FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Ein Lösungsarchiv mit Branch-and-Bound-Erweiterung für das Generalized Minimum Spanning Tree Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Christian Gruber
Matrikelnummer 0625102

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.-Prof Dr.Günther Raidl
Mitwirkung: Univ.-Ass. Dr. Bin Hu

Wien, September 5, 2011 _____ _____
(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung

Christian Gruber
Wachbergsraße 29
3382 Schollach

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.

Kurzfassung

In dieser Arbeit wird ein Algorithmus für das Generalized Minimum Spanning Tree-Problem (GMST) entwickelt. Beim GMST-Problem ist ein vollständiger Graph gegeben, bei dem die Knoten in Cluster partitioniert sind. Als Lösung wird ein Spannbaum gesucht, der von jedem Cluster genau einen Knoten beinhaltet und dessen Kosten minimal sind. Dieses Problem ist NP-schwierig. In dieser Arbeit wird eine Heuristik für dieses Problem entwickelt.

Bei diesem Verfahren wird ein Evolutionärer Algorithmus (EA) mit zwei verschiedenen Lösungsarchiven verwendet. Die Lösungsarchive werden dazu benutzt Lösungen zu speichern, um Duplikate zu erkennen und diese in neue Lösungen umzuwandeln. Das eine Lösungsarchiv beruht auf einer Kodierung, bei der die ausgewählten Knoten der Cluster einer Lösung gespeichert werden, während das andere Archiv auf einer Kodierung beruht, bei der gespeichert wird, welche Cluster in der Lösung verbunden sind.

Diese Archive werden in dieser Arbeit durch eine Bounding-Strategie basierend auf dem Branch and Bound Verfahren erweitert. Dabei wird versucht im Archiv an günstigen Positionen geeignete Bounds zu berechnen, die Auskunft darüber geben, wie gut die Lösungen in diesem Bereich des Archivs höchstens sein können. Wird eine Bound gefunden, die schlechter als die beste gefunden Lösung ist, sind diese Lösungen im weiteren Verlauf des Algorithmus uninteressant und werden nicht mehr berücksichtigt. Das führt dazu, dass mehrere Lösungen von vornherein als schlecht erkannt werden können und somit nur Lösungen verfolgt werden, die auch Verbesserungen bringen können.

Zusätzlich zu der Bounding-Strategie wird auch noch ein Nearest Neighbour Ansatz verwendet, bei dem beim Anhängen eines Clusters an den Spannbaum die n nächsten Nachbarcluster bevorzugt werden.

Am Ende der Arbeit wurden Tests durchgeführt, bei denen die Bounding Strategie in den unterschiedlichen Archiven verwendet wurde. Diese Tests führten zu dem Ergebnis, dass die Bounding Strategie zu einer Verbesserung gegenüber den Archiven ohne Bounding Strategie führt. Der Vergleich zwischen den Archiven hat ergeben, dass die Pop-Variante bessere Ergebnisse liefert als die Gosh-Variante. Die Variante, in der beide Archive gleichzeitig verwendet werden, ist wiederum besser als die anderen beiden Varianten.

Abstract

In this work, an algorithm for the generalized minimum spanning tree problem (GMST) is developed. Given is a complete graph where the nodes are partitioned into clusters. A solution is a spanning tree which contains exactly one node of each cluster and its costs are minimal. This problem is NP-hard. In this work, a heuristic is developed for this problem.

In this method, an evolutionary algorithm (EA) is used with two different solution archives. Using a solution archive, it is possible to store solutions generated by the EA in order to detect duplicates and converts duplicate solutions into new solutions. One solution archive based on an encoding in which the spanned nodes of each cluster in the solution are stored. The other archive is based on an encoding which characterizes the connections between the clusters.

These archives are extended by a bounding strategy based on the branch-and-bound technique. They try to calculate appropriate bounds at a convenient positions which give information about how good the solutions in the respective area of the archive can be in the best case. If a bound was found which is worse than the best known solution, the solutions are unattractive in the course of the algorithm and will not be considered. Therefore inferior solutions can be detected at an early stage and only promising solutions that can bring improvements will be pursued.

In addition to the bounding strategy a nearest neighbor approach is implemented in which a cluster attached to the spanning tree is preferred among the the n nearest neighboring clusters.

Tests were carried out in which the bounding strategy was used in the different variants. These tests led to the conclusion that the bounding strategy leads to an improvement in comparison to the “normal” archives. The comparison between the archives shows that the pop version lead to better results than the gosh version. When both archives are used simultaneously, the results are better than the results of the other two variants.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Generalized Minimum Spanning Tree-Problem

Das Generalized Minimum Spanning Tree-Problem (GMST) ist ein kombinatorisches Optimierungsproblem, das eine Verallgemeinerung des Minimum Spanning Tree Problems (MST) ist. Für das MST-Problem ist ein vollständiger Graph G gegeben, bei dem jeder Kante Kosten zugeordnet sind. Eine Lösung des MST-Problems entspricht einer Teilmenge von Kanten, die einen minimalen Spannbaum bilden. Ein minimaler Spannbaum ist ein kreisfreier Teilgraph von G, der mit allen Knoten des Graphen verbunden ist und dessen Summe der Kantenkosten minimal ist. Beim GMST-Problem werden zusätzlich noch die Knoten des MST-Problems durch Cluster partitioniert.
Die formale Definition des GMST-Problems sieht wie folgt aus [4]:

Gegeben ist ein vollständiger gewichteter Graph $G = (V, E, c)$, wobei V die Knotenmenge, E die Kantenmenge und $c : E \to \mathbb{R}^+$ die Kostenfunktion ist. Die Knotenmenge V ist partitioniert in m paarweise disjunkte Cluster $V_1, V_2, \ldots, V_m$, wobei $\bigcup_{i=1,\ldots,m} V_i = V$, $V_i \cap V_j = \emptyset \; \forall i, j = 1, \ldots, m$, $i \neq j$. d_i ist die Anzahl der Knoten in Cluster V_i, $i = 1, \ldots, m$. Eine Lösung für das GMST-Problem ist ein Graph $S = (P, T)$, wobei $P = \{p_1, p_2, \ldots, p_m\} \subseteq V$ enthält genau einen Knoten von jedem Cluster ($p_i \in V_i \; for \; all \; i = 1, \ldots, m$). $T \subseteq E$ ist ein Spannbaum auf die Knoten in P. Die Kosten von T ergeben sich aus den Kantenkosten, $C(T) = \sum_{(u,v)\in T} c(u, v)$. Die optimale Lösung ist dann ein Graph $S = (P, T)$ dessen Kosten $C(T)$ minimal sind. Ein Beispiel für eine solche Lösung ist in Abb. 1 zu finden.

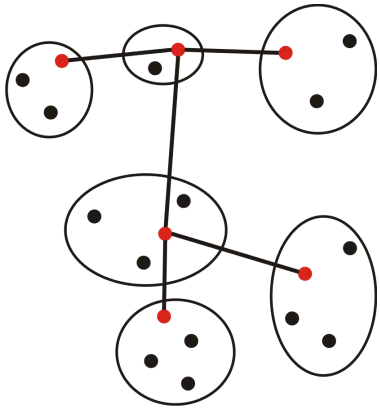

Abb. 1: Lösung für ein GMST-Problem

1.2 Evolutionäre Algorithmen

Ein Evolutionärer Algorithmus (EA) [9] ist eine Metaheuristik, mit deren Hilfe Optimierungsprobleme gelöst werden können. Der EA macht sich die Eigenschaften der natürlichen Evolution zunutze, um ausgehend von einer Anfangspopulation im Laufe der Zeit immer bessere Lösungen zu finden. Dazu werden drei Operationen angewendet: Selektion, Mutation und Rekombination.

Der EA arbeitet mit einer Population von Lösungen, wobei die einzelnen Lösungen in kodierter Form gespeichert werden. Diese kodierten Lösungen werden Genotyp genannt. Sie bestehen oft aus einem Bitstring. In dieser Arbeit werden die Lösungen aber als Integer-Vektor gespeichert. Auf die Kodierungsarten wird in Kapitel 1.4 noch genauer eingegangen. Um die Lösungen auswerten zu können ist es notwendig, die Lösungen auch wieder zu dekodieren. Die dekodierten Lösungen werden Phänotypen genannt.

Der Ablauf des EA ist wie folgt: Zunächst wird für jede Lösung der Population aus den dazugehörigen Phänotypen ein Fitnesswert berechnet. Dieser Fitnesswert ist ein Maß dafür wie gut die Lösung ist. Danach werden durch eine Selektionsfunktion mit Hilfe der Fitnesswerte, zwei Elternlösungen ausgewählt. Aus diesen beiden Elternlösungen wird dann durch eine Rekombination eine neue Lösung generiert. Die Idee dabei ist, dass die guten Teile der ausgewählten Lösungen zusammengesetzt werden und dadurch eine neue bessere Lösung entsteht. Danach wird eine Mutation angewendet, in der die Lösung zufällig an einer bestimmten Stelle verändert wird, um nicht so schnell in einem lokalen Optimum festzustecken. Die Mutation wird aber nur mit einer bestimmten Wahrscheinlichkeit ausgeführt, da sonst der Zufall einen zu großen Einfluss auf die Lösungsfindung hat.

Wie bei den meisten Metaheuristiken, ist die Definition der einzelnen Methoden allgemein gehalten. Um den EA auf ein spezifisches Problem anzuwenden, müssen diese Methoden angepasst werden.
Es gibt zwei unterschiedliche Arten von EAs:

- Steady-State-EA: Dabei wird in jedem Generationsschritt nur eine Lösung aus der Population ersetzt. Die restlichen Lösungen bleiben erhalten.
- generationalen-EA: Hier wird in jedem Generationsschritt die gesamte Population ersetzt.

In dieser Arbeit, genauso wie in den Arbeiten von Sonnleitner [13] und Wolf [15], auf denen diese Arbeit aufbaut, wurde ein Steady-State-EA verwendet.

1.3 Lösungsarchive

Bei einem EA werden neue Lösungen durch Kombination von alten Lösungen, die sich in der Population befinden, erzeugt. Das kann dazu führen, dass eine neu generierte Lösung sich schon in der Population befindet bzw. früher schon einmal untersucht wurde. Daraus können sich zwei Probleme ergeben. Zum einen führt es dazu, dass die selben Lösungen mehrmals evaluiert werden und somit unnötig Laufzeit verbraucht wird, da die erneute Evaluierung keinen Sinn macht. Zum anderen kann das mehrfache betrachten der selben Lösungen zu einem Diversitätsverlust führen, d.h. dass sich die Lösungen in der Population nach kurzer Zeit kaum mehr unterscheiden. Das führt dazu, dass der Lösungsraum nicht mehr so breit durchsucht wird und man so auch schneller in einem lokalen Optimum hängen bleibt.

Um diesen Problemen entgegen zu wirken, sollte bei jeder generierten Lösung überprüft werden, ob diese in früheren Generationen schon einmal erzeugt worden ist. Dazu reicht es aber nicht die Lösungen mit der aktuellen Population zu vergleichen, da hier nur ein kleiner Ausschnitt der bisher untersuchten Lösungen enthalten ist. Daher wird eine Speicherstruktur verwendet, die Lösungsarchiv [12] genannt wird, in der jede generierte Lösung gespeichert wird und in angemessener Zeit danach gesucht werden kann.

Das Lösungsarchiv muss drei Eigenschaften erfüllen. Es muss in angemessener Zeit überprüft werden können ob eine Lösung im Archiv vorhanden ist. Außerdem soll aus einer Lösung, die im Archiv enthalten ist, schnell eine neue Lösung generiert werden können, die der alten möglichst ähnlich ist. Zusätzlich sollen die beiden Aufgaben mit einem angemessenen Speicherverbrauch realisiert werden.

Wie ein Lösungsarchiv in einem EA verwendet wird, ist in Algorithmus 1 dargestellt. Dieser Algorithmus wurde aus [13] übernommen.
Die Frage, welche Datenstruktur sich am besten für ein Lösungsarchiv eignet, wurde in den Arbeiten von Zaubzer [16] und Šramko [14] untersucht. Sie haben drei Datenstrukturen miteinander verglichen: Hashtabellen, Binärbäume und Tries. Das Einfügen einer Lösung und auch das Prüfen, ob eine Lösung schon enthalten ist, kann mit einer Hashtabelle in $O(l)$ (l entspricht der Länge des Strings) gemacht werden. Das Finden einer neuen Lösung kann hingegen im schlimmsten Fall $O(2^l)$ Schritte benötigen. Beim Binärbaum können alle drei Operationen in $O(l * log_2(n))$ durchgeführt werden. Der Nachteil dieser Datenstruktur ist, dass in jedem Knoten eine vollständige Lösung gespeichert wird, was zu einem hohen Speicherverbrauch führt. Bei einem Trie können alle Operationen in $O(l)$ ausgeführt werden. Das führt zu dem Ergebnis, dass ein Trie die geeignetste Datenstruktur, für die vorhin erwähnten benötigten Eigenschaften, ist.

Algorithmus 1 EA mit Lösungsarchiv

1: generate random population pop and insert into archiv
2: **while** termination condition not satisfied **do**
3: $parent1 \leftarrow$ selection(pop)
4: $parent2 \leftarrow$ selection(pop)
5: $sol_{new} \leftarrow$ recombination($parent1$, $parent2$)
6: mutation(sol_{new})
7: lokaleImprovement(sol_{new})
8:
9: **if** sol_{new} included in archiv **then**
10: convert sol_{new} to new solution
11: **end if**
12: insert sol_{new} into archiv
13: replace one solution in pop with sol_{new}
14: **end while**

1.4 Lösungsrepräsentationen

In dieser Arbeit wurden zwei verschiedene Lösungsrepräsentationen verwendet: Die Kodierung von Gosh [2] und die von Pop [11]. Beide Lösungsrepräsentationen wurden auch in den Arbeiten von Wolf [15] und Sonnleitner [13], auf denen diese Arbeit aufbaut, verwendet.

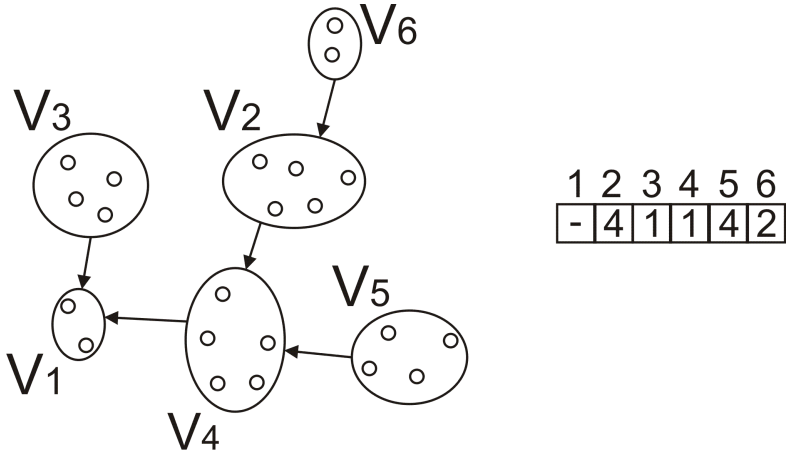

Abb. 2: Pop-Kodierung mit V_1 als Wurzelknoten

1.4.1 Gosh-Kodierung

Bei der Gosh-Kodierung wird für jeden Cluster angegeben, welcher Knoten in diesem Cluster ausgewählt wurde. Die kodierte Lösung besteht somit aus einem Vektor $P = \{p_1, p_2, \ldots, p_m\}$ wobei $p_i \in V$ für alle $i = 1 \ldots m$. Um aus der kodierten Lösung den Phänotypen und den dazugehörigen Lösungswert zu bekommen, kann dazu ein MST-Algorithmus angewendet werden (z.B. Kruskal).

