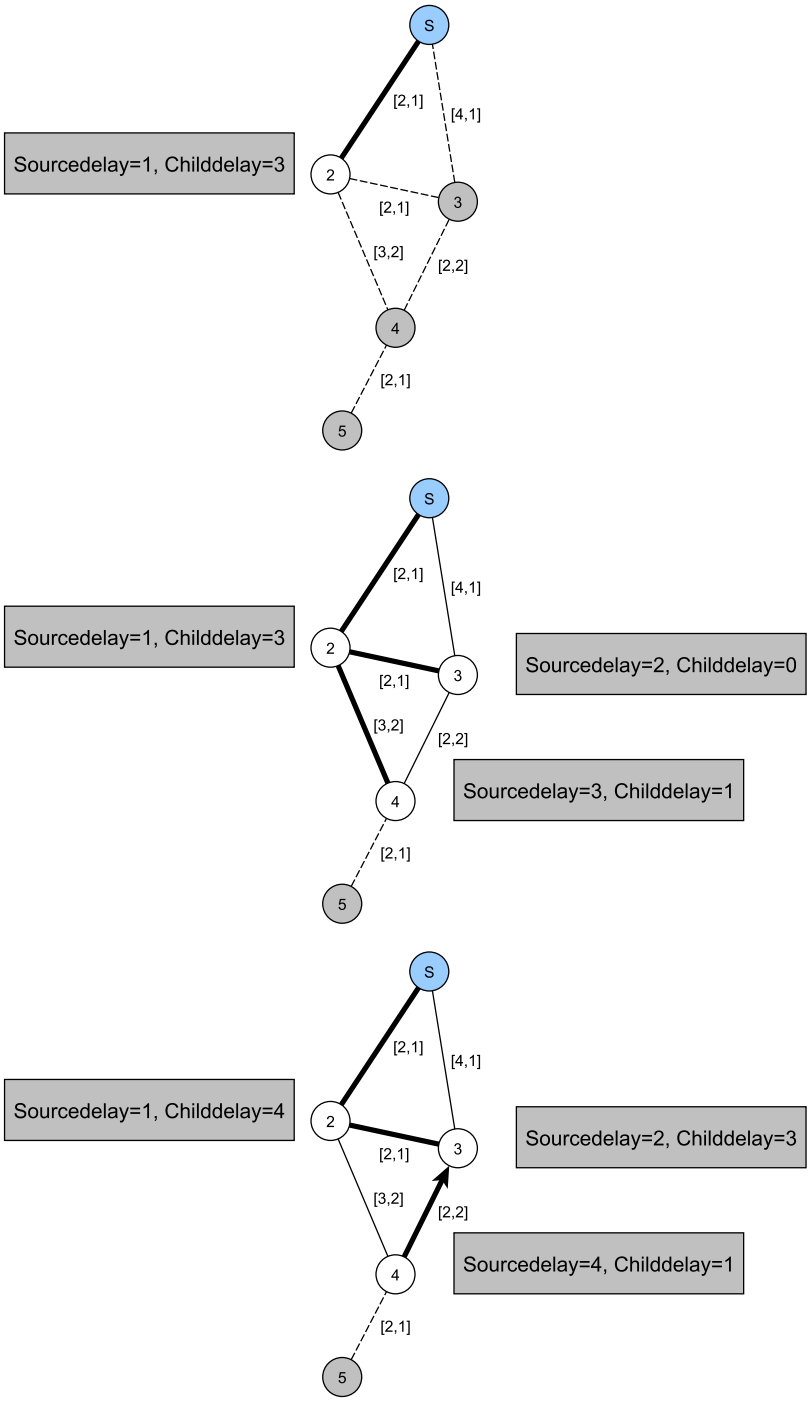

Abbildung 8.1: Ein Graph mit Delaygrenze B=5. Der Graph wird im Zuge des Refinements wiederhergestellt. Nach Berechnung der Sourcedelays wird die Kante $(2, 4)$ durch die Kante $(3, 4)$ ersetzt, da dies den Delay-Constraint nicht verletzt und die Kosten verringert.

einem Knoten, in weiterer Folge Updateknoten genannt, und dessen alten Vorgänger durch die Kante zwischen diesem Updateknoten und dem neuen Vorgänger kann die Childdelays für alle Knoten entlang des Pfades von beiden Vorgängern zum Sourceknoten verändern. Ebenso kann sich der Sourcedelay für sowohl alle direkten als auch indirekten Nachfolgerknoten verändern.

Um die kontinuierliche Korrektheit der Delays zu gewährleisten, muss nach jedem Austausch eine Aktualisierung der Delayinformationen der betroffenen Knoten vorgenommen werden. Das Update der Sourcedelays der Nachfolgeknoten gestaltet sich relativ einfach. Eine eventuelle Veränderung des Sourcedelays des Updateknotens muss an alle Nachfolger weitergegeben werden. Ist der neue Sourcedelay des Updateknotens gleich dem alten Sourcedelay des Updateknotens, muss auch für die Nachfolger kein Update vorgenommen werden. Besteht jedoch Ungleichheit, wird die Veränderung an alle jene Knoten weitergegeben, für die der Updateknoten als Vorgänger fungiert. Da aber auch indirekte Nachfolger, also Knoten, deren Pfad zum Sourceknoten den Updateknoten beinhaltet, existieren können, muss diese Veränderung auch an diese Knoten weitergegeben werden. Zur Durchführung dessen wird, ausgehend vom Updateknoten, eine Tiefensuche im aktuellen Graphen durchgeführt. Folgend werden alle gefundenen aktiven Knoten aktualisiert. Dies gewährleistet eine Aktualisierung der Sourcedelays aller direkten und indirekten Nachfolger des Updateknotens. Der Aufwand für diese Aktualisierung entspricht $\mathcal{O}(|V|)$.

Zum Gewährleisten der Korrektheit müssen nun noch die Childdelays aktualisiert werden. Dabei ist zu beachten, dass sich die Childdelays aller Knoten entlang des alten und neuen Pfades vom Updateknoten zum Sourceknoten verändern können. Der Childdelay eines Knotens gibt, wie bereits erläutert, den Delay zwischen dem Knoten und jenem Blattknoten seines Unterbaumes an, der den längsten Delaypfad aufweist. Es ist also stets nur der maximale Delay gespeichert, nicht jedoch um welchen Pfad es sich handelt. Durch Entfernen einer Kante zwischen einem Updateknoten und dessen Vorgänger kann sich der Childdelay entweder verringern oder er bleibt gleich. Genauso kann das Hinzufügen einer Kante den Childdelay des neuen Vorgängers nur erhöhen oder nicht beeinflussen.

Eine Erhöhung bzw. Verringerung des Childdelays des neuen oder alten Vorgängers kann daher auch zu einer Erhöhung bzw. Verringerung des Vorgängers dieser Knoten führen. Bleibt der Childdelay eines Knotens jedoch gleich, so kann sich der Childdelay des Vorgängers nicht verändern. Zur Illustration des Updatevorgangs werden die Abbildungen 8.2 und 8.3 verwendet. In Abbildung 8.2 wird die Kante $(5, 8)$ durch die günstigere Kante $(4, 8)$ ersetzt. Diese weist jedoch einen höheren Delay auf, wodurch sich der Sourcedelay des Knotens 8 erhöht. Ebenso muss der Sourcedelay des Nachfolgeknotens 9 erhöht werden. In Bezug auf die Childdelays werden der alte und neue Vorgänger des Knotens 8 überprüft. Knoten 5, der alte Vorgänger, hat nun keine Nachfolger mehr, daher wird der Childdelay auf 0 gesetzt. Der Knoten 4 hingegen besitzt nun einen Nachfolger mit Childdelay 1. Zusätzlich verursacht die neu hinzugefügte Kante $(4, 8)$ einen Delay von 2, was zu einer Erhöhung des Childdelays des Knotens 4 auf den Wert 3 führt. Weiters muss der Knoten 2 upgedatet werden. Während der Nachfolger 5 nun einen geringeren Childdelay verursachen würde, verursacht der Nachfolger 4 einen höheren. Dies führt auch für den Knoten 2 zu einer Erhöhung. Diese setzt sich fort bis zum Sourceknoten, welcher nun einen Childdelay von 5 aufweist.

Nun, da die Delays aktualisiert sind, wird in Abbildung 8.3 eine weitere Verbesserung vorgenommen. Hierbei wird die Kante $(6, 10)$ durch die Kante $(7, 10)$ ersetzt. Der Sourcedelay des

