FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Ein neues Lösungsarchiv für das Generalized Minimum Spanning Tree-Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Mika Sonnleitner
Matrikelnummer 0225096

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, TT.MM.JJJJ

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung

Mika Sonnleitner
Hütteldorfer Straße 150-158/21/5
1140 Wien

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

Wien, ?.9.2010

Kurzfassung

Diese Arbeit behandelt das Generalized Minimum Spanning Tree-Problem (GMST), ein kombinatorisches Optimierungsproblem, das auf dem Mininum Spanning Tree (MST)-Problem basiert.

Das GMST ist eine Verallgemeinerung des klassischen MST-Problems, das darin besteht, für einen gegebenen Graphen einen minimalen Spannbaum zu finden. Die Verallgemeinerung besteht darin, dass die Knoten in Cluster partitioniert sind. Für eine konkrete Lösung wird ein Spannbaum gebildet, der aus jedem Cluster genau einen Knoten verwendet. Während das MST in polynomieller Zeit optimal lösbar ist, ist das GMST NP-schwierig.

In dieser Arbeit wird ein evolutionärer Algorithmus (EA) verwendet, der ein triebasiertes Lösungsarchiv, basierend auf zwei verschiedenen Betrachtungsweisen des GMST, verwendet. Die erste Sichtweise besteht darin, die Knoten in den jeweiligen Clustern festzulegen. Da das verbleibende Problem dem MST entspricht und somit in polynomieller Zeit gelöst werden kann, kann eine Lösung durch die Angabe der Knoten spezifiziert werden. Die zweite Vorgehensweise ist dann, die globalen Verbindungen zwischen den Clustern festzulegen. Auch hier kann das verbleibende Problem in polynomieller Zeit gelöst werden, und zwar mittels dynamischer Programmierung.

In einem Lösungsarchiv können Lösungen gespeichert werden, um einerseits Duplikate zu erkennen, um sie nicht ein weiteres Mal bearbeiten zu müssen, und andererseits eine neue, noch nicht durchsuchte Lösung zu bekommen. Da ein Lösungsarchiv für die erste Sichtweise bereits implementiert wurde, wurde in dieser Arbeit eines für die zweite entworfen. Dieses Lösungsarchiv basiert auf der Sichtweise, die globalen Verbindungen festzulegen. Es werden zwei verschiedene Darstellungsarten von Spannbäumen verwendet, die diese globalen Kanten repräsentieren, nämlich die Darstellung der Predecessor sowie Prüfernummern. Außerdem wurden die beiden Archive kombiniert, um bessere Ergebnisse als mit einem alleine zu erhalten.

Wie die Tests gezeigt haben, konnten mit dem neuen Archiv bessere Lösungen im Vergleich zum EA gefunden werden. Mit der Verwendung beider Archive noch bessere Lösungen gefunden werden, wobei diese Version einen größeren Speicherverbrauch aufweist.

Abstract

This thesis deals with the Generalized Minimum Spanning Tree Problem (GMST), a combinatorial optimization problem based on the Minimum Spanning Tree (MST) Problem.

The GMST is a generalization of the classic MST Problem which consists in finding a minimum spanning tree for a given graph. The generalization consists in partionizing the nodes in clusters. In order to obtain a concrete solution, a spanning tree is formed, which spans exactly one node from each cluster. While the MST is solvable in polynomial time, the GMST is NP-hard.

In this thesis, an evolutionary algorithm (EA) is used which is complemented by a trie-based solution archive using two different views of the GMST. The first view is concerned with selecting the node for each cluster. As the remaining problem equals to the MST, the solution can be encoded by specifying the nodes. The second view is concerned with specifying the global edges between the clusters. The remaining problem can be solved using dynamic programming.

Using a solution archive, it is possible to store solutions generated by the EA in order to detect duplicates and furthermore to convert such duplicates into new solutions which have not yet been examined. As a solution archive for the first view mentioned has already been implemented, this thesis is concerned with designing an archive for the second view. This solution archive is based on the view to specify the global edges. Two different encodings of spanning trees are used which represent these global edges, namely the Predecessor-encoding and the Prüfer-encoding. Furthermore both archives have been combined to improve the obtained solutions.

As tests have shown, using the archive improves the quality of the solutions compared to the pure EA. Using both archives combined, even better results can be obtained at the expense of a higher memory usage.

Inhaltsverzeichnis

KAPITEL 1

Einführung

1.1 Das Minimum Spanning Tree-Problem

Das Minimum Spanning Tree-Problem (MST) ist ein bekanntes Problem in der Graphentheorie. Es ist wie folgt definiert [Lei06]: Gegeben sei ein gewichteter, vollständiger Graph $G = (V, E, c)$ mit Knotenmenge V, Kantenmenge E and der Kostenfunktion $c : E \rightarrow \mathbb{R}^+$. Dann ist ein Minimum Spanning Tree ein Subgraph von G , also ein Baum, der alle Knoten des Graphen beinhaltet und dessen Kosten minimal sind: $S = (V, T), C(T) = \sum_{c(u,v)\in T} c(u, v)$ Dabei sind die Kosten als Summe der Gewichte der Kanten definiert. Das MST ist in polynomieller Zeit lösbar, zum Beispiel mit dem Algorithmus von Kruskal [OW02].