1.4.2 Pop-Kodierung

Bei der Pop-Kodierung wird gespeichert, welche Cluster miteinander verbunden sind. Dazu wird ein globaler Graph definiert [7]: $G^G = (V^G, E^G)$, wobei V^G den Clustern des ursprünglichen Graphen G entsprechen und E^G den Kanten zwischen den Clustern, d.h. $E^G = V^G \times V^G$. Auf diesen Graphen G^G kann nun ein Spannbaum $S^G = (V^G, T^G)$ gebildet werden, wobei $T^G \subseteq E^G$.

Die Pop-Kodierung wird nun mithilfe der Predecessor-Darstellung gemacht. Dazu wird ein Cluster als Wurzelcluster ausgewählt. In dem Spannbaum S^G gibt es nun für jeden Cluster einen eindeutigen Pfad zum Wurzelcluster und somit auch einen eindeutigen Vorgänger. Für die Lösungsrepräsentation wird in einem Vektor oder Array für jeden Cluster (außer dem Wurzelcluster) der Vorgänger gespeichert. In Abb. 2 ist ein Beispiel dazu angegeben.

Ein Problem dieser Kodierung ist, dass nicht jede Kodierung eine gültige GMST-Lösung darstellt. Dieser Faktor muss bei den EA-Operationen berücksichtigt werden, damit dadurch keine ungültigen Lösungen erstellt werden. Sonnleitner [13] hat dazu in seiner

Arbeit einen Repair-Mechanismus entwickelt, mit dem ungültige Lösungen in gültige umgewandelt werden können. Mittels dynamischer Programmierung kann dann aus der kodierten Lösung die optimale Auswahl der Knoten innerhalb der Cluster erfolgen und somit der Phänotyp und den dazugehörigen Lösungswert ermittelt werden.

1.5 Branch and Bound

Branch and Bound ist ein Verfahren, mit deren Hilfe man beweisbar optimale Lösungen für kombinatorische Optimierungsprobleme finden kann. Das Verfahren beruht auf der Idee der beschränkten Enumeration und basiert auf dem Divide & Conquer Prinzip. Branch and Bound-Algorithmen bestehen prinzipiell aus zwei Bereichen: Abgrenzung (Bound) und Verzweigung (Branch).

- Verzweigung (Branch): Durch die Verzweigungsschritte wird das Problem in mehrere Teilprobleme zerlegt, die dann leichter zu lösen sind. Durch wiederholte Anwendung des Verzweigungsschritts auf die Teilprobleme entsteht ein sogenannter Entscheidungsbaum, der den Lösungsraum widerspiegelt.
- Mit der Abgrenzung (Bound) wird versucht, Bereiche im Entscheidungsbaum schon früh als schlecht zu erkennen und diese Bereiche somit nicht weiter zu verfolgen, d.h. es werden schlechte Lösungen schon vorzeitig erkannt. Dazu werden zwei Schranken definiert. Eine untere und eine obere Schranke. Bei einem Minimierungsproblem entspricht die obere Schranke einer zulässigen Lösung während die untere Schranke dem Weg von der Wurzel des Entscheidungsbaums bis zum aktuellen Teilproblem entspricht. Ist nun die untere Schranke schlechter (größer) als die obere Schranke, wird dieser Teil des Entscheidungsbaums nicht weiter verfolgt. Falls sie kleiner als die obere Schranke ist, erfolgt ein weiterer Verzweigungsschritt. Ist die untere Schranke eine zulässige Lösung und ist sie besser als die aktuelle obere Schranke, so wird die untere Schranke die neue obere Schranke.

Wie das Branch and Bound Verfahren in dieser Arbeit verwendet wurde, wird in Kapitel 2 näher erläutert.

1.6 Bisherige Ansätze

Myung, Lee und Tcha [8] haben in ihrer Arbeit das erste Mal das GMST-Problem formal definiert und haben auch gezeigt dass das Problem NP-schwierig ist.

Es gibt einige exakte Algorithmen, die aber nur Lösungen für relativ kleine Instanzen liefern. In [8] wurde ein exakter Algorithmus verwendet mit mehreren Integer Linear Programming-Formulierungen und Branch and Bound Verfahren. Pop hat in [11] eine etwas effizientere Mixed Integer Linear Programming-Formulierung verwendet. Damit konnten Instanzen mit bis zu 240 Knoten mit 30 Clustern oder 160 Knoten mit 40 Clustern optimal gelöst werden. In [1] wurde ein Branch and Cut Algorithmus verwendet.

Um größere Instanzen zu lösen, müssen Metaheuristiken verwendet werden. Eine Metaheuristik ist ein Algorithmus bei dem nicht garantiert werden kann, dass eine optimale Lösung gefunden wird. Man sucht vielmehr eine Lösung, die sich der optimalen Lösung so gut wie möglich annähert. Eine Methaheuristik wird so definiert, dass sie von den Optimierungsproblemen unabhängig sind. Eine auf ein bestimmtes Optimierungsproblem zugeschnittenes Verfahren wird dann Heuristik genannt.

Für das GMST-Problem wurden auch einige Metaheuristiken angewandt. Gosh [2] hat in seiner Arbeit einige Metaheuristik-Ansätze umgesetzt, die auf Tabusuche, Variable Neighborhood Descent und Variable Neighborhood Search basieren. Hu, Leitner und Raidl [3] haben in ihrer Arbeit einen Variable Neighborhood Search Ansatz entwickelt, der auf den Ansätzen von Gosh [2] und Pop [11] basieren. In [4] haben sie diesen Ansatz durch eine zusätzliche Nachbarschaft, die Teillösungen mittels Mixed Integer Programming optimiert, erweitert und damit recht gute Ergebnisse erzielt.

Wolf [15] und Sonnleitner [13] haben in ihren Arbeiten einen EA mit Lösungsarchiv verwendet, wobei sie unterschiedliche Lösungsarchive angewendet haben. Wolf hat das Archiv auf Basis der Gosh-Kodierung erzeugt, während Sonnleitner ein Archiv auf Basis der Pop-Kodierung verwendet hat. Außerdem hat Sonnleitner in seiner Arbeit noch eine Variante entwickelt, die beide Archive gleichzeitig verwendet.
Wolf und Sonnleitner haben mit ihren Ansätzen beide Verbesserungen im Vergleich zu einem normalen EA erreicht. Wobei Sonnleitner mit der Variante, in der er beide Archive gleichzeitig verwendet, die besten Ergebnisse erzielt hat. Die Ergebnisse dieser beiden Arbeiten wurden dann in der Arbeit [5] zusammengefasst und veröffentlicht.

Da in dieser Arbeit die beiden Archive von Wolf und Sonnleitner als Ausgangspunkt genommen wurden und durch ein Bounding Strategie basierend auf einem Branch and Bound Verfahren erweitert wurden, wird in den nächsten beiden Kapiteln auf die beiden Ansätze nochmal genauer eingegangen.

1.6.1 EA mit Gosh-Lösungsarchiv

Wie vorher schon erwähnt, hat Wolf [15] in seiner Arbeit das Lösungsarchiv auf Basis der Gosh-Kodierung aufgebaut. Bei der Gosh-Kodierung wird, wie in Kapitel 1.4.1 bereits beschrieben, eine Lösung durch einen Vektor repräsentiert, bei dem jedes Element einem Cluster entspricht, in dem gespeichert ist, welcher Knoten im Cluster ausgewählt wurde.

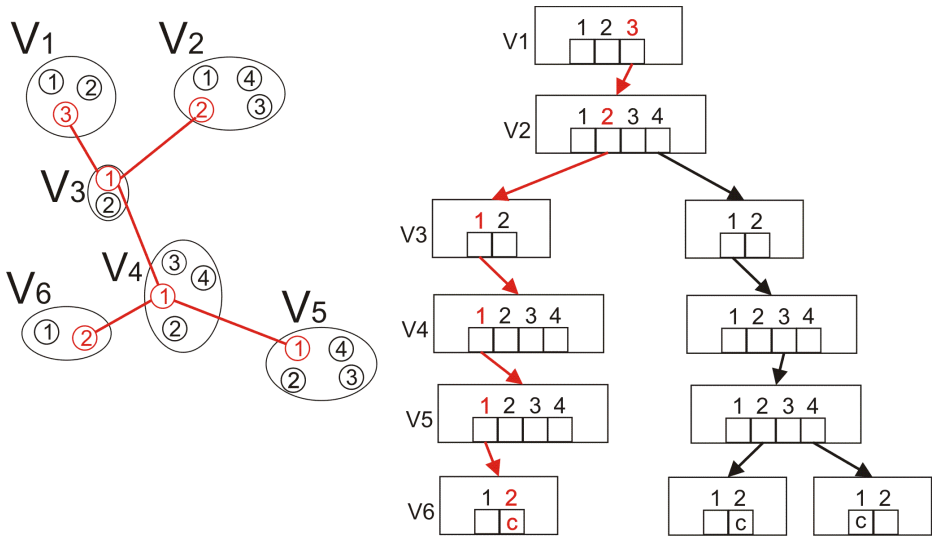

Abb. 3: Lösung <321112> im Gosh-Lösungsarchiv eingefügt

Das Gosh-Lösungsarchiv ist ein Trie, bei dem jede Ebene einem Cluster V_i entspricht. Jeder Trie-Knoten enthält d_i Pointer, d.h. für jeden Knoten im Cluster gibt es einen Pointer. Die Pointer können folgende Zustände annehmen:

- null: Dieser Pointerwert gibt an, dass hier noch keine Lösung eingefügt wurde.
- complete (c): Dieser Wert tritt auf, wenn das Ende einer Lösung erreicht wurde oder alle Lösungen in den darunter liegenden Subtrie bereits $complete$ sind.
- normaler Pointer: Pointer zum nächsten Trie-Knoten.

Abb. 3 zeigt ein Beispiel wie ein Lösungsarchiv nach dem Einfügen einer Lösung aussieht. In diesem Beispiel sind zwei Lösungen in dem Archiv bereits enthalten. Die Lösung <321112> wurde dem Archiv neu hinzugefügt. Wie in Kapitel 1.3 beschrieben werden für ein Lösungsarchiv zwei Funktionen benötigt. Eine Konvertierungs- und eine Einfüge-Methode.

Einfügen einer Lösung

Beim Einfügen einer Lösung wird, beginnend beim Wurzelknoten, immer dem Pointer des Trie-Knoten gefolgt, der dem Knoten aus dem jeweiligen Cluster in der Lösung entspricht. Ist der Pointer *null*, wird ein neuer Trie-Knoten erzeugt und der Pointer wird auf diesen neuen Knoten gesetzt. Wenn er *complete* ist, wurde die Lösung früher schon einmal eingefügt und es kann abgebrochen werden. In diesem Fall wird im nächsten Schritt die Konvertierungs-Methode aufgerufen. Bei einer erfolgreichen Einfüge-Operation, wird im untersten Trie-Knoten der jeweilige Pointer $complete$ gesetzt. Danach wird untersucht, ob es entlang der Lösung Teilbäume im Archiv gibt, die vollständig untersucht wurden. Dazu wird beginnend in der untersten Ebene geprüft, ob alle Pointer im Trie-Knoten $complete$ sind. Wenn das der Fall ist, kann der Trie-Knoten gelöscht werden und der dazugehörige Pointer im Eltern-Knoten auf $complete$ gesetzt werden. Diese Überprüfung erfolgt, beginnend bei dem untersten Trie-Knoten bis zur Wurzel, für alle Trie-Konten entlang der Lösung.

Konvertieren einer Lösung

Um aus einer Lösung im Trie eine neue Lösung zu generieren, werden zunächst jene Trie-Knoten ermittelt, die für eine Änderung in Frage kommen. Das sind jene Trie-Knoten entlang der Lösung im Archiv, die nicht in einem als $complete$ markierten Bereich liegen. Danach wird aus diesen Trie-Knoten einer zufällig ausgewählt, indem dann die Änderungen gemacht werden sollen. In diesem Trie-Knoten wird nach einem *null*-Pointer gesucht. Wenn ein solcher *null*-Pointer vorhanden ist, wird die Lösung dementsprechend abgeändert. Gibt es keinen, geht man im Trie entlang der Lösung zum nächsten Trie-Knoten und wiederholt die Suche. Ist der Pointer der Lösung aber *complete* wird ein anderer Pointer ausgewählt und zu diesem Trie-Knoten weiter gegangen.

Die Frage welche Ebene im Archiv welchem Cluster zugeordnet wird, wurde in [15] näher betrachtet. Wenn die Trie-Knoten der i-ten Ebene im Archiv dem Cluster V_i zugeordnet werden, kommt es zu einem Problem. Es entsteht ein so genanntes Bias im Trie, d.h. die Wahrscheinlichkeit einer Lösungsänderung in einem Cluster ist nicht für alle Cluster gleich groß. Die Knoten in den unteren Ebenen des Archivs haben eine größere Chance verändert zu werden. Um dieses Problem zu umgehen, hat Wolf [15] in seiner Arbeit eine zufällige Zuteilung der Cluster zu den Trie-Knoten verwendet, d.h. beim Erstellen eines neuen Trie-Knoten, wird diesem zufällig ein Cluster zugeordnet. Das hat zur Folge, dass nicht mehr einige Cluster bei der Veränderung der Lösung bevorzugt werden.

1.6.2 EA mit Pop-Lösungsarchiv

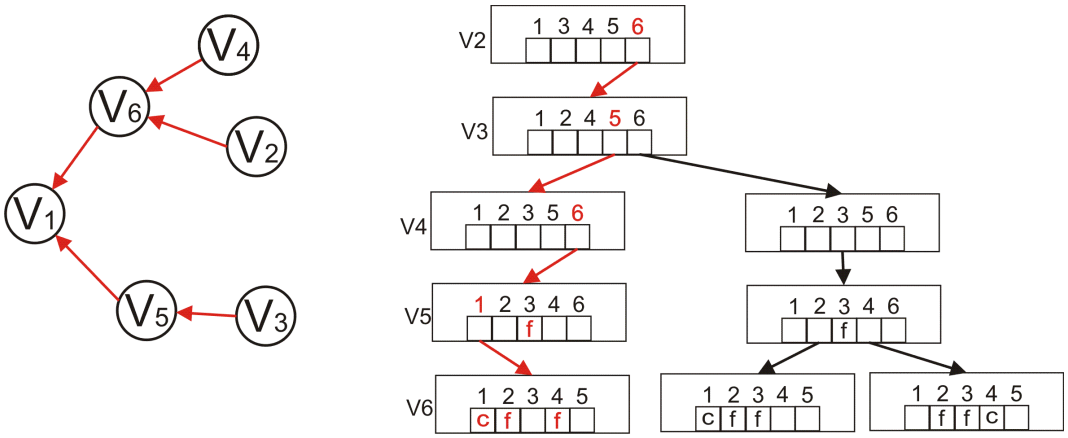

Abb. 4: Lösung <-65611> im Pop-Lösungsarchiv eingefügt, V_1 ist Wurzelcluster

Wie vorher schon erwähnt, hat Sonnleitner [13] in seiner Arbeit das Lösungsarchiv auf Basis der Pop-Kodierung aufgebaut. Bei der Pop-Kodierung werden, wie in Kapitel 1.4.2 bereits beschrieben, die globalen Kanten zwischen den Clustern berücksichtigt. Dabei werden die Vorgänger der Cluster in dem Spannbaum des globalen Graphen, in einem Vektor gespeichert.