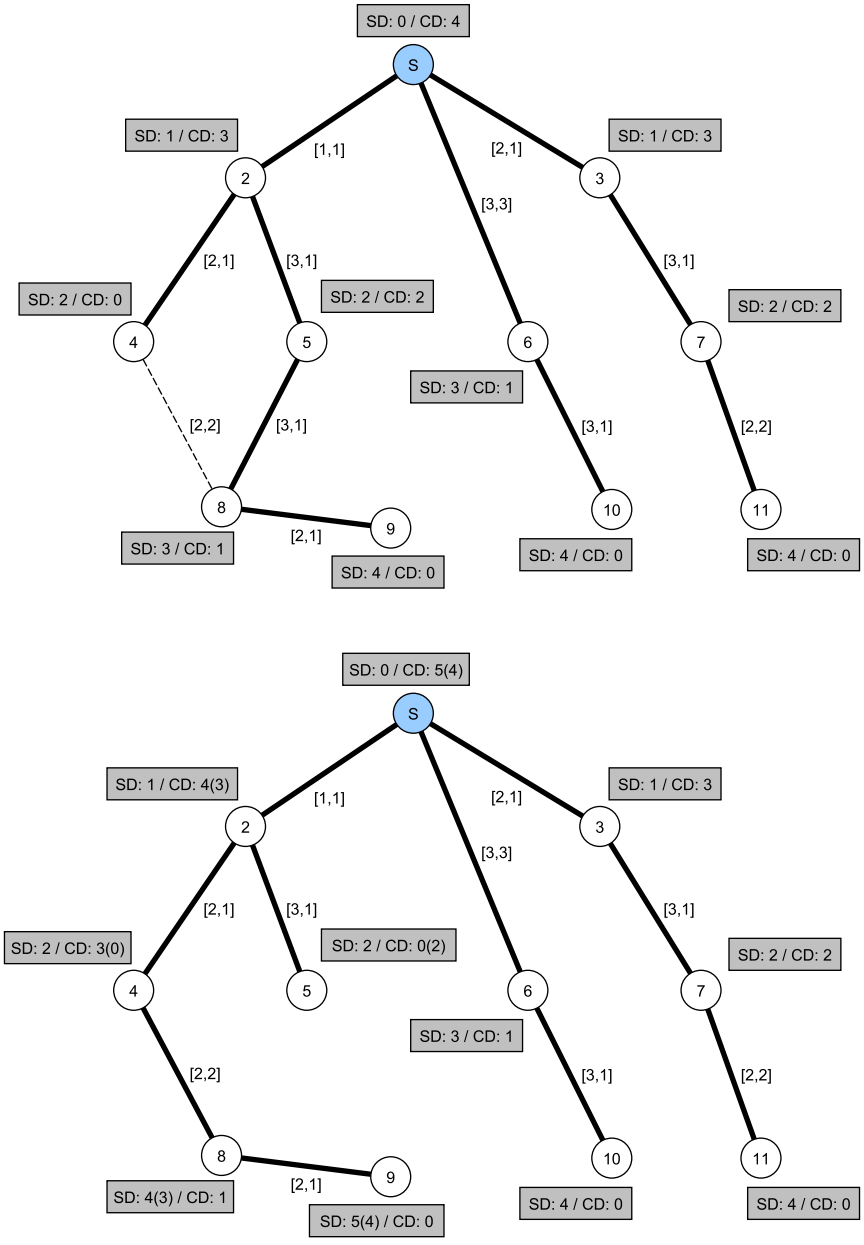

Abbildung 8.2: Ein Graph mit Delaygrenze B=5. Hier wird ein Update der Delays nach Ersetzen der Kante $(5, 8)$ durch die Kante $(4, 8)$ vorgenommen.

Knotens 10 verändert sich nicht. Folglich müssten eventuelle Nachfolger von 10 nicht aktualisiert werden. Für den alten Vorgänger 6 wird der Childdelay auf 0 gesetzt, da dieser nun keine Nachfolgeknoten mehr besitzt. Durch diese Verringerung muss auch der Childdelay des Sourceknotens aktualisiert werden, da jedoch der Pfad zwischen dem Sourceknoten und dem Knoten 6 nicht dem maximalen Delay aller Nachfolger entspricht, wird der Childdelay des Sourceknotens nicht verändert. Für den neuen Vorgänger, Knoten 7, ergibt sich kein höherer Childdelay, da sowohl der Delay zu dem bisherigen Nachfolger als auch zu dem neuen 2 beträgt. Folglich kann sich auch der Childdelay des Knotens 3 nicht verändern.

Der Aktualisierungsvorgang der Delays kann sich als relativ aufwändig gestalten. Während das Aktualisieren der Vorgängerknoten in der Regel relativ schnell ablaufen wird, kann der Pfad theoretisch alle Knoten umfassen. Auch die Anzahl der Nachfolgeknoten kann sehr hoch sein. Generell führt ein Austausch von Kanten in einem tieferen Level des Baumes zu einem höheren Aufwand beim Aktualisieren der Childdelays, während ein Austausch in höheren Ebenen mehr Nachfolgerknoten beeinflusst. Hier ist jedoch zu berücksichtigen, dass nicht immer alle Nachfolgeknoten aktiviert sind. Jedoch führt die Tatsache, dass eine Aktualisierung nach jedem einzelnen Improvement notwendig ist, dazu, dass das Improvement während des Refinements relativ aufwändig ist.

8.1.4 Refinement und Improvement

Algorithmus 11 zeigt den Ablauf des Refinements und des Improvements. Ein Refinement ohne Improvement besteht lediglich aus dem Hinzufügen der Kanten zwischen den Knoten und ihren Vorgängern. Wird begleitend nach Verbesserungen der Lösung gesucht, ist der Ablauf des Refinements komplexer. Beginnend wird der Sourceknoten aktiv gesetzt. Nun werden schrittweise die Nachfolger der Blattknoten des aktuellen Graphen aktiviert. Für alle neu hinzugekommen aktiven Knoten wird der Sourcedelay berechnet und im Fall von DBMH und RBMH werden die Kanten zwischen diesen Knoten und ihren Vorgängern zur Lösung hinzugefügt, was sich bei der CBMH erübrigt. Weiters werden alle Kanten aktiviert, die zwischen aktiven Knoten verlaufen. Es wird stets darauf geachtet, dass die aktiven Kanten aufsteigend nach ihren Kosten sortiert sind. Dies gewährleistet, dass für keinen Knoten mehrfache Aktualisierungen vorgenommen werden müssen.

Im Anschluss werden die aktiven Kanten durchlaufen. Für jede Kante (u, v) wird geprüft, welcher Knoten als Updateknoten getestet wird. Hierbei wird jener Knoten gewählt, für den der Sourcedelay plus der Childdelay des anderen Knoten geringer ist. Dadurch wird garantiert, dass der Knoten getestet wird, der einerseits mit höherer Wahrscheinlichkeit den Delay-Constraint nicht verletzt und andererseits den Delay, im Fall einer Kantenersetzung, weniger ausreizt.

Für den ausgewählten Knoten wird nun getestet, ob eine Ersetzung der Kante den Delay-Constraint verletzt. Ist dies nicht der Fall, so wird getestet, ob eine Kantenersetzung einen Kostenvorteil mit sich bringt. Hierzu wird die Kante (u, v) mit der Kante $(u, pred(u))$ verglichen. Sind die Kosten der Kante (u, v) geringer, erfolgt eine Kantenersetzung. Im Fall von gleichen Kosten wird weiters geprüft, ob eine Kantenersetzung einen geringeren Delay mit sich bringt. Dieser Fall kann eintreten, da zuvor im Coarsening nicht alle Kanten überprüft wurden. Wird eine Kantenersetzung durchgeführt, so werden die Lösung und die Knoteninformationen aktualisiert.

Das Aktualisieren der Sourcedelays erfolgt in Algorithmus 12. Die Differenz des alten und neuen

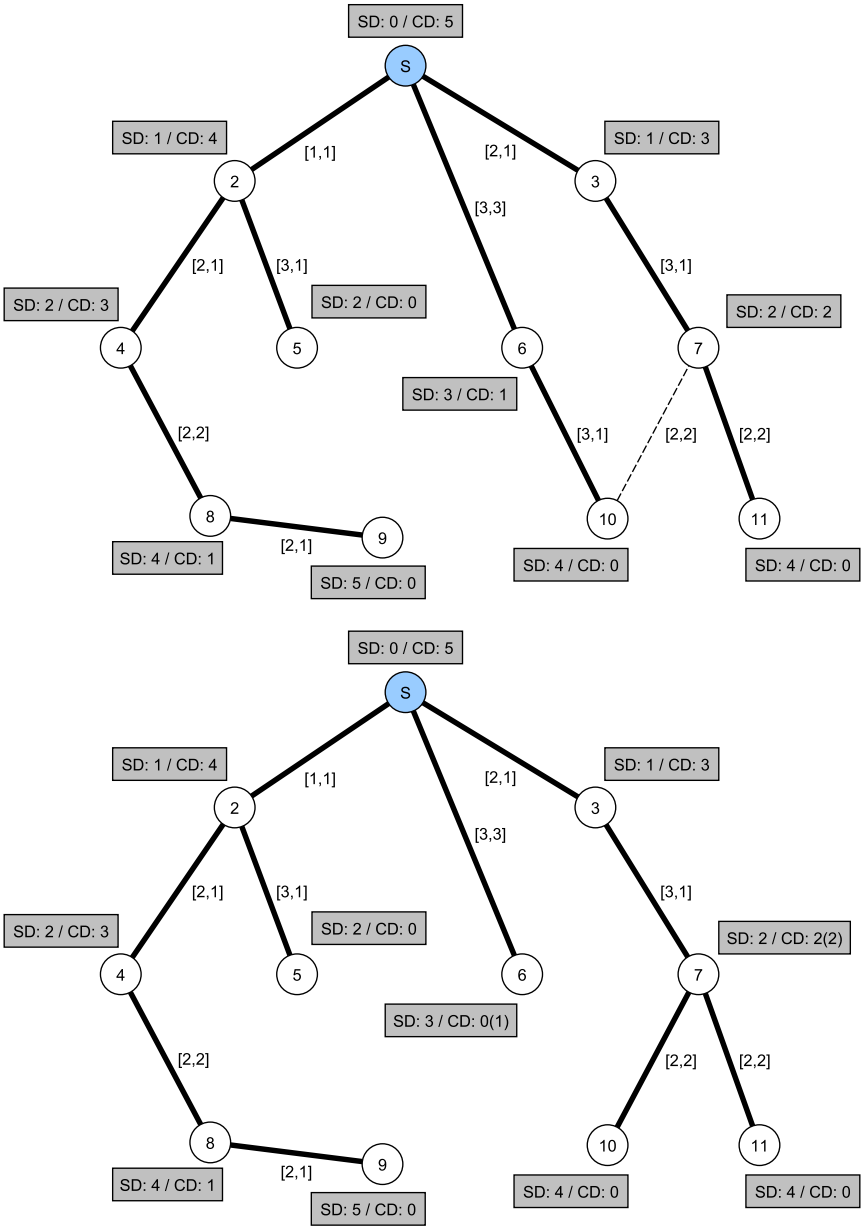

Abbildung 8.3: Ein Graph mit Delaygrenze B=5. Hier wird ein Update der Delays nach Ersetzen der Kante $(6, 10)$ durch die Kante $(7, 10)$ vorgenommen.

Sourcedelays wird berechnet. Ist die diese Differenz ungleich 0, so werden die Sourcedelays aller direkten und indirekten Nachfolger des Knotens, im aktuellen Graphen, aktualisiert.

Die Childdelays werden in Algorithmus 13 aktualisiert. Zuerst wird der Childdelay des alten Vorgängerknotens behandelt. Hierzu wird der Childdelay neu berechnet. Da der alte Vorgängerknoten nun einen Nachfolger weniger hat, kann der neuberechnete Childdelay entweder gleich oder geringer als der bisherige sein. Im Fall von gleichen Childdelays ist das Aktualisieren abgeschlossen. Ist jedoch der Childdelay geringer, so ist es möglich, dass auch der Vorgänger des alten Vorgängers einen geringeren Childdelay aufweist. Es muss also, entlang des Pfades zwischen dem alten Vorgänger und dem Sourceknoten, für jeden Knoten überprüft werden, ob sich der Childdelay verringert hat. Bleibt der Childdelay jedoch für einen Knoten gleich, so kann auch für dessen Vorgänger kein geringerer Childdelay entstehen und die Aktualisierung, für den alten Vorgängerknoten, ist abgeschlossen. Ähnlich gestaltet sich die Aktualisierung der Childdelays für den neuen Vorgängerknoten. Da dieser nach der Kantenersetzung einen weiteren Nachfolgeknoten aufweist, kann sich der Childdelay nur erhöhen oder gleich bleiben. Im Fall einer Erhöhung kann sich diese auch im Vorgänger des neuen Vorgängerknotens fortsetzen, wodurch auch hier der Pfad zum Sourceknoten überprüft werden muss. Sobald der neu berechnete Childdelay jedoch gleich dem alten Childdelay ist, ist die Aktualisierung abgeschlossen.

Sobald alle aktiven Kanten durchlaufen wurden, wird geprüft, ob in diesem Durchlauf eine Kantenersetzung durchgeführt wurde. Ist dies der Fall, so werden alle Kanten erneut durchlaufen. Dies ist notwendig, da durch das Ersetzen einer Kante neue Delays und somit eventuell neue Möglichkeiten für andere Kantenersetzungen entstehen. Dieser Fall kann beispielsweise auftreten, wenn eine alter Vorgänger nun einen niedrigeren Childdelay aufweist. Wurde keine Ersetzung durchgeführt, so wird der Graph um das nächste Level erweitert. Ist der Graph bereits vollständig wiederhergestellt, so ist das Refinement abgeschlossen.

Die Laufzeit des Refinement Algorithmus kann wie folgt berechnet werden. In dem Fall, dass kein Improvement durchgeführt wird beträgt die Laufzeit lediglich $\mathcal{O}(|V|)$. Wird jedoch Improvement durchgeführt, ist die Laufzeit deutlich höher. Prinzipiell kann die Startlösung $\mathcal{O}(|V|)$ Level aufweisen. Das Aktivieren aller Knoten erfolgt in $\mathcal{O}(|V|)$, allerdings werden nach jedem Aktivieren von Knoten auch Kanten aktiviert, wodurch eine Sortierung notwendig wird, was zu einem Aufwand von $\mathcal{O}(|V| \cdot |E| \cdot \log(|E|))$ führt. Man beachte jedoch, dass bereits aktivierte Kanten bereits sortiert sind.

In jedem Level werden die aktiven Kanten durchlaufen, welche durch $\mathcal{O}(|E|)$ begrenzt sind. Das Aktualisieren der Sourcedelays erfolgt, wie auch das Aktualisieren der Childdelays, in $\mathcal{O}(|V|)$. Die Kanten sind jedoch nach Kosten, sekundär nach Delay, sortiert, wodurch pro Durchlauf lediglich $\mathcal{O}(|V|)$ Updates durchgeführt werden. Problematisch ist, dass die Anzahl der Durchläufe, die in jedem Level durchgeführt werden, ebenfalls $\mathcal{O}(|V|)$ betragen kann, was allerdings sehr unwahrscheinlich ist, da der Delay in der Regel bereits nach wenigen Durchläufen ausgeschöpft ist. Prinzipiell bedeutet dies, dass der Aufwand eines Levels $\mathcal{O}(|V| \cdot |E| + |V|^2)$ beträgt, wobei der erste Teil dominierend ist. Für den gesamten Algorithmus bedeutet dies, durch die höchstmögliche Anzahl der Level, einen Aufwand von $\mathcal{O}(|V|^2 \cdot |E|)$. Wie bereits erwähnt, handelt es sich dabei jedoch um eine sehr pessimistische Abschätzung. In der Regel wird eine Lösung nicht aus $|V|$ Level bestehen, da die Lösung kein Pfad sondern eher breitgefächert sein wird. Zusätzlich werden im Zuge des Improvements die Kosten minimiert ohne dabei den Delay zu beachten, mit der Einschränkung, dass der Delay-Constraint nicht verletzt werden darf. Dies führt dazu, dass

Algorithmus 11 : Refinement

Input : Graph nach dem Coarsening, bool *Improvement*
Output : Lösung

```
1  Knotenliste L_aktiveKnoten, L_inaktiveKnoten
2  Kantenliste L_aktiveKanten, L_solution
3  L_inaktiveKnoten = V \ {s}
4  L_aktiveKnoten = {s}
5  if Improvement == false then
6      forall v ∈ V \ {s} do
7          L_solution = L_solution ∪ {(v, pred(v))}
8  else
9      while |L_solution| < |V| − 1 do
10         forall Blattknoten u des aktuellen Baumes do
11             forall v ∈ L_inaktiveKnoten | v = Nachfolger(u) do
12                 source_d(v) = source_d(u) + d((u, v))
13                 L_solution = L_solution ∪ {(u, v)}
14                 L_aktiveKnoten = L_aktiveKnoten ∪ {v}
15         forall
           (u, v) ∉ L_aktiveKanten |u ∈ L_aktiveKnoten ∧ v ∈ L_aktiveKnoten do
16             L_aktiveKanten = L_aktiveKanten ∪ {(u, v)}
17         Sortiere L_aktiveKanten aufsteigend nach Kosten, sekundär nach Delay
18         while Improvement == true do
19             Improvement = false
20             forall (u, v) ∈ L_aktiveKanten do
21                 if child_d(u) + source_d(v) < child_d(v) + source_d(u) then
22                     swap(u,v)
23                 parentvertex = pred(u)
24                 if source_d(v) + d((u,v)) + child_d(u) ≤ B then
25                     if (c((u,v)) < c((u,parentvertex))) ∨ (c((u,v)) =
                       c((u,parentvertex)) ∧ d((u,v)) + source_d(v) <
                       d((u,parentvertex)) + source_d(parentvertex) then
26                         L_solution = L_solution \ {(u, parentvertex)} ∪ {(u, v)}
27                         Improvement = true
28                         UpdateSourceDelay(aktuellerGraph, u, v)
29                         UpdateChildDelay(aktuellerGraph, u, v)
30                         pred(u) = v
```

Algorithmus 12 : UpdateSourceDelay

Input : Graph, Updateknoten u, neuer Vorgänger v
Output : Graph mit aktualisierten Sourcedelays

```
1  delaydiff = source_d(v) + d((u, v)) − source_d(u)
2  if delaydiff ≠ 0 then
3      source_d(u) = source_d(u) + delaydiff
4      forall direkten und indirekten Nachfolger j von u do
5          source_d(j) = source_d(j) + delaydiff
```

Algorithmus 13 : UpdateChildDelay

Input : Graph, Updateknoten u, neuer Vorgänger v
Output : Graph mit aktualisierten Childdelays

```
1  oldpred = pred(u)
2  if child_d(oldpred) == child_d(u) + d((u,oldpred)) then
3      oldchilddelay = child_d(oldpred)
4      Updated = true
5      while Updated == true do
6          child_d(oldpred) = 0
7          forall direkten Nachfolger j von oldpred do
8              if child_d(j) + d((j,oldpred)) > child_d(oldpred) then
9                  child_d(oldpred) = child_d(j) + d((j, oldpred))
10         if child_d(oldpred) == oldchilddelay ∨ oldpred == s then
11             Updated = false
12         else
13             oldpred = pred(oldpred)
14             oldchilddelay = child_d(oldpred)
15 if child_d(v) < child_d(u) + d((u,v)) then
16     newpred = v
17     child_d(newpred) = child_d(u) + d((u, newpred))
18     Updated = true
19     while Updated == true do
20         forall direkten Nachfolger j von newpred do
21             if child_d(j) + d((j,newpred)) ≥ child_d(newpred) then
22                 child_d(newpred) = child_d(j) + d((j, newpred))
23                 Updated = false
24         if Updated == true ∧ newpred ≠ s then
25             nextpred = pred(newpred)
26             child_d(nextpred) = child_d(newpred) + d((newpred, nextpred))
27             newpred = nextpred
```

auch Kanten mit höherem Delay verwendet werden, wodurch auch die Childdelays und Sourcedelays der Knoten anwachsen. Dies erschwert Kantenersetzungen in weiteren Durchläufen in demselben Level, wodurch in der Regel relativ wenige Durchläufe pro Level durchgeführt werden müssen.

9 Vergleich der Ergebnisse

Im Zuge der Diplomarbeit wurden für die drei entwickelten Heuristiken Tests mit verschiedenen Instanzen durchgeführt. Im weiteren werden die verschiedenen Instanztypen vorgestellt und die Testumgebung beschrieben. Im Anschluss werden die Ergebnisse der einzelnen Verfahren präsentiert und erläutert. Als Vergleichsgröße gilt hier das Ergebnis der Kruskal-basierten Heuristik (KBH). Für die KBH wurde weiters ein lokales Improvement mittels Kantenaustausch durchgeführt, um möglichst vergleichbare Ergebnisse zu erhalten. Im Anschluss daran werden die einzelnen Verfahren miteinander verglichen, um so einen direkten Vergleich der Verfahren und ihren Stärken und Schwächen zu erhalten.

9.1 Testumgebung und Instanztypen

Die Tests wurden auf einem Blade-Server - 2x Intel Xeon E5540 2.53 GHz, 24 GB Ram - des Instituts für Computergraphik und Algorithmen der Technischen Universität Wien durchgeführt. Als Testinstanzen dienen zum einen Random-Instanzen, die ebenfalls vom Institut zur Verfügung gestellt wurden, zum anderen euklidische Instanzen, die auch in [12] verwendet wurden.