Das Pop-Lösungsarchiv ist ein Trie, der aus $m - 1$ Ebenen besteht, wobei jede Trie-Ebene für ein Cluster steht. Außer für den Wurzelknoten, da dieser keinen Vorgänger hat. In jedem Trie Knoten gibt es $m - 1$ Pointer, da ein Cluster in einem Baum nicht sein eigener Vorgänger sein kann. In Abb. 4 ist ein Beispiel zu sehen, wie eine Lösung in einem Pop-Lösungsarchiv gespeichert wird. In diesem Beispiel sind zwei Lösungen in dem Archiv schon enthalten. Die Lösung <-65611> wurde dem Archiv neu hinzugefügt. Da in dieser Kodierung auch Lösungen dargestellt werden können, die keinen Baum darstellen, gibt es in dem Pop-Lösungsarchiv, anders als im Gosh-Archiv, noch einen zusätzlichen Pointerzustand, der eine ungültige Lösung markiert.
Die Pointer können daher folgende Zustände annehmen:

- null: Dieser Pointerwert gibt an, dass hier noch keine Lösung eingefügt wurde.
- complete (c): Dieser Wert tritt auf, wenn das Ende einer Lösung erreicht wurde oder alle Lösungen in dem darunter liegenden Subtrie bereits complete sind.
- forbidden (f): Dieser Wert gibt an, dass diese Kodierung zu einer ungültigen Lösung führt, d.h. zu keinem Baum.
- normaler Pointer: Pointer zum nächsten Trie-Knoten.

Einfüge- und Konvertierungs-Funktionen

Die Einfüge-Funktion läuft im Prinzip genauso ab wie im Gosh-Lösungsarchiv. Der einzige Unterschied ist, dass beim Einfügen von neuen Trie-Knoten in den Trie die verbotenen Felder im Trie berechnet werden. Dazu werden in dem aktuellen Trie-Knoten alle Cluster mit einem kleineren Index als dem aktuellen Cluster geprüft. Bei diesen Clustern wird nun überprüft, ob diese Cluster den aktuellen Cluster als Vorgänger haben. Ist das der Fall so kann dieser als *forbidden* markiert werden. Danach wird rekursiv überprüft, ob es bereits hinzugefügte Cluster gibt, dessen Vorgänger der nun verbotene Cluster ist. Diese werden ebenfalls als *forbidden* markiert.

Die Konvertierungs-Funktion arbeitet im Prinzip genauso wie im Gosh-Lösungsarchiv. Der Unterschied ist, dass hier bei jeder neu generierten Lösung nochmal geprüft werden muss, ob es einen Zyklus gibt und sie somit keine gültige Lösung ist. Das muss gemacht werden, da in der Einfüge-Funktion nicht alle möglichen Zyklen markiert werden können.

KAPITEL 2

Algorithmus

Zur Implementierung des Algorithmus wird ein Steady State EA gemeinsam mit einem Gosh- und einem Pop-Lösungsarchiv verwendet. Die beiden Archive wurden in den Arbeiten von Wolf [15] und Sonnleitner [13] bereits implementiert. Diese Archive werden als Ausgangsposition genommen und durch eine Bounding Strategie basierend auf einem Branch and Bound Verfahren erweitert.

Wie in Kapitel 1.5 bereits beschrieben, besteht das Branch and Bound Verfahren aus zwei Schritten, dem Verzweigen (Branch) und dem Abgrenzen (Bound). Der Branch-Schritt ergibt sich aus dem Lösungsarchiv. Das Archiv kann im Prinzip als Entscheidungsbaum angesehen werden. In jedem Trie Knoten wird für einen bestimmten Cluster eine Entscheidung getroffen und somit entstehen dadurch die verschiedenen Teilprobleme. Im Gosh-Archiv bestehen die Entscheidungen in den Trie-Knoten aus der Auswahl eines Knoten innerhalb eines Clusters, während im Pop-Archiv entschieden wird, welchen Vorgänger der Cluster in dem Spannbaum hat. Für den Bound-Schritt muss eine untere und eine obere Schranke definiert werden. Als obere Schranke wird immer die bisher beste Lösung, die vom Evolutionären Algorithmus gefunden wurde, genommen. Die untere Schranke wird in den einzelnen Trie-Knoten berechnet. Wie die Berechnung genau erfolgt, wird in den Kapiteln 2.1 und 2.2 erläutert. Falls hier eine untere Schranke gefunden wird, die größer als die beste bisher gefundene Lösung ist, kann dieser Teil des Lösungsarchivs als *complete* markiert werden. Im weiteren Verlauf des Algorithmus werden diese Lösungen schon als besucht erkannt und somit im EA nicht mehr berücksichtigt.

Es gibt prinzipiell zwei Möglichkeiten wann eine Bound berechnet wird. Zum einen beim Einfügen der Lösung im Lösungsarchiv und zum anderen beim Konvertieren einer schon eingefügten Lösung zu einer neuen. In dieser Arbeit werden beide Varianten

untersucht und auch miteinander kombiniert.

2.1 Boundberechnung im Gosh-Archiv

Die Boundberechnung erfolgt immer für einen bestimmten Pointer in einem Trie-Knoten. Bei der Gosh-Kodierung wird, wie in Kapitel 1.4.1 bereits erläutert, für jeden Cluster gespeichert, welcher Knoten in dem Cluster ausgewählt wurde. Soll in einem bestimmten Trie-Knoten die Bound berechnet werden, bedeutet das, dass für alle Cluster vom aktuellen Trie-Knoten bis hin zur Wurzel schon eine Auswahl der Knoten in den Clustern getroffen wurde. Für alle Cluster in den darunterliegenden Trie-Ebenen wurde noch keine Auswahl getroffen.

Es ergeben sich also zwei Arten von Clustern. V^1 ist die Menge der Cluster für die bereits eine Knotenauswahl getroffen wurde und V^0 ist die Menge der Cluster für die noch keine Auswahl getroffen wurde.
Um in einem bestimmten Trie-Knoten eine Bound zu berechnen, wird zunächst der Graph $G' = (V', E')$ definiert, wobei $V' = V^1 \cup V^0$. Die Menge E' setzt sich aus 3 verschiedenen Arten von Kanten zusammen:

- $E^{11} = V^1 \times V^1$ sind die Kanten zwischen allen Paaren von Clustern, für die bereits eine Auswahl getroffen wurde. Die Kantenkosten ergeben sich aus der Distanz zwischen den ausgewählten Knoten der Cluster.
- $E^{10} = V^1 \times V^0$ sind die Kanten zwischen allen Paaren von Clustern, bei denen schon eine Auswahl getroffen wurde und denen für die noch keine gemacht wurde. Die Kantenkosten zwischen den Clustern $V_i \in V^1$ und $V_j \in V^0$ lauten $c(p_i, V_j) = min\{c(p_i, p_j)|p_j \in V_j\}$, wobei $p_i \in V_i$ ist.
- $E^{00} = V^0 \times V^0$ sind die Kanten zwischen allen Paaren von Clustern, für die noch keine Auswahl getroffen wurde. Die Kantenkosten zwischen den Clustern $V_i \in V^0$ und $V_j \in V^0$ lauten $c(V_i, V_j) = min\{c(p_i, p_j)|p_i \in V_i, p_j \in V_j\}$.

Die Definition der Kantenmenge lautet also $E' = E^{11} \cup E^{10} \cup E^{00}$. Die Berechnung der Kantenkosten wird in einem Preprocessing-Schritt durchgeführt, d.h. sie erfolgt einmal zu Beginn des Algorithmus und ist somit für die Laufzeit des Algorithmus nicht mehr relevant. Um die Bound zu berechnen, wird für den Graphen G' ein minimaler Spannbaum $S' = (V', T')$ ermittelt, wobei $T' \subseteq E'$. Die Bound entspricht dabei den Kosten des Spannbaums T', $C(T') = \sum_{(u,v)\in T'} c(u, v)$. Die Berechnung des Spannbaums erfolgt mit einem Kruskal Algorithmus mit Union-Find. Die Laufzeit dieses Algorithmus wird durch das Sortieren der Kanten bestimmt. Da der Graph G' ein vollständiger Graph ist, entspricht die Anzahl der Kanten $|E| = m * (m - 1)$. Dadurch ergibt sich eine Laufzeit von $O(m^2 log(m^2))$ für die Berechnung der Bound.

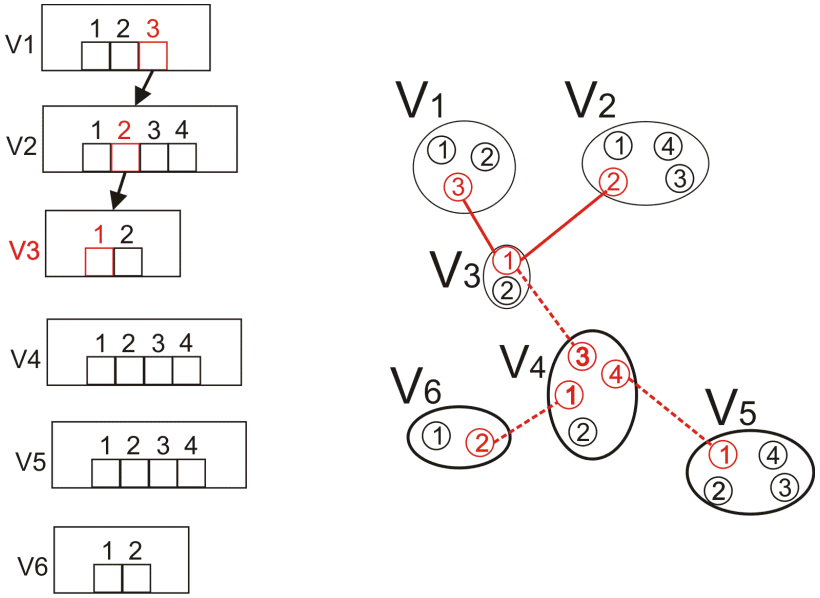

Abb. 5: Boundberechnung im Cluster V_3 beim Einfügen der Lösung <321112>

In Abb. 5 wird illustriert wie eine Boundberechnung im Gosh-Lösungsarchiv funktioniert. In diesem Beispiel wird im Trie-Knoten des Cluster V_3 die Bound berechnet. An dieser Position wurden die Cluster V_1, V_2 und V_3 schon in den oberen Ebenen eingefügt und somit eine Knotenauswahl für die Cluster getroffen. In den darunter liegenden Clustern (V_4, V_5 und V_6) wurde noch keine Auswahl getroffen. Für die Boundberechnung bedeutet das, dass hier zwischen den Clustern V_1, V_2 und V_3 als Kantenkosten der Abstand der ausgewählten Knoten genommen wird. Zwischen dem Cluster V_3 und V_4 wird als Kosten der Distanzen zwischen dem ausgewählten Knoten 1 (V_3) und dem Knoten 3 (V_4) genommen, da dieser die geringste Distanz zum Knoten 1 (V_3) hat. Zwischen den Clustern V_4, V_5 und V_6 werden die minimalen Distanzen zwischen den Clustern verwendet. D.h. zwischen Cluster V_6 und V_4 wird die Kante zwischen Knoten 2 (V_6) und Knoten 1 (V_4) genommen und zwischen V_5 und V_4 wird die Kante zwischen Knoten 1 (V_5) und Knoten 4 (V_4) genommen.

Ist die berechnete Bound schlechter als die beste bisher gefundene Lösung, kann der Pointer 1 des Clusters V_3 als *complete* markiert werden. Das kann gemacht werden, da zwischen den Clustern V_4, V_5 und V_6 immer der minimale Abstand genommen wurde. Wenn nun Knoten in den Clustern ausgewählt werden, bleiben die Kosten des Spannbaums gleich oder werden größer aber sie werden sicher nicht kleiner. Das führt dazu, dass alle Lösungen in diesem Subtrie mindestens so groß sind als die berechnete Bound und somit in diesem Fall auch größer sind als die beste bisher gefundene Lösung.

2.1.1 Einfüge-Methode

Wie vorhin schon erwähnt, ist eine Möglichkeit, wann die Boundberechnung durchgeführt werden kann, die Einfüge-Operation des Lösungsarchivs. Beim Einfügen einer Lösung wird, beginnend beim Wurzelknoten, immer dem Pointer des Trie-Knoten gefolgt, der dem Knoten aus dem jeweiligen Cluster in der Lösung entspricht. Bei einer erfolgreichen Einfüge-Operation werden also m Trie-Knoten durchlaufen. Eine Möglichkeit wäre, in jedem dieser m Trie-Knoten für den jeweiligen Pointer, der dem ausgewählten Knoten dieses Clusters in der Lösung entspricht, die Bound zu berechnen.

Das führt aber zu einem Problem. Die Berechnung der Bound hat eine Laufzeit von $O(m^2\ log(m^2))$. Wenn in jedem Trieknoten die Bound berechnet wird, müsste die Bound m-mal ermittelt werden. Das würde zu einer Laufzeit von $O(m^3\ log(m^2))$ führen, was eine erhebliche Verschlechterung gegenüber der normalen Einfüge-Operation ohne Boundberechnung, mit einer Laufzeit von $O(m)$, wäre.

Eine Möglichkeit dieses Problem abzuschwächen ist es, die Bound nicht in jedem der m Trie-Knoten zu berechnen, sondern nur in einer bestimmten Auswahl von Knoten. Dazu wurde der Parameter $branch\_and\_bound\_prob$ eingeführt, mit dem angegeben werden kann, mit welcher Wahrscheinlichkeit eine Boundberechnung für einen Trie-Knoten durchgeführt wird. Als ein guter Wert für diesen Parameter hat sich $0,05$ herausgestellt. Das bedeutet, es wird für 5% der Trie-Knoten die Bound berechnet. Die Auswahl erfolgt dabei zufällig.

In Algorithmus 2 ist der Pseudocode für die Einfüge-Operation mit Boundberechnung dargestellt. Diese beruht auf der Einfüge-Operation von Wolf [15], nur dass diese durch die Bounding Strategie erweitert wurde. Zunächst wird der Vektor V^0, in der alle Cluster gespeichert sind, die noch nicht in das Archiv eingefügt worden sind, initialisiert. Ausgehend vom Wurzelknoten wird nun immer dem Pointer des Trie-Knoten gefolgt, der dem Knoten aus dem jeweiligen Cluster in der Lösung entspricht. Der Cluster wird dabei immer aus dem Vektor V^0 entfernt und dem Vektor V^1 hinzugefügt, indem alle Cluster gespeichert sind die schon eingefügt worden sind. Diese beiden Vektoren werden dann für die spätere Boundberechnung benötigt. Ist der Pointer gleich $null$, muss ein neuer Trie-Knoten angelegt werden. Dazu wird zufällig ein Cluster aus dem Vektor V^0 ausgewählt. Danach wird, mithilfe des Parameters $branch\_and\_bound\_prob$, ermittelt ob für diesen Trie-Knoten eine Bound berechnet werden soll. Wird eine Bound berechnet und ist sie schlechter als die beste bisher gefundene Lösung, werden alle Trie-Knoten in den darunterliegenden Sub-Trie gelöscht und der Pointer als $complete$ markiert. Anschließend wird die Einfüge-Operation abgebrochen und $false$ als Rück-

Algorithmus 2 GoshTrie insert with Bound

Eingabe: sol - solution to insert; best solution $bestsol$
Ausgabe: insertion successfull (true/false)
Variablen: m=number of clusters; $root$=root from archiv

```
1:  V^0 ← list with all numbers from 0 to m − 1
2:  curr ← root
3:  for i = 0...m − 1 do
4:     pos ← sol[curr.cluster]
5:     move cluster curr from V^0 to V^1
6:     if curr.next[pos] == complete then
7:        return false
8:     end if
9:     if curr.next[pos] == null then
10:       if i == m − 1 then
11:          curr.next[pos] ← complete
12:       end if
13:       rand ← random cluster from V^0
14:       V^0 ← list with all numbers from 0 to m − 1
15:       curr.next[pos] ← new node with cluster rand
16:    end if
17:    random choice if bound has to be calculated
18:    if bound has to be calculated then
19:       if bestsol < calculateBound(V^1, V^0) then
20:          delete curr.next[pos]
21:          curr.next[pos] ← complete
22:          return false
23:       end if
24:    end if
25:    curr ← curr.next[pos]
26: end for
27: check if there are complete subtrees along sol
28: return true
```

gabewert zurück gegeben.
Das hat zur Folge, dass die Lösung so behandelt wird als wäre sie schon einmal eingefügt worden und es wird eine neue Lösung generiert. Wenn das Einfügen erfolgreich war, wird von unten nach oben entlang von sol untersucht, ob bei einem Trie-Knoten alle Pointer $complete$ sind. Ist das der Fall kann dieser gelöscht werden und der Pointer im Eltern Knoten auf $complete$ gesetzt werden.