Die Random-Instanzen (R) werden in zwei Kategorien unterteilt. Bei den Instanzen mit 500 und 1000 Knoten handelt es sich um vollständige Graphen, während bei den Instanzen mit 5000, 10000 und 50000 Knoten kein vollständiger Graph vorliegt. Für diese größeren Instanzen gilt ein Verhältnis von Knoten zu Kanten von 1:10. Die Kosten und Delays der Kanten entsprechen zufälligen Werten aus dem Intervall $[1; 99]$. Die Instanzen wurden zufällig erzeugt, wobei für die 500er und 1000er Instanzen je 30 Instanzen und für die 5000er, 10000er und 50000er Instanzen je fünf Instanzen erzeugt wurden. Es ist jedoch anzumerken, dass die Instanzen dadurch nicht an praktischer Relevanz verlieren. Denkt man beispielsweise an das Beispiel des Transportunternehmens, so kann man sich leicht Szenarien überlegen, in denen der Graph zumindest teilweise einem Randomgraphen gleicht. Unterschiedliche Länder, Spediteure oder auch Transportmethoden könnten Gründe hierfür sein.

Bei dem zweiten Instanztyp, den euklidischen Instanzen, repräsentieren die Knoten Koordinaten in einem Einheitsquadrat. Die Kanten stellen die Verbindungen zwischen diesen Koordinaten dar und deren Kosten entsprechen der Distanz mit Faktor 10^7 skaliert. Auch bei den euklidischen Instanzen werden zwei Kategorien unterschieden. Die erste Variante sind die Hop-Constraint (EUH) Instanzen. Wie bereits in Kapitel 3 beschrieben, handelt es sich dabei um Instanzen, bei welchen jede Kante einen Delay von 1 aufweist. Für die zweite Kategorie, die modifizierten, euklidischen Instanzen (EUM), wurde der Delay mittels einer Normalverteilung mit einer Standardabweichung von 10000 ermittelt. Da dadurch der Delaywert auch eine sehr hohe Zahl ist, wird der Delay für EUM Instanzen in den Tabellen in Millionen angegeben. Für alle euklidischen Kategorien und jede Größe stehen je 15 Instanzen zur Verfügung.

Für alle Instanztypen werden die durchschnittlichen Kosten ($\bar{c}$), die Standardabweichung (σ)

und die durchschnittliche Zeit in Sekunden ($\bar{t}[s]$) angegeben. Der Durchschnitt wird hierbei über alle Instanzen des selben Typs der selben Größe gebildet. Weiters wird der Vorgang je einmal ohne und mit Improvement durchgeführt und den Ergebnissen der KBH, mit lokalem Improvement durch Kantenaustausch, gegenübergestellt. Da bei der Degree-based Multilevel-Heuristik und der Ranking-based Multilevel-Heuristik die Superknotenauswahl teilweise zufallsgesteuert ist, sind beide als nicht deterministisch einzustufen. Aus diesem Grund werden in beiden Fällen 30 Durchläufe pro Test durchgeführt um statistische Ausreißer zu vermeiden.

9.2 Degree-based Multilevel-Heuristik

Die Tabellen 9.1, 9.2 und 9.3 zeigen die Testergebnisse für die Degree-based Multilevel-Heuristik (DBMH). Die Tests wurden mit den Superraten 0.25, 0.5 und 0.75 durchgeführt. Man sieht, dass die Ergebnisse ohne Improvement bei höheren Superraten besser sind. Grund hierfür ist, dass bei einer höheren Superrate wesentlich mehr Superknoten zur Verfügung stehen und somit mehr Möglichkeiten überprüft werden. Allerdings ist auch zu sehen, dass höhere Superraten generell schlechtere Ergebnisse liefern, wenn Improvement im Refinement erfolgt. Begründet wird dies damit, dass die während des Coarsenings erzeugten Anfangslösungen für das Improvement den Delay-Constraint wesentlich stärker ausreizen, da mehr Möglichkeiten überprüft werden. Dies führt dazu, dass für das Improvement wesentlich weniger Möglichkeiten zur Verfügung stehen. Im Allgemeinen werden die besten Ergebnisse also bei niedriger Superrate mit zusätzlichem Improvement erzeugt. Auch in Bezug auf die Laufzeit sind geringere Superraten im Vorteil, da im Allgemeinen weniger Level erzeugt werden bzw. weniger Durchläufe während des Coarsenings erfolgen.

Im Vergleich zur KBH sind die Ergebnisse eher schlechter. Für kleinere Random-Instanzen, vor allem mit geringerer Delaygrenze, liefert die DBMH vergleichbare, teilweise bessere, Ergebnisse. Vor allem bei den größeren Random-Instanzen ist die DBMH jedoch von der Laufzeit her unterlegen. Interessant ist jedoch, dass die DBMH, vor allem bei niedrigen Delaygrenzen, eine deutlich geringere Standardabweichung aufweist. Dies ist ein Indiz dafür, dass die DBMH robuster bezüglich unterschiedlicher Testinstanzen ist. Geringfügige Unterschiede bei den Kosten- und Delaywerten der Testinstanzen bewirken also eine weniger starke Streuung bei den Ergebnissen.

Bei den EUH-Instanzen zeigt sich die DBMH bei geringer Delaygrenze überlegen, jedoch liefert sie auch hier bei höheren Delaygrenzen deutlich schlechtere Ergebnisse als die KBH. Bei den EUM-Instanzen schneidet die DBMH allerdings im Vergleich zur KBH sehr schlecht ab. Ein Grund hierfür könnte sein, dass der Delay einer Kante nicht in die Entscheidungen während des Coarsenings einfließt und somit eine schlechte Startlösung für das Improvement erzeugt wird. Weiters muss man auch beachten, dass die DBMH bei vollständigen Graphen prinzipiell, zumindest im ersten Level, zufällige Knoten als Superknoten auswählt. Das Grundprinzip der Knotenauswahl der DBMH wird also von den vollständigen Graphen ad absurdum geführt. Weiters lässt sich bei den euklidischen Instanzen beobachten, dass mit wachsender Delaygrenze die Standardabweichung der Ergebnisse bei der DBMH weniger stark zurückgeht als bei der KBH. Die KBH zeigt sich also bei den euklidischen Instanzen robuster.

Generell ist die Laufzeit der DBMH jedoch für die euklidischen Instanzen, also EUH und EUM, geringer als bei der KBH. Mögliche Gründe könnten schlechte Startlösungen sein, die weniger Improvement zulassen, oder auch, dass die DBMH mit kleineren Instanzen besser zurecht

kommt. Eine Möglichkeit wäre auch, dass die DBMH bei vollständigen Graphen schneller ist als die KBH. Diese Theorie wäre auch durch die Ergebnisse für die kleineren Random-Instanzen unterstützt, da die Instanzen mit 500 und 1000 Knoten prinzipiell vollständige Graphen sind. Allerdings ist hier zu beachten, dass durch das Preprocessing, vor allem bei niedrigen Delaygrenzen, bereits sehr viele Kanten wegfallen.

Generell ist die DBMH der KBH jedoch auf die Lösungsqualität bezogen unterlegen, wenn die Delaygrenze vergleichsweise hoch angelegt ist. Man sieht deutlich, dass bei allen Instanzen die Ergebnisse bei wachsender Delaygrenze im Vergleich zur KBH schlechter werden. Der Hauptgrund dafür wird darin vermutet, dass bei der DBMH der Delay nicht in die Entscheidungen während des Coarsenings, mit der Ausnahme der Überprüfung ob der Delay-Constraint verletzt wird, einfließt und gleichzeitig immer nur die Superknoten als Anschlussmöglichkeiten zur Verfügung stehen. Ein Versuch den Delay in den Entscheidungsprozess einzubauen wurde, wie bereits in Kapitel 6 erwähnt, mit der Ranking-based Multilevel-Heuristik (RBMH) unternommen.

DBMH - SR 0.25		**ohne Improvement**			**mit Improvement**			**KBH**		
Instanz	**B**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**
R 500	10	11498	453	**0,04**	**7218**	328	0,26	11093	2615	0,27
	25	7718	383	**0,07**	**4525**	317	0,64	5099	765	0,75
	50	6102	382	**0,13**	3512	296	1,31	**3007**	221	1,56
R 1000	10	17459	644	**0,15**	**10672**	1362	1,17	12683	2391	1,17
	50	9788	429	**0,49**	5665	348	6,27	**4449**	454	7,61
	100	7927	551	**0,91**	4767	338	11,18	**2075**	277	15,38
R 5000	100	181053	2846	**2,61**	**119406**	2807	5,08	178947	10258	2,95
	200	126740	1451	**1,32**	**71595**	1004	4,30	96950	1965	2,84
	300	117324	1375	**0,87**	**59054**	901	3,86	68102	1339	2,61
R 10000	100	379204	10748	12,17	**252776**	9094	19,51	403274	34318	**7,88**
	200	258395	3488	**5,71**	**145658**	3132	14,12	199483	5177	7,83
	300	236620	1884	**3,42**	**118429**	1535	13,81	140679	2860	6,82
R 50000	250	1229461	15658	119,27	**665792**	16004	266,98	913686	57630	**82,58**
	500	1152276	2761	**66,87**	546390	3169	247,61	**504626**	13034	85,00
EUH 500	5	345955053	15449539	**0,22**	**252733326**	10278169	2,23	505484492	63046580	2,28
	25	341936746	15221231	**0,23**	210284568	4496964	3,25	**172345504**	4597137	4,38
	50	341759293	14596509	**0,23**	209866988	4253567	3,41	**156214042**	3658226	5,85
EUH 1000	10	485265819	12097646	**0,88**	**303746005**	3731068	15,30	423169170	37969089	12,41
	50	487499481	17232580	**0,87**	299930822	4450724	16,62	**225429996**	2721578	28,65
	100	485234986	11586630	**0,87**	300325974	4167389	16,08	**214219403**	3039304	42,17
EUM 500	20M	332416667	10700920	**0,24**	184880104	3716907	3,07	**159749339**	12043599	6,79
	30M	333369094	15269483	**0,23**	184042566	2822912	2,97	**148939912**	3206474	7,56
	40M	334586642	11044145	**0,24**	184431032	2950844	2,91	**147470508**	2597686	8,04
EUM 1000	20M	485469454	11390428	**0,93**	264541488	3438115	14,94	**223545135**	9934603	44,44
	30M	482074790	12742608	**0,91**	263576951	3661885	14,17	**210955939**	2923102	47,23
	40M	483532683	12957501	**0,90**	263648528	3139409	14,97	**209121906**	1748267	50,33

Tabelle 9.1: Degree-based Multilevel Heuristik, Superrate 0.25

DBMH - SR 0.5		**ohne Improvement**			**mit Improvement**			**KBH**		
Instanz	**B**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**
R 500	10	11141	452	**0,05**	**7729**	408	0,29	11093	2615	0,27
	25	7836	429	**0,08**	5115	492	0,73	**5099**	765	0,75
	50	6294	416	**0,14**	4066	298	1,45	**3007**	221	1,56
R 1000	10	16994	693	**0,16**	**11422**	503	1,31	12683	2391	1,17
	50	10243	528	**0,53**	6540	652	6,53	**4449**	454	7,61
	100	8439	516	**0,97**	5480	387	13,13	**2075**	277	15,38
R 5000	100	162267	3693	**2,79**	**121697**	3010	5,36	178947	10258	2,95
	200	108349	1847	**1,70**	**73464**	1286	4,20	96950	1965	2,84
	300	89649	1332	**1,07**	**58065**	870	3,98	68102	1339	2,61
R 10000	100	339143	8469	13,32	**258257**	8052	20,29	403274	34318	**7,88**
	200	220422	2143	**7,43**	**150725**	2406	15,56	199483	5177	7,83
	300	182656	1558	**4,57**	**117086**	1393	13,71	140679	2860	6,82
R 50000	250	1016337	24727	156,65	**680958**	20884	276,44	913686	57630	**82,58**
	500	806436	4432	**74,88**	515034	2865	206,79	**504626**	13034	85,00
EUH 500	5	305589217	13733593	**0,24**	**254391744**	9603253	2,58	505484492	63046580	2,28
	25	272165450	8748808	**0,24**	210955018	4569534	2,96	**172345504**	4597137	4,38
	50	272442041	6958637	**0,24**	209156121	4706560	3,42	**156214042**	3658226	5,85
EUH 1000	10	390212799	8088856	**0,93**	**302837212**	3515415	15,96	423169170	37969089	12,41
	50	388376837	7689906	**0,93**	299373552	4399323	16,59	**225429996**	2721578	28,65
	100	390396159	8694704	**0,94**	298591385	4407618	17,03	**214219403**	3039304	42,17
EUM 500	20M	271709201	7849998	**0,25**	186925604	3269352	3,29	**159749339**	12043599	6,79
	30M	270888687	6284901	**0,25**	186855922	3668304	3,15	**148939912**	3206474	7,56
	40M	272989302	8854358	**0,25**	186727685	3969087	3,33	**147470508**	2597686	8,04
EUM 1000	20M	387886697	7769524	**0,95**	265093375	3212414	16,83	**223545135**	9934603	44,44
	30M	383436901	8057646	**0,96**	266255879	3062792	17,23	**210955939**	2923102	47,23
	40M	389228341	7787164	**0,96**	265645882	3911860	17,48	**209121906**	1748267	50,33

Tabelle 9.2: Degree-based Multilevel Heuristik, Superrate 0.5

DBMH - SR 0.75		**ohne Improvement**			**mit Improvement**			**KBH**		
Instanz	**B**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**
R 500	10	10896	551	**0,06**	**7983**	441	0,43	11093	2615	0,27
	25	7830	471	**0,10**	5419	416	1,08	**5099**	765	0,75
	50	6444	458	**0,17**	3937	449	2,12	**3007**	221	1,56
R 1000	10	16961	797	**0,21**	**11886**	580	1,98	12683	2391	1,17
	50	10299	518	**0,63**	6986	381	10,50	**4449**	454	7,61
	100	8545	468	**1,13**	5833	432	19,77	**2075**	277	15,38
R 5000	100	154957	2640	3,50	**122934**	2532	6,90	178947	10258	**2,95**
	200	102302	1834	**2,31**	**76502**	659	5,98	96950	1965	2,84
	300	81755	1432	**1,73**	**58989**	845	5,41	68102	1339	2,61
R 10000	100	324303	8940	18,02	**259622**	8412	27,62	403274	34318	**7,88**
	200	211666	1895	11,69	**156638**	2417	21,85	199483	5177	**7,83**
	300	167328	2193	8,51	**120958**	1292	18,60	140679	2860	**6,82**
R 50000	250	965076	25206	270,21	**703995**	22049	395,22	913686	57630	**82,58**
	500	688955	4607	170,91	506833	3198	293,22	**504626**	13034	**85,00**
EUH 500	5	332639407	22170306	**0,28**	**288632951**	17165365	3,61	505484492	63046580	2,28
	25	240725094	7231927	**0,28**	210563931	4979967	4,08	**172345504**	4597137	4,38
	50	240458985	6097056	**0,27**	208798767	4477418	4,35	**156214042**	3658226	5,85
EUH 1000	10	351788953	5535982	**1,08**	**304529865**	4770564	21,53	423169170	37969089	12,41
	50	346248626	5630546	**1,08**	300118784	4603969	22,43	**225429996**	2721578	28,65
	100	344847209	6231467	**1,08**	298309232	5578106	23,31	**214219403**	3039304	42,17
EUM 500	20M	242352945	5062521	**0,29**	188258787	3845255	4,56	**159749339**	12043599	6,79
	30M	242556720	4933040	**0,29**	188995666	4435995	4,70	**148939912**	3206474	7,56
	40M	240943467	6517110	**0,28**	188664871	3911809	4,58	**147470508**	2597686	8,04
EUM 1000	20M	346139145	6484424	**1,09**	267239437	3611057	25,14	**223545135**	9934603	44,44
	30M	348507924	6073429	**1,10**	267233726	3602177	25,02	**210955939**	2923102	47,23
	40M	346628355	8228833	**1,10**	269120060	3974915	23,78	**209121906**	1748267	50,33

Tabelle 9.3: Degree-based Multilevel Heuristik, Superrate 0.75

9.3 Ranking-based Multilevel-Heuristik

Die Ergebnisse der RMBH mit verschiedenen Superraten finden sich in den Tabellen 9.4, 9.5 und 9.6. Auch hier wurden die Superraten 0.25, 0.5 und 0.75 getestet. Ähnlich wie bei der DBMH werden die Ergebnisse ohne Improvement mit steigender Superrate besser, wobei auch hier die Laufzeit ansteigt. Im Unterschied zur DBMH sind bei der RBMH jedoch, im Allgemeinen, auch die Ergebnisse mit Improvement für höhere Superraten besser. Der wesentliche Unterschied zur DBMH besteht jedoch in der Lösungsqualität, auch im Vergleich zur KBH. Speziell bei den großen Random-Instanzen liefert die RBMH relativ schlechte Ergebnisse, wenn kein Improvement durchgeführt wird. Dies ist darauf zurückzuführen, dass während des Coarsenings stark auf den Delay geachtet wird und eine Lösung erzeugt wird, welche mehr Raum für Improvement bieten soll. Dass diese Überlegung auch zutrifft, zeigt sich in den Ergebnissen mit Improvement. Anzumerken ist noch, dass die RBMH zwar nicht deterministisch ist, da ein Zufallselement bei der Knotenauswahl vorkommt. Jedoch wird lediglich dann eine Zufallsauswahl getroffen, wenn zwei Knoten den gleichen Score aufweisen, was relativ selten vorkommt.

Bei den Random-Instanzen liefert die RBMH im Vergleich zur KBH immer bessere Ergebnisse. Eine höhere Delaygrenze scheint, anders als bei der DBMH keine deutlich erkennbare negative Auswirkung auf die RBMH zu haben. Die Lösungsqualität hat jedoch ihren Preis in Form der Laufzeit. Diese ist im Vergleich zur KBH doch deutlich höher. Hauptgrund dafür ist, dass die RBMH eine Startlösung für das Improvement liefert, welche viel Raum für Verbesserungen lässt. Dies führt natürlich auch dazu, dass sehr viele Improvements durchgeführt werden, was zu einer deutlich höheren Laufzeit führt. Ähnlich wie bei der DBMH ist auch die Standardabweichung der Ergebnisse bei RBMH im Bereich der Random-Instanzen wesentlich geringer als die der KBH.

Ein Problem für die RBMH stellen die euklidischen Instanzen dar. Die EUH-Instanzen führen das Prinzip der RBMH ad absurdum, da hier der Delay aller Kanten 1 beträgt. Dies führt zu einem Ranking, welches lediglich die Kosten miteinbezieht. Die Superknotenentscheidung wird somit darauf reduziert, dass jene Knoten verwendet werden, zu denen die günstigsten Kanten verlaufen. Diese Ausartung in Richtung Greedy-Auswahl ist jedoch der zufälligen Auswahl der DBMH, speziell bei vollständigen Graphen, überlegen. Im Vergleich zur KBH kann die RBMH aber bei sehr niedriger Delaygrenze für EUH-Instanzen immer noch gute Ergebnisse erzielen, bei höheren Delaygrenzen ist die KBH allerdings deutlich besser. Bei EUM-Instanzen versagt die RBMH im Vergleich zur KBH allerdings. Ein Grund hierfür könnte sein, dass der Delay der Kanten, bis auf die Abweichung aus der Normalverteilung, sehr dem der Kosten ähnelt. Dies scheint zu ähnlichen Schwierigkeiten zu führen wie schon bei den EUH-Instanzen, da auch hier wieder günstige Kanten, bzw. Knoten zu denen diese verlaufen, bevorzugt werden, was auch wieder einer Greedy-Auswahl ähnelt. Während die Ergebnisse generell schlechter sind, ist die Laufzeit der RBMH auch bei den euklidischen Instanzen deutlich höher als jene der KBH. Auch die Standardabweichung der Ergebnisse ist im Allgemeinen höher als bei der KBH.