2.1.2 Konvertierungs-Methode

Die zweite Möglichkeit, wann Boundberechnungen durchgeführt werden können, ist bei der Konvertierungs-Operation des Lösungsarchivs. Bei der Konvertierungs-Operation wird aus einer im Archiv enthaltenen Lösung eine neue Lösung gemacht, die der alten möglichst ähnlich ist. Die Boundberechnung erfolgt nach jeder Änderung, die in der alten Lösung gemacht wird. Ist diese Bound besser als die beste bisher gefundene Lösung, wird diese neue Lösung akzeptiert. Ist die Bound aber schlechter, so werden die Änderungen wieder rückgängig gemacht, die Lösung im Archiv als *complete* markiert und danach weiter nach einer anderen neuen Lösung gesucht.

In Algorithmus 3 ist dargestellt wie die Konvertierung mit Berücksichtigung der Bounds funktioniert. Zuerst wird ein zufälliger Trie-Knoten der Lösung sol ausgewählt. In Zeile 5 wird dann geprüft, ob es einen null-Pointer in dem ausgewählten Trie-Knoten gibt. Wenn es einen gibt, wird die Lösung geändert und überprüft ob die Bound der neuen Lösung an dieser Position schlechter ist als die beste bisher gefunden Lösung. Wenn sie besser ist, wird sie akzeptiert und es kann abgebrochen werden. Ist sie schlechter, wird die Änderung wieder rückgängig gemacht und der Pointer als $complete$ markiert. Die Suche wird danach fortgesetzt. In den Zeilen 14-21 wird versucht, entlang der Lösung im Archiv eine Ebene nach unten zu gehen. Dieser Schritt erfolgt nur, wenn der Pointer nicht $complete$ ist und die Bound besser ist als die beste Lösung. Wenn entlang von sol nicht nach unten gegangen werden kann, wird in den Zeilen 22-31 versucht einen anderen Pointer, der nicht $complete$ ist, zu finden und an diesen eine Ebene nach unten zu gehen. Hier erfolgt wieder eine Überprüfung der Bound. Der ausgewählte Pointer wird nur weiter verfolgt, wenn die Bound besser ist als die beste Lösung.

Es kann vorkommen, dass in einem Trie-Knoten kein Pointer vorhanden ist, der zu einer guten Bound führt. Das hat zur Folge, dass alle Pointer als $complete$ markiert werden und somit nicht mehr weiter nach unten gegangen werden kann. Deshalb muss bei jedem Schleifendurchlauf überprüft werden, ob es noch einen Pointer gibt, der nicht $complete$ ist. Wenn nicht, wird dieser Trie-Knoten gelöscht und der zugehörige Pointer im Eltern-Knoten als $complete$ markiert. Danach kann Abgebrochen werden und $false$ zurück gegeben werden. Die Konvertierungs-Methode wird danach nochmal aufgerufen.
Das kann zu einem Problem führen. Wenn die Ausgangslösung relativ schlecht ist, kann die Suche nach einer neuen Lösung recht lange dauern. Tests haben gezeigt, dass dieser Fall nicht sehr oft auftritt und in den meisten Fällen schon nach wenigen Versuchen eine neue Lösung gefunden werden kann. In wenigen Fällen wurde aber lange nach einer Lösung gesucht. Um dieses Problem zu umgehen, wurde ein Parameter num_conv eingeführt, mit dem die Anzahl der Konvertierungsversuche begrenzt werden kann.

Algorithmus 3 GoshTrie convert with Bound

Eingabe: sol - solution to convert, best solution $bestsol$

Ausgabe: convertion successfull (true/false)

1: $curr \leftarrow$ random Trie-node along sol
2: add all clusters from $root$ to $curr$ to V^1 and the other clusters to V^0
3: **while** $curr! = null \;\&\&\; curr! = complete$ **do**
4: check if all pointers in $curr$ are $complete$, so we can cancel
5: **if** $curr$ has a $null$-pointer **then**
6: $sol[curr.cluster] \leftarrow$ random $null$-pointer of $curr$
7: **if** $bestsol <$ $calculateBound(V^1, V^0)$ **then**
8: $curr.next[sol[curr.cluster]] \leftarrow complete$
9: undo changes of sol
10: **else**
11: **return** $true$
12: **end if**
13: **end if**
14: **if** $curr.next[sol[curr.cluster]]! = complete]$ **then**
15: move $curr.next[sol[curr.cluster]].cluster$ from V^0 to V^1
16: **if** $bestsol <$ $calculateBound(V^1, V^0)$ **then**
17: delete $curr.next[sol[curr.cluster]]$
18: $curr.next[sol[curr.cluster]] \leftarrow complete$
19: **else**
20: $curr \leftarrow curr.next[sol[curr.cluster]]$
21: **end if**
22: **else**
23: $sol[curr.cluster] \leftarrow$ a random not-complete-pointer, of $curr$
24: move $curr.next[sol[curr.cluster]].cluster$ from V^0 to V^1
25: **if** $bestsol <$ $calculateBound(V^1, V^0)$ **then**
26: delete $curr.next[sol[curr.cluster]]$
27: $curr.next[sol[curr.cluster]] \leftarrow complete$
28: undo changes of sol
29: **else**
30: $curr \leftarrow curr.next[sol[cur.cluster]]$
31: **end if**
32: **end if**
33: **end while**
34: **return** $false$

2.2 Boundberechnung im Pop-Archiv

Die Boundberechnung erfolgt, genauso wie beim Gosh-Archiv auch, immer für einen bestimmten Pointer in einem Trie-Knoten. Bei der Pop-Kodierung wird, wie in Kapitel 1.4.2 erläutert, für jeden Cluster der Vorgänger des minimalen Spannbaums, der aus den globalen Graphen G^G erzeugt wird, gespeichert. Wenn in einem bestimmten Trie-Knoten die Bound berechnet werden soll, bedeutet das, dass für alle Cluster vom aktuellen Trie-Knoten bis hin zur Wurzel der Vorgänger bereits ausgewählt wurde. Diese Menge wird als V^1 bezeichnet. Für alle Cluster in den darunterliegenden Trie-Ebenen wurde noch keine Auswahl getroffen. Sie sind noch mit keinem Cluster verbunden. Diese Menge wird als V^0 bezeichnet.

Dadurch ergibt sich ein Wald G''. In Abb. 6 wird illustriert, wie so ein Wald aussehen kann. Die Abbildung zeigt einen Graph G'' bei einer Boundberechnung im Trie-Knoten von Cluster 6.

Die Berechnung der Bound erfolgt nun in mehreren Schritten. In Algorithmus 4 wird in dem Pseudocode gezeigt, wie die Boundberechnung funktioniert. Zunächst wird ein Vektor $data$ erzeugt, indem am Ende der Funktion die ausgewählten Knoten in den Clustern gespeichert werden. Die Elemente von $data$ werden mit -1 initialisiert.
Danach wird mit den Methoden $calcClusters$ und $generateList$ durch dynamische Programmierung für die einzelnen Bäume von G'' ermittelt, welche Knoten innerhalb

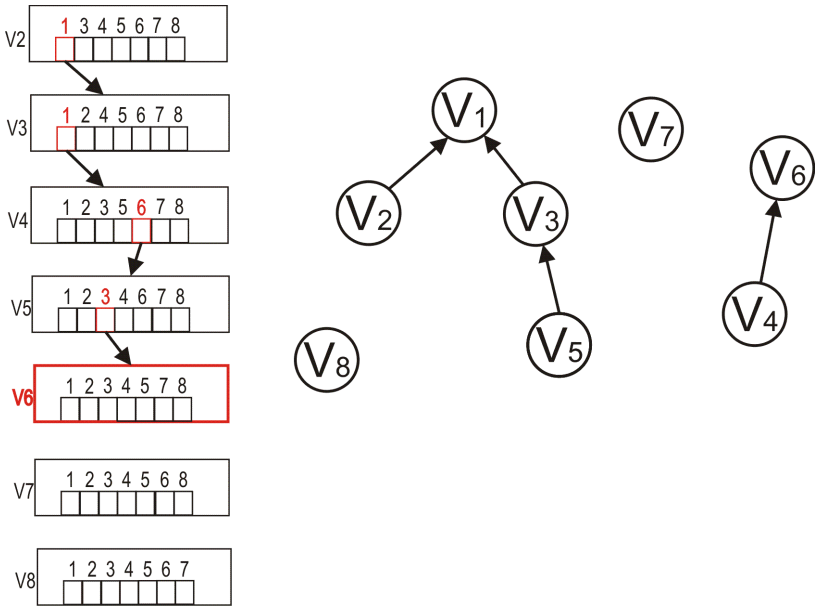

Abb. 6: Boundberechnung im Cluster V_6

Algorithmus 4 calcBoundByDynProg

Eingabe: root-node cNr; solution sol; current level of the trie $aktPos$
Ausgabe: $bound$
Variablen: $data$ = list of the selected nodes of the clusters

1: init $data$ with -1
2: $nextcNr \leftarrow cNr$
3: **repeat**
4: calcClusters ($nextcNr$, sol, $data$, $aktPos$)
5: generateList ($nextcNr$, $data[nextcNr]$, sol, $aktPos$)
6: $nextinnerNode \leftarrow$ next cluster with $data[nextinnerNode] = -1$ and $nextinnerNode <= aktPos$
7: **if** it exists a $nextinnerNode$ **then**
8: $nextcNr \leftarrow$ getRootNode(sol, $nextinnerNode$, $aktPos$)
9: **end if**
10: **until** it exists a $nextinnerNode$
11: $bound \leftarrow$ calculateBound(sol, $data$, $aktPos$)
12: **return** $bound$

der Cluster ausgewählt werden sollen. Die Knoten werden so ausgewählt, dass die Kosten der daraus entstehenden Lösung minimal sind. Begonnen wird dabei mit dem Baum der vom Wurzelknoten des Lösungsarchivs ausgeht. Danach wird geprüft, ob es noch ein Cluster in $C \in V^1$ gibt, für den noch kein Knoten ausgewählt wurde, d.h. dessen Eintrag in $data$ noch -1 ist. Falls es so einen Cluster gibt, wird für diesen Cluster C der Wurzelknoten des Baums, in dem er sich in G'' befindet, ermittelt. Von diesem Wurzelknoten wird nun wieder mit $calcClusters$ und $generateList$ eine Auswahl der Knoten für die Cluster des Baums getroffen. Das wird solange wiederholt bis für alle Cluster in V^1 eine Auswahl getroffen wurde.

Danach wird mit der Methode $calculateBound$ mit einem angepassten Kruskal Algorithmus ein minimum Spanning-Tree ermittelt. Ähnlich wie bei der Boundberechnung im Gosh-Archiv, werden im Pop-Archiv auch drei Arten von Kanten zur Berechnung verwendet:

- $E^{11} = V^1 \times V^1$ ist die Menge aller Kanten von G''. Die Kantenkosten ergeben sich aus dem Abstand zwischen den ausgewählten Knoten der Cluster.
- $E^{10} = V^1 \times V^0$ sind die Kanten zwischen allen Paaren von Clustern, bei denen schon eine Auswahl getroffen wurde und denen für die noch keine gemacht wurde. Die Kantenkosten zwischen den Clustern $V_i \in V^1$ und $V_j \in V^0$ lauten $c(p_i, V_j) = min\{c(p_i, p_j)|p_j \in V_j\}$, wobei $p_i \in V_i$ ist.

- $E^{00} = V^0 \times V^0$ sind die Kanten zwischen allen Paaren von Clustern, für die noch keine Auswahl getroffen wurde. Die Kantenkosten zwischen den Clustern $V_i \in V^0$ und $V_j \in V^0$ lauten $c(V_i, V_j) = min\{c(p_i, p_j)|p_i \in V_i, p_j \in V_j\}$.

Im Kruskal-Algorithmus werden jetzt zunächst alle Kanten von E^{11} eingefügt. Danach werden die restlichen Kanten $E^{10} \cup E^{00}$, wie bei einem normalen Kruskal-Algorithmus, in der Reifenfolge ihrer Kantenkosten hinzugefügt, sodass kein Zyklus entsteht.

Algorithmus 5 calcClusters

Eingabe: root-node cNr; solution sol; $data$; current level of the trie $aktPos$

```
1: for i=0...aktPos do
2:    if sol[i]==cNr then
3:       calcClusters (i, sol, data, aktPos)
4:    end if
5: end for
6: calcClusterweights (cNr, sol, data)
```

Wie vorhin bereits erwähnt, wird mit den Methoden $calcClusters$ und $generateList$ die bestmögliche Auswahl von Knoten innerhalb der Cluster getroffen.
Zunächst wird mit $calcClusters$ (Algorithmus 5) für jeden Knoten der zu untersuchenden Cluster die Kosten berechnet, die entstehen wenn dieser Knoten ausgewählt wird. Dazu wird $calcClusters$ rekursiv aufgerufen, um, angefangen bei den Clustern in der untersten Ebene, die Kosten für die Knoten zu berechnen.

Die Kosten werden in der Methode $calcClusterweights$ (Algorithmus 6) berechnet. Dazu wird für jeden Knoten $p_i \in C_{cur}$ des jeweiligen Clusters $C_{cur} \in V^1$ die minimalen Kosten berechnet, die entstehen würden, wenn dieser ausgewählt werden würde. Das geschieht, indem für alle Nachfolgecluster $C_{suc} \in V^1$ ein Knoten $p_j \in C_{suc}$ ausgewählt wird. Der Knoten p_j wird dabei so gewählt, dass die Summe der Kosten von p_j und die Distanz von p_i zu p_j minimal ist. Die Kosten des Knoten sind also $C(p_i) = \sum_{p_j \in P_{suc}}(C(p_j) + dist(p_i, p_j))$, wobei P_{suc} die Menge der minimalen Knoten der Nachfolgecluster von C_{cur} ist. In der Datenstruktur $nodes$ werden für alle Knoten deren Kosten $C(p_i)$ gespeichert und berechnet und auch alle Knoten der Menge P_{suc} gespeichert. Diese Datenstruktur wird für die spätere Auswahl der Knoten in den Clustern benötigt.

Mit der Methode $generateList$ (Algorithmus 7) werden nun die ausgewählten Knoten innerhalb der Cluster ermittelt. Dazu wird beginnend beim Wurzelcluster der Knoten mit den geringsten Kosten in $data$ gespeichert. Dazu wird die Datenstruktur $nodes$ verwendet, in der für jeden Knoten die günstigsten Nachfolgeknoten gespeichert sind.

Algorithmus 6 calcClusterweights

Eingabe: current cluster cNr; solution sol; $data$
Variablen: $minWeight$=current minimal weight of the cluster

```
1:  childCl ← all cluster which predecessor is cNr in sol
2:  for all nodes parentNode in cluster cNr do
3:     for all child in childCl do
4:        for all nodes childNode in cluster child do
5:           value= nodes[childNode].weight + distance(parentNode, childNode)
6:           if value < minValue then
7:              minValue=value
8:              minChildNodeNr=childNode
9:           end if
10:       end for
11:       nodes[parentNode].addFolower(child, minChildNodeNr, minValue)
12:    end for
13:    if nodes[parentNode].weight < minWeight then
14:       minWeight = nodes[parentNode].weight
15:       data[cNr] = parentNode
16:    end if
17: end for
```

Algorithmus 7 generateList

Eingabe: current cluster cNr; $node$ - selected node of cluster cNr $node$; solution sol; $data$; current level of the trie $aktPos$

```
1: data[cNr] = node
2: for i=0 ... aktPos do
3:    if sol[i] == cNr then
4:       generateList(i, nodes[node].getFollower(i), data, sol, aktPos)
5:    end if
6: end for
```

Die Laufzeit der Boundberechnung wird durch die Berechnung der Kosten für die einzelnen Knoten bestimmt. Im schlimmsten Fall müssen für alle Knoten des Graphen die Kosten berechnet werden. Deshalb kommt es zu einer Laufzeit von $O(n^2)$.

2.2.1 Inkrementelle Boundberechnung

In [4] wurde eine Nachbarschaft verwendet, in der die neuen Lösungen inkrementell berechnet wurden. Derselbe Ansatz soll auch in dieser Arbeit verwendet werden. Beim Einfügen einer Lösung wird an verschiedenen Stellen entlang der Lösung im Trie die Bound berechnet. Dabei werden aber für manche Teile der Lösung Berechnungen mehrfach durchgeführt. Dasselbe gilt auch für das Berechnen von Bounds bei der Konvertierung von Lösungen.

Die Idee der inkrementellen Berechnung ist es, dass bei der ersten Berechnung der Bound innerhalb einer Einfüge- bzw. Konvertierungs-Methode die Bound, wie vorhin beschrieben, "normal" berechnet wird. Bei den nächsten Berechnungen der Bound müssen nicht mehr für alle Teile des Baums die Kosten der Knoten berechnet werden. Die Knotenkosten innerhalb eines Clusters V_i können sich nur ändern, wenn V_i einen neuen Nachfolgecluster im Graphen G'' bekommt oder die Kosten eines Nachfolgers von V_i sich geändert haben.

Zur Umsetzung der inkrementellen Boundberechnung wurde ein Vektor $valid$ verwendet, indem für jeden Cluster angegeben wird, ob sich die Knotenkosten in den Cluster ändern. Die einzige Änderung die, gegenüber der "normalen" Boundberechnung zu machen ist, ist in der Methode $calcCluster$ vorzunehmen. In Algorithmus 8 ist zu sehen, wie die geänderte Methode aussieht. In der Methode wird der Vektor $valid$ dazu verwendet, um festzustellen, ob die Bound berechnet werden soll oder nicht.

Algorithmus 8 calcClustersImprove

Eingabe: root-node cNr; solution sol; $data$; current level of the trie $aktPos$

```
1: if !valid[cNr] then
2:    for i=0...aktPos do
3:       if sol[i]==cNr then
4:          calcClusters (i, sol, data, aktPos)
5:       end if
6:    end for
7:    calcCluster (cNr, sol, data)
8: end if
```

2.2.2 Pop mit Nearest Neighbours Reduktion

In [10] wurde ein Ansatz verfolgt, indem für die Berechnung eines GTSP sogenannte "candidate lists" verwendet wurden, um den Suchraum einzuschränken. Dabei wurden für jeden Knoten die n nächsten Nachbarn in einer Liste gespeichert und nur diese als mögliche Nachfolger in der Tour betrachtet. Der Ansatz wird auch in dieser Arbeit verfolgt. Dazu wird die Rekombinations- und Mutations-Methode des EAs entsprechend angepasst. Außerdem wird in der Konvertierungs-Methode des Pop-Lösungsarchivs die Nearest Neighbour Reduktion auch angewendet. Darauf wird später in Kapitel 2.2.4 genauer eingegangen werden.

Rekombination

Bei der Rekombinations-Methode wurde die in [13] verwendete Methode als Grundlage genommen und durch den Nearest Neighbour Ansatz ergänzt.
Die Rekombination wird dabei wie folgt durchgeführt: Für jedes Cluster C_i wird überprüft, ob es in den beiden Lösungen den gleichen Vorgänger hat. Wenn das der Fall ist, wird dieser Vorgänger in die neue Lösung übernommen. Hat der Cluster C_i in den beiden Lösungen unterschiedliche Vorgänger, gibt es drei Möglichkeiten:

- Ein Vorgänger C_j ist in der Menge der Nearest Neighbours von C_i und der andere C_k nicht: In diesem Fall wird C_j in der neuen Lösung aufgenommen.
- Beide Vorgänger sind in der Menge der Nearest Neighbours von C_i: Hier wird zufällig einer der beiden ausgewählt.

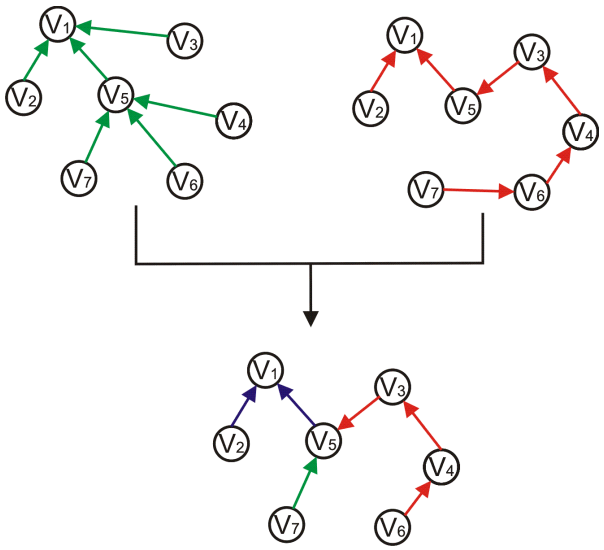

Abb. 7: Rekombination von zwei Lösungen

- Beide Vorgänger sind nicht in der Menge der Nearest Neighbours von C_i: Hier wird ebenfalls einer der beiden zufällig ausgewählt.

Beim Einfügen eines Clusters wird mittels Union Find überprüft, ob durch das Hinzufügen des Clusters ein Zyklus entsteht. Ist das der Fall, so wird der Vorgänger der anderen Lösung übernommen, d.h. der Cluster an dieser Stelle eingefügt. Führt das auch zu einem Zyklus, wird dieser Cluster als nicht eingefügt markiert. Diese markierten Cluster werden zum Schluss in den Baum hinzugefügt, indem zufällig ein Knoten aus der Nearest Neighbours Liste des jeweiligen Clusters ausgewählt wird und er dann an dieser Stelle eingefügt wird. Ist es nicht möglich den Cluster an einem Nearest Neighbour anzuhängen, ohne einen Zyklus zu erzeugen, wird er zufällig an einer Stelle im Baum angehängt. In Abb. 7 ist ein Beispiel für eine solche Rekombination illustriert.

Mutation

Wie vorhin schon erwähnt, wurde in der Mutation-Methode der Nearest Neighbours Ansatz auch umgesetzt. Dazu wurde die in [13] verwendete Methode als Grundlage genommen und durch den Nearest Neighbour Ansatz erweitert.

In der Mutations-Methode wird an einer zufällig ausgewählten Stelle in der Lösung eine Änderung vorgenommen. Der Ablauf der Mutations-Methode ist wie folgt: Zuerst wird zufällig ein Cluster C_i in der Lösung ausgewählt, dessen Vorgänger geändert werden soll. Danach werden alle Nearest Neighbours ermittelt, deren direkten oder indirekten Vorgänger nicht C_i ist. Im nächsten Schritt wird aus dieser Liste zufällig ein neuer Vorgänger ermittelt und in die neue Lösung gespeichert. Gibt es keinen solchen Nearest Neighbour, werden mit Hilfe eines Tiefensuche-Algorithmus alle möglichen Cluster, deren direkten oder indirekten Vorgänger nicht C_i ist, ermittelt. Danach wird aus diesen Clustern einer zufällig ausgewählt und als neuer Vorgänger für C_i genommen. In Algorithmus 9 wird gezeigt wie die Mutations-Methode funktioniert.

Algorithmus 9 mutatePredecessor

Eingabe: Lösung sol

1: $mutate \leftarrow$ random cluster
2: $cand \leftarrow$ nearest neighbours from $mutate$ where $mutate$ is not the direct or indirect predeccessor in sol
3: **if** $cand.size() != 0$ **then**
4: $\quad cand \leftarrow$all cluster where $mutate$ is not the direct or indirect predeccessor in sol
5: **end if**
6: $pred_{new} \leftarrow$ random element from $cand$
7: $sol[mutate] \leftarrow pred_{new}$

2.2.3 Einfüge-Methode

Genau wie beim Gosh-Archiv wird auch beim Pop-Archiv die Bounding Strategie in die Einfüge-Methode integriert. Bei der Pop-Einfüge-Methode wird, wie auch in Kapitel 2.1.1, die Bound nicht in jedem Einfügeschritt berechnet, sondern nur für einen Teil der Trie-Knoten der einzufügenden Lösung. Dazu wird wieder der Parameter $branch\_and\_bound\_prob$ verwendet, der angibt mit welcher Wahrscheinlichkeit die Bound für einen Trie-Knoten berechnet wird.

Algorithmus 10 PopTrie insert with Bound

Eingabe: solution to insert sol; best solution $bestsol$
Ausgabe: insertion successfull (true/false)
Variablen: m=number of clusters; $root$=root from archiv

1: $curr \leftarrow root$
2: **for** i=0...$m-1$ **do**
3: $pos \leftarrow sol[curr.cluster]$
4: **if** $curr.next[pos] = complete$ **then**
5: **return** false
6: **end if**
7: **if** $curr.next[pos] == null$ **then**
8: **if** $i == m-1$ **then**
9: $curr.next[pos] \leftarrow complete$
10: **end if**
11: $curr.next[pos] \leftarrow$ new trie-node
12: calculate forbidden pointers
13: **end if**
14: $invalidate(i, sol)$
15: **if** $bestsol <$ $calcBoundByDynProg(root.cluster, sol, i)$ **then**
16: delete $curr.next[pos]$
17: $curr.next[pos] \leftarrow complete$
18: check if trie-nodes are complete
19: **return** false
20: **end if**
21: $curr \leftarrow curr.next[pos]$
22: **end for**
23: check if trie-nodes are complete
24: **return** true

Der Ablauf der Insert-Methode wird in Algorithmus 10 gezeigt. Beim Einfügen einer Lösung wird, beginnend beim Wurzelknoten, immer dem Pointer des Trie-Knoten gefolgt, der dem Knoten aus dem jeweiligen Cluster in der Lösung entspricht. Ist ein solcher

Pointer noch $null$, wird ein Trie-Knoten erzeugt und an dieser Stelle in den Trie eingefügt. Beim ersten Aufruf der Methode $calcBoundByDynProg$ wird die Bound mithilfe der "normalen" dynamischen Programmierung ermittelt. Für alle weiteren Aufrufe der Methode $calcBoundByDynProg$ innerhalb derselben Einfügeoperation wird die inkrementelle Boundberechnung, die in Kapitel 2.2.1 beschrieben wird, verwendet. Dazu müssen vorher alle Cluster, für die sich die Kosten der Knoten ändern, markiert werden. Deshalb wird vor der Boundberechnung die Methode $invalidate$ aufgerufen. In dieser Methode werden alle Cluster, vom aktuellen Cluster bis hin zur Wurzel im Spannbaum der Lösung, markiert.

2.2.4 Konvertierungs-Methode

In der Konvertierungs-Methode des Pop-Archivs wird die Bounding Strategie ebenfalls angewendet. Die Konvertierungs-Methode mit Bounds funktioniert dabei ähnlich wie die des Gosh-Archivs. Bei jeder Änderung der Lösung wird die Bound berechnet und überprüft, ob diese besser als die beste bisher gefundene Lösung ist. Ist das nicht der Fall, wird die Änderung rückgängig gemacht und eine andere Lösung gesucht. Zusätzlich zur Bounding Strategie wurde die Konvertierungs-Methode durch den Nearest Neighbour Ansatz erweitert.

In Algorithmus 11 wird der Ablauf der Konvertierungs-Methode gezeigt. Dabei wird zunächst zufällig ein Trie-Knoten $curr$ aus den möglichen Startknoten entlang der Lösung ausgewählt. Im nächsten Schritt wird zuerst zufällig aus den Nearest Neighbours von $curr$ ein $null$-Pointer ausgewählt. Ist keiner vorhanden wird aus den restlichen Pointern zufällig ein $null$-Pointer ausgewählt. Wurde ein $null$-Pointer gefunden, kann eine neue Lösung generiert werden. Bei dieser neuen Lösung erfolgt dann eine Boundüberprüfung. Die Boundberechnung wird mit Methode $calcBoundByDynProg$ durchgeführt. Ist die Bound schlechter als die beste bisher gefundene Lösung, wird die Änderung der Lösung wieder rückgängig gemacht, der vorher ausgewählte Pointer auf $complete$ gesetzt und mit der Suche fortgesetzt. Ist die Bound nicht schlechter, kann abgebrochen werden. Wurde kein $null$-Pointer gefunden, wird zunächst versucht, entlang der Lösung im Archiv eine Ebene nach unten zu gehen. Hierbei wird wieder eine Boundüberprüfung durchgeführt. Kann entlang der Lösung nicht weiter nach unten gegangen werden, wird im aktuellen Trie-Knoten zufällig ein Pointer aus den Nearest Neighbours von $curr$ ausgewählt, der nicht $complete$ ist. Wird so ein Pointer nicht gefunden, wird aus den restlichen Pointern zufällig ein Pointer ausgewählt, der nicht $complete$ ist. Die Lösung wird dementsprechend geändert und es erfolgt wieder eine Boundüberprüfung.

Genauso wie bei der Insert-Methode in Kapitel 2.2.3, wird in dieser Methode beim ersten Aufruf von $calcBoundByDynProg$ die Bound mittels "normalen" dynamischen Programmierung ermittelt. Bei allen weiteren Boundberechnungen innerhalb derselben Konvertierungsoperation, wird die inkrementelle dynamische Programmierungs-Methode verwendet. Dazu wird vor jeder Boundberechnung die Methode $invalidate$ aufgerufen, die alle Cluster markiert, für die die Kosten neu berechnet werden müssen.

Algorithmus 11 PopTrie convert with Bound

Eingabe: solution to convert sol; best solution $bestsol$
Ausgabe: convertion successfull (true/false)
Variablen: $nn[i]$... Nearest Neighbours of Cluster i

1: $curr \leftarrow$ random trie-node of the possible startpoints from sol
2: **while** $curr! = null \;\&\&\; curr! = complete$ **do**
3: $p_{null} \leftarrow$ random $null$-pointer from $nn[curr]$, if none exist, random $null$-pointern from all pointer in $curr$
4: **if** $p_{null} exists$ **then**
5: change solution and check if cycle exist
6: $invalidate(curr.cluster, sol)$
7: **if** $bestsol <$ $calcBoundByDynProg(root.cluster, sol, curr.cluster)$ **then**
8: $curr.next[sol[curr.cluster]] \leftarrow complete$
9: undo changes of sol
10: **else**
11: **return** $true$
12: **end if**
13: **else**
14: **if** $curr.next[sol[curr.cluster]]! = complete]$ **then**
15: $invalidate(curr.cluster, sol)$
16: **if** $bestsol <$ $calcBoundByDynProg(root.cluster, sol, curr.cluster)$ **then**
17: delete $curr.next[sol[curr.cluster]]$ and set it $complete$
18: **else**
19: $curr \leftarrow curr.next[sol[curr.cluster]]$
20: **end if**
21: **else**
22: $sol[curr.cluster] \leftarrow$ random not-complete-pointer from $nn[curr]$, if none exist, random not-complete-pointer from all pointer in $curr$
23: $invalidate(curr.cluster, sol)$
24: **if** $bestsol <$ $calcBoundByDynProg(root.cluster, sol, curr.cluster)$ **then**
25: delete $curr.next[sol[curr.cluster]]$ and set it $complete$
26: undo changes of sol
27: **else**
28: $curr \leftarrow curr.next[sol[cur.cluster]]$
29: **end if**
30: **end if**
31: **end if**
32: **end while**

KAPITEL 3

Ergebnisse

3.1 Vorgehensweise

Die folgenden Tests wurden mit 14 TSPLib-Instanzen[1] durchgeführt. Diese Instanzen sind ursprünglich für das Traveling Salesman Problem erzeugt worden und wurden für das GMST-Problem angepasst. Dazu wurden Cluster der Instanz hinzugefügt und die einzelnen Knoten mittels geografischem Clustering zugeordnet. Die verwendeten Instanzen bestehen aus 150-442 Knoten und 30-89 Clustern. Im Schnitt hat also jedes Cluster 5 Knoten. Die erweiterten TSPLib-Instanzen wurden auch von Wolf [15] und Sonnleitner [13] verwendet, was einen Vergleich der Ergebnisse erleichtert.