RBMH - SR 0.25		**ohne Improvement**			**mit Improvement**			**KBH**		
Instanz	**B**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**
R 500	10	11164	456	0,29	**6155**	328	0,51	11093	2615	**0,27**
	25	7032	267	**0,75**	**2796**	137	1,32	5099	765	**0,75**
	50	5263	183	1,59	**1695**	112	2,82	3007	221	**1,56**
R 1000	10	16208	514	1,28	**7845**	429	2,29	12683	2391	**1,17**
	50	7603	220	**7,52**	**2215**	139	12,98	4449	454	7,61
	100	5684	188	16,04	**1527**	91	27,47	2075	277	**15,38**
R 5000	100	168730	3563	3,78	**112246**	3839	6,55	178947	10258	**2,95**
	200	140825	1952	**2,66**	**65814**	939	6,13	96950	1965	2,84
	300	138981	1774	**2,52**	**57416**	390	6,24	68102	1339	2,61
R 10000	100	351992	9592	15,10	**240723**	10461	22,29	403274	34318	**7,88**
	200	281598	1847	9,23	**133444**	1799	20,17	199483	5177	**7,83**
	300	279198	2408	8,32	**114338**	1250	20,12	140679	2860	**6,82**
R 50000	250	1403887	3616	222,45	**619496**	13024	396,73	913686	57630	**82,58**
	500	1389305	1952	203,55	541183	5573	413,19	**504626**	13034	**85,00**
EUH 500	5	336279582	11124943	3,21	**253712486**	12261015	5,21	505484492	63046580	**2,28**
	25	336279582	11124943	**3,20**	206496557	3606008	6,60	**172345504**	4597137	4,38
	50	336279582	11124943	**3,20**	206496557	3606008	6,66	**156214042**	3658226	5,85
EUH 1000	10	486091440	10471966	15,18	**300813360**	5523776	29,94	423169170	37969089	**12,41**
	50	486091440	10471966	**15,21**	296434016	5676657	29,37	**225429996**	2721578	28,65
	100	486091440	10471966	**15,11**	296434016	5676657	29,17	**214219403**	3039304	42,17
EUM 500	20M	791534886	28768749	**3,15**	178330787	3390072	6,31	**159749339**	12043599	6,79
	30M	791534886	28768749	**3,17**	178330787	3390072	6,10	**148939912**	3206474	7,56
	40M	791534886	28768749	**3,15**	178330787	3390072	6,14	**147470508**	2597686	8,04
EUM 1000	20M	1497392812	55529058	**14,94**	255772899	2293475	31,42	**223545135**	9934603	44,44
	30M	1497392812	55529058	**14,82**	255772899	2293475	31,33	**210955939**	2923102	47,23
	40M	1497392812	55529058	**14,98**	255772899	2293475	31,54	**209121906**	1748267	50,33

Tabelle 9.4: Ranking-based Multilevel Heuristik, Superrate 0.25

RBMH - SR 0.5		**ohne Improvement**			**mit Improvement**			**KBH**		
Instanz	**B**	$\bar{c}$	σ	**t[s]**	$\bar{c}$	σ	**t[s]**	$\bar{c}$	σ	**t[s]**
R 500	10	9745	357	0,32	**6041**	261	0,57	11093	2615	**0,27**
	25	5727	196	0,80	**2640**	151	1,49	5099	765	**0,75**
	50	4100	146	1,69	**1550**	123	3,15	3007	221	**1,56**
R 1000	10	13882	496	1,36	**7605**	381	2,63	12683	2391	**1,17**
	50	6075	180	7,96	**2007**	130	14,84	4449	454	**7,61**
	100	4572	111	16,93	**1426**	80	30,43	2075	277	**15,38**
R 5000	100	148381	2661	4,09	**111026**	2872	6,82	178947	10258	**2,95**
	200	115276	1046	**2,83**	**61667**	850	6,36	96950	1965	2,84
	300	110616	1249	**2,60**	**52530**	477	6,53	68102	1339	2,61
R 10000	100	311737	8883	15,98	**238295**	9707	23,84	403274	34318	**7,88**
	200	229890	1954	9,98	**124999**	1521	21,11	199483	5177	**7,83**
	300	220784	2486	9,04	**105384**	815	20,71	140679	2860	**6,82**
R 50000	250	1129992	7203	244,45	**576934**	13902	418,01	913686	57630	**82,58**
	500	1097613	2843	233,27	**500864**	2769	421,45	504626	13034	**85,00**
EUH 500	5	303790168	11273859	3,38	**255263263**	11028524	5,80	505484492	63046580	**2,28**
	25	272175632	6657184	**3,37**	205136872	3897479	6,71	**172345504**	4597137	4,38
	50	272175632	6657184	**3,40**	205136872	3897479	6,63	**156214042**	3658226	5,85
EUH 1000	10	388186444	9118427	16,05	**299549626**	4828223	31,41	423169170	37969089	**12,41**
	50	388186444	9118427	**16,09**	293818248	4787136	32,67	**225429996**	2721578	28,65
	100	388186444	9118427	**16,22**	293818248	4787136	32,67	**214219403**	3039304	42,17
EUM 500	20M	483752885	19212784	**3,36**	178574928	3082106	6,74	**159749339**	12043599	6,79
	30M	483752885	19212784	**3,32**	178574928	3082106	6,65	**148939912**	3206474	7,56
	40M	483752885	19212784	**3,34**	178574928	3082106	6,63	**147470508**	2597686	8,04
EUM 1000	20M	888653452	30305239	**15,77**	256404678	2693250	32,72	**223545135**	9934603	44,44
	30M	888653452	30305239	**15,79**	256404678	2693250	32,74	**210955939**	2923102	47,23
	40M	888653452	30305239	**15,72**	256404678	2693250	33,02	**209121906**	1748267	50,33

Tabelle 9.5: Ranking-based Multilevel Heuristik, Superrate 0.5

RBMH - SR 0.75		**ohne Improvement**			**mit Improvement**			**KBH**		
Instanz	**B**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**
R 500	10	9282	415	0,38	**6064**	316	0,77	11093	2615	**0,27**
	25	5381	294	0,95	**2633**	201	2,03	5099	765	**0,75**
	50	3711	161	1,99	**1471**	85	4,35	3007	221	**1,56**
R 1000	10	13288	593	1,64	**7581**	368	3,67	12683	2391	**1,17**
	50	5513	174	9,40	**1977**	114	20,93	4449	454	**7,61**
	100	4180	128	19,48	**1405**	105	40,19	2075	277	**15,38**
R 5000	100	141311	2509	5,10	**111515**	3331	9,16	178947	10258	**2,95**
	200	105109	1231	3,80	**60741**	851	8,56	96950	1965	**2,84**
	300	98429	1243	3,48	**50272**	284	8,70	68102	1339	**2,61**
R 10000	100	295842	9160	21,04	**238417**	10822	30,91	403274	34318	**7,88**
	200	210136	2351	14,22	**122801**	2720	27,22	199483	5177	**7,83**
	300	196465	2083	12,42	**100999**	542	27,19	140679	2860	**6,82**
R 50000	250	1020284	8777	378,29	**562509**	15021	560,02	913686	57630	**82,58**
	500	966673	2749	362,61	**476838**	1857	566,73	504626	13034	**85,00**
EUH 500	5	335585492	25440609	3,94	**292254854**	20020821	7,51	505484492	63046580	**2,28**
	25	241655999	3663911	**3,94**	204951634	4289561	8,40	**172345504**	4597137	4,38
	50	241655999	3663911	**3,94**	204951634	4289561	8,49	**156214042**	3658226	5,85
EUH 1000	10	353314426	8856501	19,08	**299588651**	5014246	39,00	423169170	37969089	**12,41**
	50	348760164	7165210	**19,06**	291284178	4542205	41,72	**225429996**	2721578	28,65
	100	348760164	7165210	**19,06**	291284178	4542205	41,38	**214219403**	3039304	42,17
EUM 500	20M	311326252	8456623	**3,90**	174184845	2629833	9,10	**159749339**	12043599	6,79
	30M	311326252	8456623	**3,91**	174184845	2629833	9,15	**148939912**	3206474	7,56
	40M	311326252	8456623	**3,91**	174184845	2629833	9,14	**147470508**	2597686	8,04
EUM 1000	20M	521471434	11850162	**19,08**	253563575	2086372	45,04	**223545135**	9934603	44,44
	30M	521471434	11850162	**18,93**	253563575	2086372	45,44	**210955939**	2923102	47,23
	40M	521471434	11850162	**18,69**	253563575	2086372	45,62	**209121906**	1748267	50,33

Tabelle 9.6: Ranking-based Multilevel Heuristik, Superrate 0.75

9.4 Component-based Multilevel-Heuristik

Anders als bei der DBMH und RBMH wird bei der Component-based Multilevel-Heuristik (CBMH) weder ein Zufallselement verwendet, es handelt sich also um einen deterministischen Algorithmus, noch muss ein Parameter wie die Superrate gewählt werden. Die Ergebnisse für die CBMH finden sich in Tabelle 9.7.

Bei den Random-Instanzen ist die CBMH der KBH sowohl in Lösungsqualität als auch in Laufzeit überlegen. Speziell bei sehr niedrigen Delaygrenzen fällt die Überlegenheit der CBMH relativ deutlich aus. Was hier noch hervorzuheben ist, ist die Tatsache, dass die CBMH bei den Random-Instanzen selbst ohne zusätzliches Improvement großteils bessere Ergebnisse als die KBH mit Improvement liefert. Anders sieht es bei den euklidischen Instanzen aus. Sowohl bei den EUH als auch den EUM-Instanzen ist die KBH, mit Ausnahme bei niedriger Delaygrenze, bezüglich der Lösungsqualität überlegen. Die Laufzeit ist jedoch, mit Ausnahme bei den EUH-Instanzen mit niedriger Delaygrenze, bei der CBMH niedriger. Grund hierfür könnte sein, dass das vorsichtigere Verschmelzen bei der CBMH mit dem Hop-Constraint bzw. den euklidischen Delays schlechter zurecht kommt. Warum dies eintritt, ist nicht sicher, jedoch liegt eine Vermutung darin, dass hier durch die Tatsache, dass die CBMH mehr delaygesteuert als die KBH ist, einige Fehlentscheidungen während des Coarsenings getroffen werden. Für die Standardabweichung der Ergebnisse zeigt sich hier ein ähnliches Bild wie bei den zuvor diskutierten Heuristiken. Bei den Random-Instanzen ist die CBMH der KBH überlegen, jedoch ist auch hier die Standardabweichung der Ergebnisse bei den euklidischen Instanzen, zumindest bei höheren Delaygrenzen, höher.