Da der EA ein randomisierter Algorithmus ist, wurden für alle Tests 30 Runs gemacht und für die Auswertung der Mittelwert $C_{avg}(T) = 1/n \sum_{i=1}^{n} C(T_i)$ und die Standardabweichung $S = \sqrt{1/(n-1) \sum_{i=1}^{n} (C(T_i) - C_{avg})^2}$ genommen. Die Tests wurden auf dem Cluster des Institut für Computergraphik und Algorithmen der TU-Wien durchgeführt, welcher aus 14 Maschinen mit jeweils zwei QuadCore-CPUs und 24 GB Arbeitsspeicher besteht.

Grundsätzlich wurden zwei Arten von Tests durchgeführt. Zum einen sind Testläufe gemacht worden, bei denen der EA nach einer fixen Laufzeit terminiert und zum anderen Testläufe bei denen nach einer bestimmten Anzahl von Generationen der EA terminiert. Letztere wurden unter anderem dazu verwendet, um den Speicherverbrauch des Archivs genauer zu untersuchen.

[1] http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsp/index.html

3.2 Tests mit fixer Laufzeit

Die folgenden Tests wurden mit einer fixen Laufzeit durchgeführt. Die Daten der verwendeten Testinstanzen und die Laufzeit werden in Tabelle 1 dargestellt.

Tabelle 1: Testinstanzen und die verwendete Laufzeit

Instanz	Knoten	Cluster	Laufzeit[s]
kroa150	150	30	150
rat195	195	39	150
d198	198	40	150
krob200	200	40	150
ts225	225	45	200
pr226	226	46	200
gil262	262	53	300
pr264	264	54	300
pr299	299	60	300
lin318	318	64	400
rd400	400	80	450
fl417	417	84	450
pr439	439	88	600
pcb442	442	89	600

3.2.1 Analyse der Cuts

Zum Beginn der Tests sollte die Frage geklärt werden, ob und wie viele Teilbereiche des Archivs abgeschnitten (als $complete$ markiert) werden können. Außerdem sollte geklärt werden, in welchen Bereichen des Archivs Teilbereiche abgeschnitten werden, also Cuts gemacht werden. Cuts in den oberen Ebenen sind besser, da dadurch mehr Lösungen im vorhinein ausgeschlossen werden können als bei Cuts in den unteren Ebenen. Dieser Test wurde für beide Archive (Pop- und Gosh-Archiv) und für alle 14 Test-Instanzen durchgeführt. Dabei wurde der EA nach einer konstanten Zeit terminiert.

In Tabelle 2 werden die Ergebnisse dieses Tests dargestellt. In dieser Tabelle wird gezeigt in welchem Teil des Archivs wie viele Cuts durchgeführt wurden. Die Spalte $0 - 25\%$ gibt an, wie viele Cuts in den oberen 25% der Ebenen im Archiv durchgeführt wurden, in der Spalte $25 - 50\%$ für die nächsten 25%, usw.

Tabelle 2: Anzahl der Cuts in den unterschiedlichen Bereichen des Archivs

	Gosh-Archiv				Pop-Archiv			
Instanz	0-25%	25-50%	50-75%	75-100%	0-25%	25-50%	50-75%	75-100%
kroa150	0	0	26	11.081	61	857	10.721	23.394
rat195	0	0	1	5.982	32	635	3.376	14.438
d198	0	0	3	5.103	410	5.500	11.913	14.468
krob200	0	0	1	5.376	19	243	1.193	7.590
ts225	0	0	0	3.362	0	1	99	5.646
pr226	0	0	3	3.301	721	6.687	12.744	15.731
gil262	0	0	5	6.937	13	295	3.067	9.529
pr264	0	0	10	5.473	525	5.764	14.587	13.011
pr299	0	0	1	3.318	45	544	2.541	9.306
lin318	0	0	6	5.187	332	3.119	12.809	16.113
rd400	0	0	1	3.475	55	637	4.898	9.744
fl417	0	0	0	1.835	2.741	14.811	10.400	7.802
pr439	0	0	1	3.141	457	2.238	4.790	7.617
pcb442	0	0	1	3.349	12	360	2.510	6.367

Hier ist zu sehen, dass im Gosh-Archiv in den ersten 50% keine Bound gefunden wird, die schlechter ist als die beste bisher gefunden Lösung, und somit auch kein Cut gemacht werden kann. Da in den ersten 50% keine Cuts gemacht werden, ist es auch nicht nötig die Bound in diesem Bereich zu berechnen, da das nur unnötig Laufzeit verbraucht. Deshalb wurde der Parameter $skip\_bound$ eingeführt, mit dem angegeben werden kann, in wie viel Prozent der oberen Ebenen die Bound nicht berechnet werden soll. Aufgrund der erhaltenen Ergebnisse, wurde für die folgenden Tests der Parameter $skip\_bound$ für das Gosh-Archiv auf $0,5$ gesetzt. Das bedeutet, dass für die oberen 50% des Archivs wird die Bound nicht berechnet.

Im Pop-Archiv sieht es hingegen anders aus. Hier kommt es auch in den oberen Ebenen zu Cuts. Die Frage, die sich jetzt stellt, ist: Wieso werden im Pop-Archiv in den oberen Ebenen Cuts gemacht aber im Gosh-Archiv nicht? Die Antwort auf die Frage findet sich in den unterschiedlichen Kodierungen der Lösung, die in den Archiven verwendet werden. Bei der Gosh-Kodierung wird für jeden Cluster gespeichert, welcher Knoten in dem Cluster ausgewählt wird. Wird ein Eintrag für einen Cluster geändert, d.h. ein anderer Knoten in dem Cluster ausgewählt, werden die Kosten der Gesamtlösung nicht stark ansteigen, da die Knoten innerhalb eines Clusters relativ nahe zusammen liegen. Bei der Pop-Kodierung wird für jeden Cluster der Vorgänger im Spannbaum des glob-

alen Graphen gespeichert. Kommt es hier zu einer Änderung für einen Cluster, können die Kosten der Gesamtlösung stark ansteigen, falls die Cluster im Graphen weit auseinander liegen. D.h. wenn in den oberen Ebenen schon ein Vorgänger für einen Cluster ausgewählt wird, der weiter entfernt liegt, kann das schon zu einer schlechten Bound führen. Die unterschiedliche Kodierung ist auch der Grund, warum im Pop-Archiv mehr Cuts gemacht werden als im Gosh-Archiv.

3.2.2 Gosh-Archiv

In den nächsten Tests wurde untersucht, wie sich die Bounding-Strategie auf das Gosh-Archiv auswirkt. Dabei wurden drei Varianten betrachtet: Bounding-Strategie innerhalb der Einfüge-Methode, Bounding-Strategie innerhalb der Konvertierungs-Methode und die Variante, in der sie in beiden Methoden gleichzeitig verwendet wird. Dabei wurde der Parameter $skip\_bound$ auf $0,5$ gesetzt. Für den Parameter $branch\_and\_bound\_prob$ wurden Tests gemacht, bei denen der Wert dieses Parameters variiert wurde, um festzustellen, welcher Wert die besten Ergebnisse liefert. Dabei hat sich heraus gestellt, dass bei $branch\_and\_bound\_prob > 0,1$ die Anzahl der erzeugten Generationen zu stark abnimmt und somit schlechtere Ergebnisse geliefert werden. Auf die Ergebnisse der Tests mit den Werten $0,1$ und $0,05$ wurde ein Wilcoxon-Rangsummenstatistik-Test durchgeführt mit dem Resultat, dass beide Werte ähnliche Ergebnisse liefern, die sich statistisch nicht signifikant unterscheiden. Da mit einem Wert von $0,05$ aber in mehr Fällen bessere Resultate erzielt wurden, wurde für die folgenden Tests dieser Wert genommen.

Die Ergebnisse dieses Tests sind in Tabelle 3 zu sehen. In der Tabelle werden in der Spalte $Trie$ die Ergebnisse des "normalen" Gosh-Archivs ohne Bounding-Strategie, unter $Trie+insert-Bound$ die Variante mit Bounding-Strategie innerhalb der Einfüge-Methode, im Bereich $Trie + convert - Bound$ die Bounding-Strategie innerhalb der Konvertierungs-Methode und unter $Trie + ins\&conv - Bound$ bei der die Bounding-Strategie in beiden Methoden gleichzeitig verwendet wird dargestellt. C_{avg} entspricht dem durchschnittlichen Lösungswert über alle Runs, darunter in Klammern steht die Standardabweichung und $\overline{Gen}$ ist die durchschnittliche Anzahl der erzeugten Generationen.

In den Ergebnissen ist zu sehen, dass alle drei Bounding Varianten gegenüber des "normalen" Archivs bessere Ergebnisse liefern. Die Variante, in der die Bounding Strategie in beiden Methoden verwendet wurde, liefert jedoch gegenüber der beiden anderen Varianten schlechtere Ergebnisse. Dieses Resultat ist auf die geringere Anzahl von erzeugten Generationen zurück zu führen. In einigen Instanzen wurden nur halb so viele Generationen erzeugt als in den anderen Varianten.

Tabelle 3: Vergleich der verschiedenen Bounding-Varianten im Gosh-Archiv

	Trie		Trie+insert-Bound		Trie+convert-Bound		Trie+ins&conv-Bound	
Instanz	$C_{avg}(T)$	$\overline{Gen}$	$C_{avg}(T)$	$\overline{Gen}$	$C_{avg}(T)$	$\overline{Gen}$	$C_{avg}(T)$	$\overline{Gen}$
kroa150	9.822,0 (21,4)	60.747	9.822,0 (12,7)	52.511	9.819,9 (17,7)	29.509	**9.819,6** (17,8)	26.609
rat195	754,4 (4,7)	33.232	**752,7** (3,1)	28.739	752,9 (3,5)	17.602	753,9 (4,1)	14.481
d198	7.054,6 (7,6)	29.272	**7.051,5** (5,3)	23,219	7.052,4 (4,5)	14.513	7.053,3 (3,5)	12.707
krob200	11.248,8 (8,7)	31.790	11.246,1 (6,3)	22.863	**11.245,3** (5,1)	16.887	11.260,6 (11,0)	14.727
ts225	62.306,3 (59,3)	32.698	**62.275,4** (17,7)	26.467	62.286,5 (23,9)	20.472	62.292,1 (46,5)	18.346
pr226	**55.515,0** (0,0)	31.541	**55.515,0** (0,0)	25.289	**55.515,0** (0,0)	12.098	**55.515,0** (0,0)	10.576
gil262	944,5 (3,8)	37.349	**942,7** (2,2)	22.549	944,1 (4,3)	19.123	942,8 (2,7)	15.878
pr264	21.895,1 (11,7)	33.368	21.891,1 (6,6)	21.467	**21.890,1** (6,0)	16.071	21.890,6 (8,4)	11.679
pr299	20.354,3 (41,0)	26.404	**20.339,7** (21,6)	15.985	20.346,2 (30,0)	13.081	20.341,0 (25,4)	11.389
lin318	18.540,5 (25,4)	27.309	18.535,1 (21,2)	21.289	18.532,4 (16,3)	13.220	**18.528,0** (15,1)	10.738
rd400	5.950,6 (13,9)	21.185	**5.945,4** (10,8)	14.815	5.947,6 (13,4)	9.260	5.945,6 (11,5)	8.261
fl417	**7.982,0** (0,0)	22.146	**7.982,0** (0,0)	15.208	**7.982,0** (0,0)	6.324	**7.982,0** (0,0)	5.457
pr439	51.873,9 (54,5)	17.653	51.828,2 (35,5)	7.855	51.827,0 (35,7)	10.339	**51.826,3** (26,7)	8.651
pcb442	19.667,2 (38,0)	18.106	19.657,4 (40,3)	7.576	**19.648,9** (27,3)	10.707	19.654,5 (34,3)	8.707

3.2.3 Nearest Neighbours

Als nächstes wurde versucht herauszufinden, wie sich der Nearest-Neighbours-Ansatz, der in Kapitel 2.2.2 erläutert wurde, auf den Evolutionären Algorithmus auswirkt. Bei diesen Tests wurde der EA ohne Archiv verwendet. Dabei wurde der EA einmal mit und einmal ohne dem Nearest-Neighbours-Ansatz verwendet und miteinander verglichen.

Bei diesem Test wurde der Parameter $num\_nearest\_neighbour$, der angibt, wie viele Nachbarn in der Nearest-Neighbour-List eines Clusters enthalten sind, auf 5 gesetzt. D.h. es werden für jeden Cluster die 5 nächsten Nachbarn bevorzugt. In Tabelle 4 sind die Ergebnisse dieses Tests dargestellt. Hier ist zu sehen, dass der EA mit dem Nearest-Neighbours-Ansatz für alle Instanzen eine Verbesserung liefert. Für die nachfolgenden Tests wurde daher der Nearest-Neighbours-Ansatz auch verwendet.

Tabelle 4: Vergleich von "normalen" EA mit einem EA mit Nearest Neighbours

	EA		EA+NN	
Instanz	$C_{avg}(T)$	$\overline{Gen}$	$C_{avg}(T)$	$\overline{Gen}$
kroa150	9.831,8 (30,9)	302.576	**9.831,5** (30,3)	303.589
rat195	761,1 (6,3)	199.953	**760,3** (5,8)	199.391
d198	7.068,7 (10,9)	183.405	**7.057,3** (14,1)	180.270
krob200	11.291,4 (45,3)	185.572	**11.277,6** (36,2)	185.052
ts225	62.565,7 (131,1)	213.360	**62.456,7** (152,7)	213.926
pr226	**55.515,0** (0,0)	199.954	**55.515,0** (0,0)	200.307
gil262	949,8 (5,6)	244.354	**946,8** (5,1)	243.635
pr264	21.950,9 (29,9)	229.337	**21.936,9** (32,2)	227.352
pr299	20.411,4 (55,6)	187.223	**20.369,1** (59,7)	185.952
lin318	18.553,7 (33,9)	223.100	**18.550,4** (30,3)	221.801
rd400	5.993,6 (33,0)	166.799	**5.978,6** (32,6)	165.978
fl417	7.996,0 (6,9)	148.236	**7.989,5** (8,2)	147.630
pr439	52.045,4 (124,9)	178.455	**52.016,5** (109,2)	177.110
pcb442	19.830,7 (113,7)	180.509	**19.762,4** (113,6)	181.069

Tabelle 5: Vergleich der verschiedenen Bounding-Varianten im Pop-Archiv

	Trie		Trie+insert-Bound		Trie+convert-Bound		Trie+ins&conv-Bound	
Instanz	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$
kroa150	**9.815,0** (0,0)	62.747	**9.815,0** (0,0)	44.525	**9.815,0** (0,0)	32.376	**9.815,0** (0,0)	24.844
rat195	752,5 (3,0)	36.732	**751,5** (2,0)	25.765	751,9 (2,5)	22.772	752,2 (2,7)	16.854
d198	7.046,1 (3,9)	30.787	**7.044,0** (0,0)	20.328	**7044,0** (0,0)	14.011	7.044,3 (1,6)	10.249
krob200	11.245,3 (5,1)	34.399	**11.244,0** (0,0)	24.306	**11.244,0** (0,0)	21.614	11.246,0 (6,1)	17.198
ts225	62.268,7 (0,5)	38.580	**62.268,2** (0,4)	28.141	62.268,5 (0,5)	29.270	62.268,5 (0,5)	22.654
pr226	**55.515,0** (0,0)	33.056	**55.515,0** (0,0)	19.713	**55.515,0** (0,0)	17.736	**55.515,0** (0,0)	10.571
gil262	942,4 (2,0)	38.873	**942,0** (0,0)	25.842	942,3 (2,0)	24.078	942,7 (2,7)	18.201
pr264	**21.886,0** (0,0)	36.538	**21.886,0** (0,0)	23.104	**21.886,0** (0,0)	22.734	**21.886,0** (0,0)	13.226
pr299	20.318,6 (14,2)	28.971	20.316,1 (0,5)	18.548	**20.316,0** (0,0)	20.506	20.320,6 (17,5)	13.950
lin318	18.525,5 (15,0)	34.659	18.523,9 (9,4)	17.283	18.523,0 (14,3)	22.782	**18.522,2** (10,5)	11.108
rd400	5.945,2 (16,5)	25.867	5.943,2 (7,9)	14.627	**5.942,5** (6,9)	19.263	5.942,7 (7,2)	10.666
fl417	**7.982,0** (0,0)	23.200	**7.982,0** (0,0)	8.683	**7.982,0** (0,0)	13.515	**7.982,0** (0,0)	2.578
pr439	51.796,5 (29,7)	24.933	51.794,0 (16,6)	14.215	51.793,5 (10,8)	20.692	**51.791,6** (2,7)	9.734
pcb442	19.630,2 (23,6)	26.755	19634,4 (20,2)	15.859	19.627,7 (15,7)	22.892	**19.627,4** (15,6)	13.407

3.2.4 Pop-Archiv

In diesem Kapitel wurde der Vergleich der verschiedenen Bounding-Strategien auch für das Pop-Archiv durchgeführt. In Kapitel 3.2.1 wurde in dem Test herausgefunden, dass Cuts in allen Ebenen des Tries vorkommen können. Deshalb wurde für den Vergleich der Bounding-Strategien der Parameter $skip\_bound$ hier nicht verwendet, d.h. die

Bound wurde in jeder Ebene berechnet. Für den Parameter $branch\_and\_bound\_prob$ wurden für das Pop-Archiv, genau wie im Kapitel 3.2.2 für das Gosh-Archiv, Tests mit unterschiedlichen Werten für diesen Parameter gemacht. Mit dem Resultat, dass auch hier der Wert $0,05$ die besseren Ergebnisse liefert.

Tabelle 5 zeigt die Ergebnisse für diesen Test. Hier ist zu sehen, dass die Varianten mit den Bounding-Strategien meistens besser sind als die Variante mit "normalem" Archiv. Außer für die Instanzen $kroa150$, $pr226$, $pr264$ und $fl417$ für die auch mit dem Archiv ohne Bounding-Strategie das Optimum erreicht wurde.

3.2.5 Beide Archive

In [13] wurde eine Variante getestet, bei der beide Archive gleichzeitig verwendet wurden. In dieser Arbeit soll auch getestet werden, wie sich die Bounding-Strategie in den verschiedenen Varianten auswirkt, wenn beide Archive gleichzeitig verwendet werden. Der Parameter $skip\_bound$ wird auf $0,5$ gesetzt. Dieser hat aber nur Auswirkungen auf das Gosh-Archiv, d.h. im Pop-Archiv werden in allen Ebenen Bounds berechnet. Der Parameter $branch\_and\_bound\_prob$ wurde, genau wie bei den vorherigen beiden Tests, auf $0,05$ gesetzt.

In Tabelle 6 sind die Ergebnisse dieses Tests zu sehen. Dabei wurden wieder 4 Varianten miteinander verglichen: "normalen" Gosh und Pop Archive ohne Bounding-Strategie ($Trie$), die Variante mit Bounding-Strategie innerhalb der Einfüge-Methode ($Trie + insert-Bound$), die Bounding-Strategie innerhalb der Konvertierungs-Methode ($Trie+ convert-Bound$) und die Bounding-Strategie in beiden Methoden gleichzeitig ($Trie+ ins\&conv - Bound$). Hier ist zu sehen, dass bessere Ergebnisse bei den Varianten mit Bounding-Strategie erzielt werden, außer bei den Instanzen bei denen das Optimum schon mit den "normalen" Archiven erreicht wird. Außerdem ist zu sehen, dass in der Variante, in der in beiden Methoden die Bounding-Strategie verwendet wird, schlechtere Ergebnisse erreicht werden, als in den Varianten, wo die Bounding-Strategie nur bei der Einfüge- oder Konvertierungs-Methode verwendet wird. Das ist auf die geringere Anzahl von Generationen zurück zu führen, die oft nur halb so groß ist als bei den beiden anderen Varianten.

Tabelle 6: Vergleich der verschiedenen Bounding-Varianten bei der Kombination von Gosh- und Pop Archiv

	Trie		Trie+insert-Bound		Trie+convert-Bound		Trie+ins&conv-Bound	
Instanz	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$	$C_{avg}(T)$	$\overline{\lvert Gen\rvert}$
kroa150	**9.815,0** (0,0)	21.637	**9.815,0** (0,0)	14.801	**9.815,0** (0,0)	9.853	**9.815,0** (0,0)	7.638
rat195	752,1 (2,8)	18.540	**751,3** (1,4)	11.167	752,8 (3,4)	9.476	752,6 (3,2)	5.372
d198	**7.044,0** (0,0)	11.199	**7.044,0** (0,0)	7.149	**7.044,0** (0,0)	5.418	**7.044,0** (0,0)	3.924
krob200	**11.244,0** (0,0)	16.633	**11.244,0** (0,0)	10.592	**11.244,0** (0,0)	10.370	**11.244,0** (0,0)	7.531
ts225	62.268,5 (0,5)	17.676	**62.268,3** (0,5)	11.333	**62.268,3** (0,5)	13.357	**62.268,3** (0,5)	9.474
pr226	**55.515,0** (0,0)	1.148	**55.515,0** (0,0)	625	**55.515,0** (0,0)	416	**55.515,0** (0,0)	323
gil262	**942,0** (0,0)	20.694	**942,0** (0,0)	11.228	**942,0** (0,0)	11.494	**942,0** (0,0)	8.025
pr264	**21.886,0** (0,0)	16.198	**21.886,0** (0,0)	9.744	**21.886,0** (0,0)	9.458	**21.886,0** (0,0)	6.099
pr299	**20.316,0** (0,0)	15.475	**20.316,0** (0,0)	8.480	**20.316,0** (0,0)	9.625	**20.316,0** (0,0)	6.671
lin318	18.522,9 (9,2)	16.954	**18.515,9** (12,4)	8.576	18.517,7 (8,7)	8.670	18.519,5 (8,3)	5.218
rd400	5.940,3 (6,4)	14.236	**5.937,7** (5,8)	6.692	5.938,9 (7,6)	8.545	5.939,9 (5,6)	5.184
fl417	**7.982,0** (0,0)	1.486	**7.982,0** (0,0)	875	**7.982,0** (0,0)	585	**7.982,0** (0,0)	418
pr439	**51.791,0** (0,0)	12.837	51.792,2 (6,4)	6.101	**51.791,0** (0,0)	9.331	**51.791,0** (0,0)	4.716
pcb442	19.625,8 (19,1)	17.457	19.623,6 (17,4)	8.198	**19.620,5** (18,7)	10.872	19.622,8 (18,3)	6.961

3.3 Fixe Anzahl von Generationen

In den bisherigen Tests ist zu sehen, dass durch die Verwendung der Bounding-Strategie die Anzahl der erzeugten Generationen geringer sind. Dadurch ist ein Vergleich des Speicherverbrauchs der beiden Varianten nicht möglich, da bei dem Archiv ohne Bounding mehr Lösungen in das Archiv eingefügt werden als beim Archiv mit Bounding. Um jetzt den Speicherverbrauch der unterschiedlichen Varianten miteinander zu vergleichen, wird als Abbruchbedingung für den EA eine fixe Anzahl von Generationen genommen. So werden bei allen Varianten gleich viele Generationen erzeugt.

Für diese Tests wurde die Abbruchbedingung des EAs auf 10.000 Generationen gesetzt. In den Tabellen 7 und 8 sind die Ergebnisse für diese Tests zu sehen. In der Tabelle 7 wurde das Gosh-Archiv untersucht, während in Tabelle 8 die Ergebnisse für das Pop-Archiv zu sehen sind. Es wurden jeweils das "normale" Archiv ohne Bounding-Strategie, das mit Bounding-Strategie in der Einfüge-Methode und das mit Bounding-Strategie in der Konvertierungs-Methode miteinander verglichen. Die Spalte C_{avg} entspricht dabei wieder dem Durchschnitt der Lösungswerte über alle Runs, unter diesem Wert steht in Klammern die Standardabweichung, $\overline{Zeit}$ entspricht der durchschnittlich benötigten Zeit und $\overline{Mem}$ den benötigten Speicher für das Archiv.

In diesen Ergebnissen ist zu sehen, dass die Varianten mit Bound immer mehr Speicher benötigen als die ohne Bound. Das kommt zunächst überraschend, da man zuerst annehmen würde, dass durch die Bounding-Strategie auch weniger Speicher verbraucht würde. Der Grund für den erhöhten Speicherverbrauch bei der Einfüge-Methode ist in Abb. 8 zu sehen. In diesem Beispiel wird zunächst versucht, im Gosh-Archiv die Lösung <321112> (in der Grafik rot dargestellt) einzufügen. In Knoten $V5$ wird eine Bound festgestellt, die schlechter ist als die beste bisher gefundene Lösung, und somit der Pointer 1 als $complete$ markiert. Danach wird die Einfügeoperation abgebrochen und $false$ zurück geben. Für den EA sieht es daher so aus, als wäre die Lösung im Archiv schon enthalten. Im nächsten Schritt wird mit der Konvertierungsmethode eine neue Lösung <341112> erzeugt. Diese wird nun erfolgreich in das Archiv eingefügt. Würde die Bounding-Strategie nicht verwendet werden, wäre die Lösung <321112> beim ersten Mal eingefügt worden. Da die Bounding-Strategie verwendet wurde, ist nun die Lösung <321112> bis zum Knoten $V5$ und die Lösung <341112> ganz eingefügt worden. D.h. beim Erzeugen einer neuen Lösung für die neue Generation wurden die drei Knoten $V3$, $V4$ und $V5$ des ersten Einfügeversuchs zusätzlich eingefügt. Deshalb kommt es bei der Verwendung der Bounding-Strategie zu einem höheren Speicherverbrauch.

Um zu belegen, dass das der Grund für den erhöhten Speicherverbrauch ist, wurden weitere Tests durchgeführt. Bei diesen Tests wurden Lösungen, bei denen eine Bound

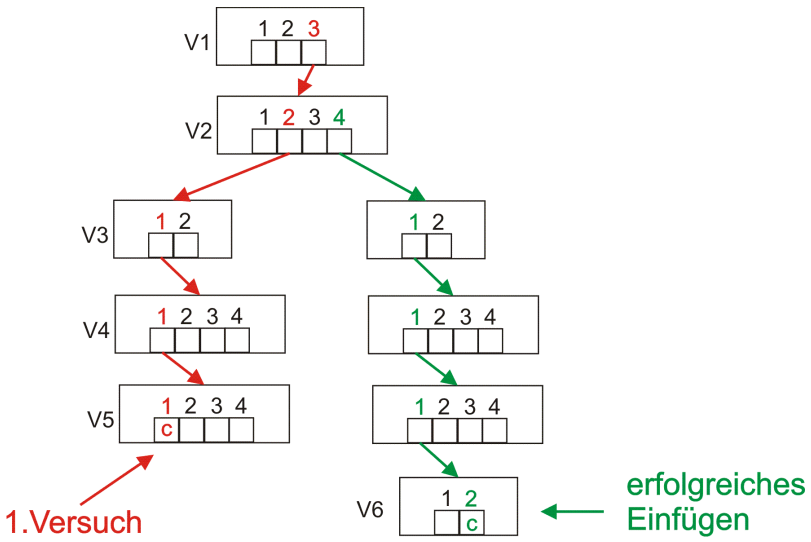

Abb. 8: Grund für erhöhten Speicherverbrauch

gefunden wurde, die schlechter ist als die beste bisher gefundene Lösung, trotzdem in der neuen Generation akzeptiert. D.h. wenn beim Einfügen einer Lösung eine schlechte Bound gefunden wird, wird der jeweilige Pointer als *complete* markiert und dann *true* zurück gegeben. Das hat zur Folge, dass der EA die Lösung als neu akzeptiert.

Die Ergebnisse dieser Variante werden in Tabelle 9 gezeigt. Hier ist zu sehen, dass für die meisten Instanzen bei dieser Variante weniger Speicher für das Archiv gebraucht wurde.

Bei der Bounding-Strategie in der Konvertierungs-Methode ist der Grund für den erhöhten Speicherverbrauch ähnlich wie bei der Einfüge-Methode. Wenn bei der Konvertierungs-Methode in einem zufällig ausgewählten Startknoten der Lösung keine *null*-Pointer gefunden werden, wird versucht, entlang der Lösung im Trie eine Ebene nach unten zu gehen. Wenn das nicht möglich ist, wird versucht bei einem anderen nicht *complete*-Pointer eine Ebene nach unten zu gehen. Wenn jetzt aufgrund von Bounds, die schlechter als die beste bisher gefunden Lösung sind, in diesem neuen Knoten alle Pointer *complete* werden, wird die Konvertierung abgebrochen und an einer anderen Stelle nochmal versucht. Da der Pointer der Lösung in dem Startknoten in diesem Fall auf *complete* gesetzt wurde, wird beim nächsten Konvertierungsversuch der Startknoten aus den darüber liegenden Ebenen ausgewählt. D.h. bei der Verwendung der Bounding-Strategie ist die Wahrscheinlichkeit höher, dass in den oberen Ebenen die Konvertierung durchgeführt wird. Das führt dazu, dass beim anschließenden Einfügen einer Lösung der neu einzufügende Subtrie größer ist. Deshalb werden mehr Trie-Knoten mit der

Bounding-Strategie erzeugt und somit kommt es auch zu einem erhöhten Speicherverbrauch.