Im Allgemeinen zeigt sich die CBMH allerdings als eine sehr gute und vor allem schnelle Heuristik. Die Laufzeit ist, speziell bei den größeren Random-Instanzen, deutlich geringer als bei den anderen Verfahren, während die Lösungsqualität im Bereich der KBH liegt und bei den Random-Instanzen teilweise sogar ohne zusätzliches Improvement bessere Ergebnisse erzielt werden.

CBMH		**ohne Improvement**			**mit Improvement**			**KBH**		
Instanz	**B**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**
R 500	10	8085	394	**0,08**	**7515**	391	0,20	11093	2615	0,27
	25	4952	331	**0,20**	**4427**	275	0,57	5099	765	0,75
	50	3375	284	**0,42**	**2969**	240	1,18	3007	221	1,56
R 1000	10	11769	593	**0,34**	**10769**	466	0,98	12683	2391	1,17
	50	4311	368	**1,93**	**3873**	292	5,53	4449	454	7,61
	100	2521	180	**3,98**	2296	150	11,73	**2075**	277	15,38
R 5000	100	125107	3233	**0,46**	**118660**	2697	1,44	178947	10258	2,95
	200	79594	2078	**0,44**	**72571**	1009	1,55	96950	1965	2,84
	300	62895	880	**0,44**	**57775**	677	1,59	68102	1339	2,61
R 10000	100	262769	9457	**1,24**	**249497**	9342	3,67	403274	34318	7,88
	200	162789	1868	**1,17**	**148517**	1854	4,32	199483	5177	7,83
	300	128020	1071	**1,17**	**116720**	1170	4,32	140679	2860	6,82
R 50000	250	735644	23263	**6,67**	**673614**	23182	40,46	913686	57630	82,58
	500	523316	3864	**8,28**	**482970**	3473	40,25	504626	13034	85,00
EUH 500	5	253498254	7059986	**0,69**	**240669620**	7278938	2,58	505484492	63046580	2,28
	25	211206540	4117753	**0,69**	188958839	4486080	3,29	**172345504**	4597137	4,38
	50	208819295	4323813	**0,70**	186462657	3561965	3,12	**156214042**	3658226	5,85
EUH 1000	10	343455988	3600177	**3,34**	**318864486**	3879330	13,18	423169170	37969089	12,41
	50	301960034	4232189	**3,14**	269348994	4149767	15,64	**225429996**	2721578	28,65
	100	301952412	4235131	**3,21**	269395723	4108097	15,80	**214219403**	3039304	42,17
EUM 500	20M	158923933	2210991	**0,70**	**157739085**	2253126	2,05	159749339	12043599	6,79
	30M	158619635	2373546	**0,70**	157450261	2382908	2,19	**148939912**	3206474	7,56
	40M	158619635	2373546	**0,71**	157450261	2382908	2,09	**147470508**	2597686	8,04
EUM 1000	20M	226789042	2327111	**3,32**	224883530	1986942	11,65	**223545135**	9934603	44,44
	30M	226752323	2309018	**3,45**	224846198	1968713	11,01	**210955939**	2923102	47,23
	40M	226752323	2309018	**3,41**	224846198	1968713	11,32	**209121906**	1748267	50,33

Tabelle 9.7: Component-based Multilevel Heuristik

9.5 Vergleich der Heuristiken

In Tabelle 9.8 finden sich die drei entwickelten Verfahren direkt gegenübergestellt. Für DBMH und RBMH wurde, für jede Instanz bzw. Delaygrenze, das Testergebnis aus den drei verfügbaren Superraten ausgewählt, welches die geringsten Kosten aufweist. Man sieht, dass bei den Random-Instanzen klar die RBMH die besten Ergebnisse liefert. Die DBMH bildet hier mit einigen Ausnahmen das Schlusslicht. Bei den euklidischen Instanzen liefert die CBMH die besten Ergebnisse, mit Ausnahme der EUH-1000-Instanz bei Delaygrenze 10. Hier ist die CBMH deutlich schlechter als die anderen beiden Verfahren, wobei es sich wohl um einen Ausreißer handelt.

Auffällig ist, dass alle drei Verfahren Probleme mit den euklidischen, speziell mit den EUM-Instanzen haben. Speziell die RBMH scheint sich schnell in ein lokales Optimum zu verlaufen und kann aus unerklärlichen Gründen keine besseren Ergebnisse bei höherer Delaygrenze erzielen. Ähnlich geht es der CBMH wobei hier noch ein Unterschied zwischen Delaygrenze 20M und 30M erkennbar ist. Bei der DBMH sieht man, dass die stark zufällige Auswahl zu fragwürdigen Ergebnissen führt, da hier eine höhere Delaygrenze schlechtere Ergebnisse beschert. Im Allgemeinen kann man sagen, dass keines der drei Verfahren sonderlich gut mit den euklidischen Instanzen zurechtkommt. Dies liegt jedoch vermutlich eher an der Beschaffenheit der Delays als an der Tatsache, dass es sich um euklidische Instanzen handelt.

Was die Laufzeit betrifft ist die CBMH klarer Sieger, vor allem bei den großen Random-Instanzen. Klar abgeschlagen in Sachen Laufzeit ist die RBMH, wobei dies wieder darauf zurückzuführen ist, dass hier mehr Improvement durchgeführt wird. Die teilweise weit besseren Ergebnisse der RBMH rechtfertigen jedoch die höhere Laufzeit.

Allgemein kann man sagen, dass die RBMH die besten Lösungen liefert, jedoch auch die höchste Laufzeit mit sich bringt. Die CBMH produziert schnell vergleichsweise gute Lösungen und stellt eine gute Alternative zur KBH dar. Die DBMH ist verstärkt zufallgesteuert und nutzt weniger Informationen und dementsprechend sind auch die Ergebnisse teilweise vergleichsweise gut, teilweise weit schlechter als bei den anderen Verfahren, wobei sie im Vergleich zur RBMH recht schnell ist. Generell kann man sagen, dass die RBMH und die CBMH die eher erfolgreichen Verfahren sind, wobei die DBMH auch eine Grundlage für die RBMH darstellt.

		DBMH			RBMH			CBMH		
Instanz	**B**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**	$\bar{\mathbf{c}}$	σ	**t[s]**
R 500	10	7218	328	0,26	**6041**	261	0,57	7515	391	**0,20**
	25	4525	317	0,64	**2633**	201	2,03	4427	275	**0,57**
	50	3512	296	1,31	**1471**	85	4,35	2969	240	**1,18**
R 1000	10	10672	1362	1,17	**7581**	368	3,67	10769	466	**0,98**
	50	5665	348	6,27	**1977**	114	20,93	3873	292	**5,53**
	100	4767	338	**11,18**	**1405**	105	40,19	2296	150	11,73
R 5000	100	119406	2807	5,08	**111026**	2872	6,82	118660	2697	**1,44**
	200	71595	1004	4,30	**60741**	851	8,56	72571	1009	**1,55**
	300	58989	845	5,41	**50272**	284	8,70	57775	677	**1,59**
R 10000	100	252776	9094	19,51	**238295**	9707	23,84	249497	9342	**3,67**
	200	145658	3132	14,12	**122801**	2720	27,22	148517	1854	**4,32**
	300	117086	1393	13,71	**100999**	542	27,19	116720	1170	**4,32**
R 50000	250	665792	16004	266,98	**562509**	15021	560,02	673614	23182	**40,46**
	500	506833	3198	293,22	**476838**	1857	566,73	482970	3473	**40,25**
EUH 500	5	252733326	10278169	**2,23**	253712486	12261015	5,21	**240669620**	7278938	2,58
	25	210284568	4496964	**3,25**	204951634	4289561	8,40	**188958839**	4486080	3,29
	50	208798767	4477418	4,35	204951634	4289561	8,49	**186462657**	3561965	**3,12**
EUH 1000	10	302837212	3515415	15,96	**299549626**	4828223	31,41	318864486	3879330	**13,18**
	50	299373552	4399323	16,59	291284178	4542205	41,72	**269348994**	4149767	**15,64**
	100	298309232	5578106	23,31	291284178	4542205	41,38	**269395723**	4108097	**15,80**
EUM 500	20M	184880104	3716907	3,07	174184845	2629833	9,10	**157739085**	2253126	**2,05**
	30M	184042566	2822912	2,97	174184845	2629833	9,15	**157450261**	2382908	**2,19**
	40M	184431032	2950844	2,91	174184845	2629833	9,14	**157450261**	2382908	**2,09**
EUM 1000	20M	264541488	3438115	14,94	253563575	2086372	45,04	**224883530**	1986942	**11,65**
	30M	263576951	3661885	14,17	253563575	2086372	45,44	**224846198**	1968713	**11,01**
	40M	263648528	3139409	14,97	253563575	2086372	45,62	**224846198**	1968713	**11,32**

Tabelle 9.8: Vergleich der drei Verfahren

10 Zusammenfassung und Ausblick

Im Zuge dieser Arbeit wurde nach Multilevel-Heuristiken für das Rooted Delay-Constrained Minimum Spanning Tree Problem gesucht. Eine Multilevel-Heuristik besteht immer aus zwei Teilen, dem Coarsening und dem Refinement. Während des Coarsenings werden die Graphen sukzessive vereinfacht indem Knoten zusammengefasst bzw. Kanten verschmolzen werden. Anschließend wird während des Refinements der Graph schrittweise wiederhergestellt. Dadurch kann der Graph in verschiedenen Detaillevels betrachtet und bearbeitet werden. Nach Analyse der Problemstellung wurden entsprechende Überlegungen angestellt und erläutert. Aus diesen Überlegungen entstanden drei Multilevel-Heuristiken, welche auf verschiedenen Vorgangsweisen basieren.