Tabelle 7: Vergleich von Gosh-Trie mit 10000 Generationen

	Trie			Trie+insert-Bound			Trie+convert-Bound		
Instanz	$C_{avg}(T)$	$\overline{Zeit}$ $[s]$	$\overline{Mem}$ $[MB]$	$C_{avg}(T)$	$\overline{Zeit}$ $[s]$	$\overline{Mem}$ $[MB]$	$C_{avg}(T)$	$\overline{Zeit}$ $[s]$	$\overline{Mem}$ $[MB]$
kroa150	9.828,8 (27,5)	23	6,7	9.819,3 (11,4)	36	8,0	9.823,1 (17,8)	33	6,6
rat195	755,9 (5,2)	39	10,7	758,3 (5,6)	75	12,5	757,6 (5,0)	52	11,4
d198	7.061,7 (9,5)	49	8,2	7.062,9 (9,0)	82	9,7	7.063,3 (8,4)	76	8,4
krob200	11.269,1 (25,2)	42	10,0	11.273,9 (42,0)	68	11,6	11.279,9 (48,3)	58	10,3
ts225	62.422,1 (106,2)	49	17,3	62.406,5 (92,5)	73	19,0	62.404,7 (101,6)	65	16,6
pr226	55.515,0 (0,0)	64	6,8	55.515,0 (0,0)	126	7,7	55.515,0 (0,0)	128	6,7
gil262	948,9 (5,1)	74	17,9	947,5 (4,7)	123	21,3	948,4 (4,3)	95,6	18,2
pr264	21.913,2 (31,4)	80	15,1	21.905,0 (18,2)	139	17,8	21.914,9 (26,1)	118	15,3
pr299	20.402,5 (55,4)	101	18,8	20.398,5 (45,7)	174	22,1	20.370,9 (39,3)	144	19,1
lin318	18.548,9 (26,6)	113	17,8	18.545,6 (22,1)	217	22,1	18.556,1 (27,5)	183	17,7
rd400	5.977,7 (22,3)	168	25,8	5.974,1 (25,9)	318	31,9	5.981,9 (21,0)	263	25,7
fl417	7.982,0 (0,0)	204	18,1	7.982,0 (0,0)	499	20,5	7.982,0 (0,0)	478	18,3
pr439	51.954,4 (54,3)	243	28,9	51.949,2 (50,3)	466	34,3	51.947,1 (64,4)	365	62,8
pcb442	19.829,7 (105,0)	217	34,4	19.791,0 (75,2)	406	41,1	19.837,9 (109,1)	309	34,5

Tabelle 8: Vergleich von Pop-Trie mit 10000 Generationen

	Trie			Trie+insert-Bound			Trie+convert-Bound		
Instanz	$C_{avg}(T)$	$\overline{Zeit}$ $[s]$	$\overline{Mem}$ $[MB]$	$C_{avg}(T)$	$\overline{Zeit}$ $[s]$	$\overline{Mem}$ $[MB]$	$C_{avg}(T)$	$\overline{Zeit}$ $[s]$	$\overline{Mem}$ $[MB]$
kroa150	9.815,0 (0,0)	24	28,9	9.815,0 (0,0)	35	42,3	9.815,0 (0,0)	37	32,8
rat195	751,6 (2,1)	41	55,4	751,5 (2,0)	64	84,1	752,1 (2,8)	58	59,5
d198	7.046,5 (4,0)	49	49,0	7.045,4 (3,3)	90	95,9	7.044,6 (2,3)	99	54,1
krob200	11.246,1 (6,2)	44	58,3	11.246,3 (6,4)	66	75,7	11.244,7 (3,7)	55	61,2
ts225	62.268,5 (0,5)	53	84,7	62.268,5 (0,5)	74	96,9	62.268,4 (0,5)	61	85,3
pr226	55.515,0 (0,0)	61	76,1	55.515,0 (0,0)	140	171,2	55.515,0 (0,0)	100	89,3
gil262	942,4 (2,0)	78	111,4	942,5 (2,3)	128	158,3	942,4 (2,0)	93	115,8
pr264	21.886,0 (0,0)	83	108,5	21.886,0 (0,0)	169	210,9	21.886,0 (0,0)	106	115,9
pr299	20.325,3 (19,5)	105	133,6	20.318,6 (9,3)	202	225,2	20.317,7 (7,8)	129	137,4
lin318	18.521,4 (13,2)	116	163,5	18.519,5 (14,2)	298	384,1	18.524,2 (12,8)	152	174,2
rd400	5.943,0 (9,7)	178	264,7	5.942,8 (7,1)	502	600,1	5.944,7 (7,8)	212	276,9
fl417	7.982,0 (0,0)	196	265,0	7.982,0 (0,0)	1.041	845,7	7.982,0 (0,0)	326	302,5
pr439	51.792,3 (6,7)	248	309,3	51.793,0 (8,0)	687	679,3	51.791,1 (0,3)	267	315,3
pcb442	19.628,9 (20,5)	229	360,0	19.628,1 (19,1)	527	615,6	19.633,6 (21,4)	244	367,4

Tabelle 9: Vergleich von Pop-Tries mit 10000 Generationen ohne return

	Trie			Trie+insert-Bound		
Instanz	$C_{avg}(T)$	$\overline{Zeit}\ [s]$	$\overline{Mem}\ [MB]$	$C_{avg}(T)$	$\overline{Zeit}\ [s]$	$\overline{Mem}\ [MB]$
kroa150	9.815,0 (0,0)	24	28,9	9.815,0 (0,0)	36	29,2
rat195	751,6 (2,1)	41	55,4	751,5 (2,0)	61	54,6
d198	7.046,5 (4,0)	49	49,0	7.045,6 (3,4)	89	44,4
krob200	11.246,1 (6,2)	44	58,3	11.244,7 (3,7)	62	58,3
ts225	62.268,5 (0,5)	53	84,7	62.268,4 (0,5)	71,9	82,8
pr226	55.515,0 (0,0)	61	76,1	55.515,0 (0,0)	106	75,8
gil262	942,4 (2,0)	78	111,4	942,3 (2,0)	111	111,9
pr264	21.886,0 (0,0)	83	108,5	21.886,0 (0,0)	121	101,8
pr299	20.325,3 (19,5)	105	133,6	20.320,3 (13,0)	158	130,9
lin318	18.521,4 (13,2)	116	163,5	18.522,4 (11,2)	183	156,8
rd400	5.943,0 (9,7)	178	264,7	5.944,7 (7,7)	292	259,6
fl417	7.982,0 (0,0)	196	265,0	7.982,0 (0,0)	384	239,5
pr439	51.792,3 (6,7)	248	309,3	51.791,0 (0,0)	377,9	299,2
pcb442	19.628,9 (20,5)	229	360,0	19.630,5 (21,8)	354	355,2

3.4 State of the Art

In Tabelle 10 wurden die Ergebnisse dieser Arbeit mit den Ergebnissen anderer Arbeiten verglichen, um zu sehen, wie der EA Ansatz mit Lösungsarchiv und Bounding-Strategie im Vergleich abschneidet. Die Ergebnisse wurden aus der Arbeit [5] übernommen. In

Tabelle 10 werden vier Verfahren miteinander verglichen: ein Tabu Search Verfahren (TS) von [2], ein Variable Neighborhood Search Verfahren (VNS) von [4], ein Verfahren basierend auf Dynamic Candidates Sets (DCS) von [6] und einem EA mit einem Archiv in Kombination mit einer Bounding-Strategie. Bei dem Verfahren mit Boundig-Strategie wurden die Ergebnisse genommen, bei denen beide Archive kombiniert wurden und die Bound nur in der Konvertierungs-Methode verwendet wurde. Hier ist zu sehen, dass das Verfahren mit Bounding-Strategie durchaus mit den anderen Verfahren mithalten kann und in einigen Instanzen auch bessere Ergebnisse liefert.

Tabelle 10: Vergleich mit anderen State of the Art Verfahren

Instanz	TS $C_{avg}(T)$	VNS $C_{avg}(T)$	DCS $C_{avg}(T)$	EA+Archiv+Bound $C_{avg}(T)$
kroa150	**9.815,0**	**9.815,0** (0,0)	**9.815,0** (0,0)	**9.815,0** (0,0)
d198	7.062,0	**7.044,0** (0,0)	**7.044,0** (0,0)	**7.044,0** (0,0)
krob200	11.245,0	**11.244,0** (0,0)	**11.244,0** (0,0)	**11.244,0** (0,0)
ts225	62.366,0	62.268,5 (0,5)	**62.268,3** (0,5)	**62.268,3** (0,5)
pr226	**55.515,0**	**55.515,0** (0,0)	**55.515,0** (0,0)	**55.515,0** (0,0)
gil262	**942,0**	942,3 (1,0)	**942,0** (0,0)	**942,0** (0,0)
pr264	**21.886,0**	21.886,5 (1,8)	**21.886,0** (0,0)	**21.886,0** (0,0)
pr299	20.339,0	20.322,6 (14,7)	20.317,4 (1,5)	**20.316,0** (0,0)
lin318	18.521,0	**18.506,8** (11,6)	18.513,6 (7,8)	18.517,7 (8,7)
rd400	5.943,0	5.943,6 (9,7)	5.941,5 (9,9)	**5.938,9** (7,6)
fl417	7.990,0	**7.982,0** (0,0)	7.982,7 (0,5)	**7.982,0** (0,0)
pr439	51.852,0	51.847,9 (40,9)	51.833,8 (36,0)	**51.791,0** (0,0)
pcb442	19.621,0	19.702,8 (52,1)	19.662,5 (39,8)	**19.620,5** (18,7)

KAPITEL 4

Zusammenfassung

In dieser Arbeit wurde ein Evolutionärer Algorithmus mit zwei verschiedenen Varianten von Lösungsarchiven (Gosh-Archiv und Pop-Archiv) durch eine Bounding-Strategie basierend auf dem Branch and Bound Verfahren erweitert. Zusätzlich wurde auch ein Nearest Neighbour Ansatz angewendet, bei dem beim Anhängen eines Clusters an den Spannbaum die n nächsten Nachbarcluster bevorzugt werden.

Bei der Bounding-Strategie werden in den einzelnen Trie-Knoten geeignete Bounds berechnet, die eine Aussage darüber machen können, wie gut die Lösungen im darunter liegenden Subtrie bestenfalls sein können. Mit diesen Bounds können viele Lösungen von vornherein als schlecht markiert werden. Die Boundberechnung erfolgt in 3 verschiedenen Varianten: Boundberechnung in der Einfüge-Methode des Archivs, in der Konvertierungs-Methode und in beiden Methoden gleichzeitig.

Diese Varianten wurden in den beiden Archiven getestet und verglichen. Im Gosh-Archiv haben die Bounding Varianten in allen Instanzen bessere Ergebnisse erzielt, als bei der “normalen” Variante. Die Variante, in der in beiden Methoden die Bounding-Strategie verwendet wird, hat jedoch schlechtere Ergebnisse gebracht als in den anderen beiden Bounding-Varianten. Das ist auf die geringere Anzahl von erzeugten Generationen zurück zu führen. Im Pop-Archiv waren die Ergebnisse der Bounding Varianten ebenfalls besser als die der “normalen” Variante. Bei der Variante, in der beide Archive gemeinsam verwendet wurden, haben die Bounding-Varianten ebenfalls eine Verbesserung erzielt. Beim Vergleich zwischen den Archiven kann man sagen, dass die Pop-Variante bessere Ergebnisse liefert als die Gosh-Variante. Die Variante, in der beide Archive gleichzeitig verwendet werden, ist wiederum besser als die anderen beiden Varianten.

Die Tests haben ebenfalls ergeben, dass der Speicherverbrauch der Archive durch die Verwendung der Bounds erhöht wird. Das ist darauf zurück zu führen, dass wenn beim Einfügen einer Lösung eine Bound gefunden wird, die schlechter ist als die beste bisher gefundene Lösung, diese Lösung teilweise eingefügt wird und zusätzlich noch eine neue konvertierte Lösung.

Die Ergebnisse dieser Arbeit wurden auch mit den Ergebnissen anderer Arbeiten verglichen, um zu sehen wie das Verfahren im Vergleich zu anderen abschneidet. Dabei hat sich gezeigt dass die Bounding-Strategie mit anderen Verfahren mithalten kann und in einigen Instanzen bessere Ergebnisse liefert.

Die Ergebnisse dieser Arbeit haben gezeigt, dass die Bounding-Strategie eine Verbesserung gegenüber dem Lösungsarchiv ohne Bounding-Strategie bringt. Ein wichtiger Punkt bei der Bounding-Strategie ist es, eine effiziente Methode für die Boundberechung zu finden, damit die Anzahl der erzeugten Generationen nicht zu stark abnimmt. Für andere Problemstellungen könnte dieses Verfahren auch Verbesserungen bringen, wenn eine effiziente Methode zur Boundberechnung gefunden wird.

Literaturverzeichnis

[1] FEREMANS, C.: *Generalized Spanning Trees and Extensions*, Universite Libre de Bruxelles, Diss., 2001

[2] GHOSH, D.: Solving medium to large sized Euclidean generalized minimum spanning tree problems / Indian Institute of Management, Research and Publication Department. 2003. – Forschungsbericht

[3] HU, B. ; LEITNER, M. ; RAIDL, G. R.: Computing Generalized Minimum Spanning Trees with Variable Neighborhood Search. In: HANSEN, P. (Hrsg.) ; MLADENOVIĆ, N. (Hrsg.) ; PÉREZ, J. A. M. (Hrsg.) ; BATISTA, B. M. (Hrsg.) ; MORENO-VEGA, J. M. (Hrsg.): *Proceedings of the 18th Mini Euro Conference on Variable Neighborhood Search*. Teneriffa, Spanien, 2005

[4] HU, B. ; LEITNER, M. ; RAIDL, G. R.: Combining Variable Neighborhood Search with Integer Linear Programming for the Generalized Minimum Spanning Tree Problem. In: *Journal of Heuristics* 14 (2008), Nr. 5, S. 473–499

[5] HU, B. ; RAIDL, G. R.: An Evolutionary Algorithm with Solution Archive for the Generalized Minimum Spanning Tree Problem. In: QUESADA-ARENCIBIA, A. (Hrsg.) u. a.: *Proceedings of EUROCAST 2011 – 13th International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, February 6–11, 2011*, 2011, S. 256–259

[6] JIANG, H. ; CHEN, Y.: An efficient algorithm for generalized minimum spanning tree problem. In: *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*. New York, NY, USA : ACM, 2010. – ISBN 978–1–4503–0072–8, S. 217–224

[7] LEITNER, M.: *Solving Two Generalized Network Design Problems with Exact and Heuristic Methods*, Technische Universität Wien, Diplomarbeit, 2006. – supervised by G. Raidl and B. Hu

[8] MYUNG, Y. S. ; LEE, C. H. ; TCHA, D. W.: On the Generalized Minimum Spanning Tree Problem. In: *Networks* 26 (1995), S. 231–241

[9] NISSEN, V.: *Einführung in Evolutionäre Algorithmen.: Optimierung nach dem Vorbild der Evolution.* Vieweg, 1997

[10] POP, P. C. ; IORDACHE, S.: A hybrid heuristic approach for solving the generalized traveling salesman problem. In: KRASNOGOR, Natalio (Hrsg.) ; LANZI, Pier L. (Hrsg.): *GECCO*, ACM, 2011, S. 481–488

[11] POP, P.C.: *Generalized Minimum Spanning Tree Problem*, University of Twente, Diss., 2002

[12] RAIDL, G. R. ; HU, B.: Enhancing Genetic Algorithms by a Trie-Based Complete Solution Archive. In: *Evolutionary Computation in Combinatorial Optimisation – EvoCOP 2010* Bd. 6022, Springer, 2010 (LNCS), S. 239–251

[13] SONNLEITNER, M.: *Ein neues Lösungsarchiv für das Generalized Minimum Spanning Tree-Problem*, Technische Universität Wien, Diplomarbeit, 2010

[14] ŠRAMKO, A.: *Enhancing a Genetic Algorithm by a Complete Solution Archive Based on a Trie Data Structure*, Technische Universität Wien, Diplomarbeit, 2009

[15] WOLF, M.: *Ein Lösungsarchiv-unterstützter evolutionärer Algorithmus für das Generalized Minimum Spanning Tree-Problem*, Diplomarbeit, 2009

[16] ZAUBZER, S.: *A Complete Archive Genetic Algorithm for the Multidimensional Knapsack Problem*, Technische Universität Wien, Diplomarbeit, 2008