Bei der ersten vorgestellten Heuristik handelte es sich um die Degree-based Multilevel-Heuristik. Die grundlegende Vorgangsweise besteht darin, Knoten mit hohem Knotengrad als Superknoten auszuwählen und durch Verbinden der übrigen Knoten des aktuellen Levels Baumstrukturen zu erstellen, wobei hier die Kosten minimal gehalten werden. Der selbe Vorgang wird auf folgende Level angewandt, um eine vollständige Lösung zu erhalten. Das Verfahren liefert vergleichsweise schnell akzeptable Ergebnisse, ist jedoch im Allgemeinen den anderen Verfahren unterlegen.

Aufbauend auf den Erkenntnissen der Degree-based Multilevel-Heuristik wurde eine weitere Heuristik entwickelt. Diese basiert auf der Einstufung der generellen Qualität von Kanten, welche durch ein Ranking berechnet wird. Dies ist auch namensgebend für die Ranking-based Multilevel-Heuristik. Durch Berücksichtigung des Delays bei der Kantenentscheidung eines aktuellen Levels werden mehr Möglichkeiten für spätere Level offen gehalten. Diese vorsichtige Vorgangsweise liefert zwar generell schlechtere Ergebnisse nach dem Coarsening, doch durch zusätzliches lokales Improvement während des Refinements können sehr gute Ergebnisse erzielt werden. Im Allgemeinen ist die Ranking-based Multilevel-Heuristik den anderen Verfahren in Bezug auf die Lösungsqualität überlegen, weist jedoch Probleme bei Spezialfällen und auch eine allgemein höhere Laufzeit auf.

Die dritte und letzte entwickelte Heuristik basiert nicht auf dem Verbinden von Knoten zu zuvor festgelegten Superknoten sondern versucht vielmehr sukzessiv Teillösungen aufzubauen, sogenannte Komponenten. Diese Component-based Multilevel-Heuristik ähnelt stark dem Vorgehen der Kruskal-basierten Heuristik, jedoch unterliegt das Coarsening zusätzlichen Einschränkungen, wodurch eine Levelstruktur entsteht. Diese ist jedoch nicht mit den klassischen Multilevel-Strukturen der anderen Verfahren vergleichbar. Die Component-based Multilevel-Heuristik liefert im Vergleich zu den anderen Verfahren sehr gute Ergebnisse, wobei die Laufzeit geringer als die der anderen Verfahren ist.

Das erwähnte lokale Improvement besteht aus dem Austausch von Kanten während des Refinementschritts der Multilevel-Heuristiken. Während aus den aus dem Coarsening erhaltenen Informationen der Lösungsbaum erstellt wird, wird nach möglichen Verbesserungen gesucht. Da Veränderungen der Lösung unmittelbar zu Veränderungen der Delaystruktur führen, müssen nach jedem Austausch Updates mit einer Laufzeit von $\mathcal{O}(|V|^2 * |E|)$ durchgeführt werden. Dies

führt dazu, dass ein Improvement während des Refinements sehr aufwändig sein kann, abhängig von der Qualität der Ausgangslösung.

Es zeigt sich, dass eine klassische Multilevel-Struktur Probleme mit sich bringt. Prinzipiell kann das Multilevel-Prinzip als zusätzliche Einschränkung während des Konstruktion der Lösung gesehen werden. Auch stellt das Verwalten der Delayinformationen in einer Multilevel-Struktur einige Probleme dar. Lokale Entscheidungen in einem Level wirken sich immer auf die gesamte Lösung aus und die Delayinformation muss stets aktuell gehalten werden, um informierte Entscheidungen zu treffen. Durch das Coarsening können Fehlentscheidungen in einem niedrigeren Level weder rückgängig gemacht noch eingesehen werden. Erst durch zusätzliches Improvement konnten gute Ergebnisse erzielt werden, wobei hierfür auch ein entsprechender Aufwand in Kauf genommen werden muss. Bessere Ergebnisse nach dem Coarsening wurden erzielt, indem der Multilevel-Begriff stark gelockert wurde, siehe Component-based Multilevel-Heuristik. Hier bewegt man sich jedoch schon stark an der Grenze zwischen dem Multilevel- und dem Kruskal-Prinzip.

Generell kann man sagen, dass die entwickelten Multilevel-Verfahren keine signifikanten Verbesserungen gegenüber bestehenden Verfahren darstellen. Allerdings hat sich auch gezeigt, dass Multilevel-Verfahren durchaus positive Ergebnisse bringen können und die CBMH, die nicht in die Kategorie klassischer Multilevel-Ansätze fällt, liefert sehr gute Ergebnisse. In zukünftigen Arbeiten sollte versucht werden, durch weitere Relaxierung des Multilevel-Prinzips bessere Konstruktionsheuristiken zu entwickeln. Weiters kann im Bereich des Refinements versucht werden bessere Nachbarschaften zu finden. Alternativ sollte auch eine Multilevel-Heuristik entwickelt werden, die als Verbesserungsheuristik fungiert. Möglicherweise kann durch Multilevel eine bestehende Lösung besser in Teillösungen unterteilt werden und die Lösung so in unterschiedlichen Levels verbessert werden. Ein Ansatz für eine weitere Multilevel-Variante wäre das Iterated Multilevel-Prinzip. Hierbei soll ein Teil einer zuvor durch Coarsening erzeugten Lösung nach dem Refinement durch erneutes Coarsening iterativ verbessert werden. Man könnte gezielt versuchen alternative Operationen während des Coarsenings durchzuführen, um so eine bessere Lösung zu finden, was durch eine Tabusuche geschehen könnte.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Blum, C., Aguilera, M. J. B., Roli, A., and Sampels, M., Eds. *Hybrid Metaheuristics, An Emerging Approach to Optimization*, vol. 114 of *Studies in Computational Intelligence*. Springer, 2008.

[2] Dahl, G., Gouveia, L., and Requejo, C. On formulations and methods for the hop-constrained minimum spanning tree problem. In *Handbook of Optimization in Telecommunications*. Springer Science + Business Media, 2006, ch. 19, pp. 493–515.

[3] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 1 (1959), 269–271.

[4] Dumitrescu, I., and Boland, N. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks 42*, 3 (2003), 135–153.

[5] Fiduccia, C. M., and Mattheyses, R. M. A linear-time heuristic for improving network partitions. In *25 years of DAC: Papers on Twenty-five years of electronic design automation* (New York, NY, USA, 1988), ACM, pp. 241–247.

[6] Friedman, H. Programming pearls, 2nd edition. *Linux J.* (2000), 18–19.

[7] Garey, M. R., and Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[8] Geoffrion, A. Lagrangian relaxation for integer programming. *mathprogstud 2* (1974), 82–114.

[9] Ghaboosi, N., and Haghighat, A. T. A Path Relinking Approach for Delay-Constrained Least-Cost Multicast Routing Problem. In *19th IEEE International Conference on Tools with Artificial Intelligence* (2007), pp. 383–390.

[10] Gouveia, L. Using the Miller-Tucker-Zemlin constraints to formulate a minimal spanning tree problem with hop constraints. *Computers and Operations Research 22*, 9 (1995), 959–970.

[11] Gouveia, L. Using variable redefinition for computing lower bounds for minimum spanning and steiner trees with hop constraints. *INFORMS J. on Computing 10*, 2 (1998), 180–188.

[12] Gouveia, L., Paias, A., and Sharma, D. Modeling and Solving the Rooted Distance-Constrained Minimum Spanning Tree Problem. *Computers and Operations Research 35*, 2 (2008), 600–613.

[13] Hansen, P., and Mladenovic, N. Variable neighborhood search: Principles and applications. *European Journal of Operational Research 130*, 3 (May 2001), 449–467.

[14] Held, M., Wolfe, P., and Crowder, H. P. Validation of subgradient optimization. *mathprog 6*, 1 (1974), 62–88.

[15] Hendrickson, B., and Leland, R. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing* (New York, USA, 1995), ACM, p. 28.

[16] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (1998), 359–392.

[17] Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematics Society 7*, 1 (1956), 48–50.

[18] Leggieri, V., Haouari, M., Layeb, S., and Triki, C. The Steiner Tree Problem with Delays: A Tight Compact Formulation and Reduction Procedures. Tech. rep., Universita del Salento, Lecce, Italy, 2007.

[19] Prim, R. C. Shortest connection networks and some generalizations. *Bell System Technical Journal 36* (1957), 1389–1401.

[20] Qu, R., Xu, Y., and Kendall, G. A Variable Neighborhood Descent Search Algorithm for Delay-Constrained Least-Cost Multicast Routing. In *Proceedings of Learning and Intelligent OptimizatioN (LION3)* (Berlin, Heidelberg, 2009), Springer, pp. 15–29.

[21] Rodney, D., Soper, A., and Walshaw, C. Multilevel Refinement Strategies for the Capacity Vehicle Routing Problem. *Intl J. Information Technology & Intelligent Computing 2*, 3 (2007).

[22] Ruthmair, M., and Raidl, G. R. A Kruskal-Based Heuristic for the Rooted Delay-Constrained Minimum Spanning Tree Problem. In *Twelfth International Conference on Computer Aided Systems Theory (EUROCAST 2009)* (Gran Canaria, Spain, 2009), R. M.-D. et al., Ed., vol. 5717 of *LNCS*, Springer, pp. 713–720.

[23] Salama, H. F., Reeves, D. S., and Viniotis, Y. An Efficient Delay-Constrained Minimum Spanning Tree Heuristic. In *Proceedings of the 5th International Conference on Computer Communications and Networks* (1996).

[24] Salama, H. F., Reeves, D. S., and Viniotis, Y. The Delay-Constrained Minimum Spanning Tree Problem. In *Proceedings of the 2nd IEEE Symposium on Computers and Communications* (1997), C. Blum, A. Roli, and M. Sampels, Eds., pp. 699–703.

[25] Skorin-Kapov, N., and Kos, M. A GRASP heuristic for the delay-constrained multicast routing problem. *Telecommunication Systems 32*, 1 (2006), 55–69.

[26] Walshaw, C. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research 131*, 1 (2004), 325–372.

[27] Walshaw, C., and Cross, M. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput. 22*, 1 (2000), 63–80.

[28] Xu, Y., and Qu, R. A GRASP approach for the Delay-constrained Multicast routing problem. In *Proceedings of the 4th Multidisplinary International Scheduling Conference (MISTA4)* (Dublin, Ireland, 2009), pp. 93–104.