1.2 Das Generalized Minimum Spanning Tree-Problem

Als Verallgemeinerung des MST wurde das Generalized Minimum Spanning Tree-Problem (GMST) definiert. Dabei werden die Knoten V aus dem MST in Cluster partitioniert. Für eine Lösung des GMST wird aus jedem Cluster ein konkreter Knoten zu einem Spannbaum verbunden. Somit kann das GMST wie folgt definiert werden [Lei06]: Gegeben sei ein gewichteter, vollständiger Graph $G = (V, E, c)$. Dieser Graph ist dann in n Cluster unterteilt, die jeweils disjunkte Teilmengen der Knoten des Graphens sind: $V_0, V_1, ..., V_n$ mit $V_i \subset V$, sodass $V_i \cap V_j = \emptyset \; \forall i, j, i \neq j$. Eine gültige Lösung ist dann ein Subgraph von G, $S = (P, T)$, mit $P = \{p_1, p_2, ..., p_n\} \subset V \; p_i \in V_i \forall i = 1, ..., r$, sodass aus jedem Cluster V genau ein Knoten verbunden wird. Der Spannbaum besteht dann aus der Kantenmenge $T \subseteq P \times P \subset E$ Die Bewertung bzw. die Kosten ergibt sich dann aus der Summe der Werte der Kanten, also $C(T) = \sum_{c(u,v)\in T} c(u, v)$ Die optimale Lösung für dieses Problem ist dann ein solcher Spannbaum $S = (P, T)$, dessen Kosten $C(T)$ minimal sind. Eine Illustration des MST und des GMST ist in Abbildung 1.1 zu finden.

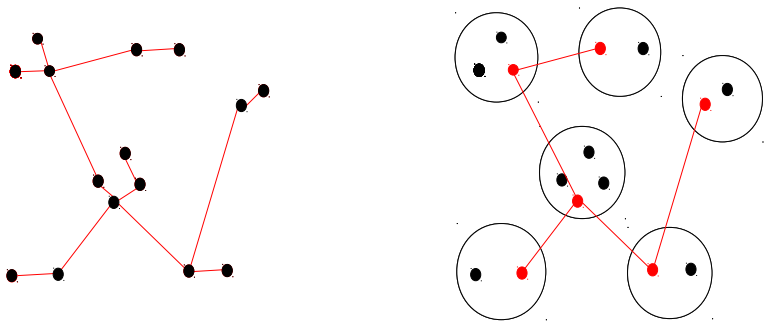

Abbildung 1.1: MST und GMST

1.3 Bisherige Arbeit

Das GMST-Problem wurde 1995 von Myung, Lee, and Tcha [MLT95] definiert. Dabei haben sie auch gezeigt, dass es NP-schwierig ist und eine mathematische Definition angegeben. Pop [Pop05] hat einerseits Spezialfälle angegeben, unter denen das Problem polynomiell lösbar ist, und andererseits einige Beispiele angeführt, für die das Problem in der Praxis verwendet werden kann. Als Beispiel führt er die Verbindung von lokalen Netzwerken an, bei denen eine Station als Hub verwendet wird. Außerdem hat er des GMST in zwei Teilprobleme aufgeteilt. Er hat den Ansatz gewählt, die globalen Kanten festzulegen. Außerdem hat er seinen Ansatz dann in einem Algorithmus mit Simulated Annealing verwendet. Ghosh [Gho03] hat das Verfahren entwickelt, für jeden Cluster einen Knoten festzulegen, der dann die Knotenmenge für den Spannbaum bildet. Nach dieser Belegung kann für beide Verfahren die optimale Lösung für das verbleibende Problem eindeutig bestimmt werden. Daher können Lösungen für das Problem mit der Angabe der Knoten bzw. der globalen Kanten kodiert werden. Da diese Arbeit auf diesen Verfahren aufbaut, werden sie im Abschnitt 2.1 genauer dargestellt. Hu, Leitner und Raidl [HLR08] haben den Ansatz der Variable Neighborhood Search mit Nachbarschaften, die auf diesen beiden Darstellungen aufbauen, verwendet und sind zu guten Resultaten gekommen.

Wolf [Wol09] hat in seiner Arbeit einen EA zur Lösung des GMST verwendet, den er um ein Lösungsarchiv erweitert hat. Dafür hat er den Trie als geeignete Datenstruktur für das Speichern und Generieren von neuen Lösungen eingesetzt. Nachdem die Lösungen in der Kodierung nach den Knoten gespeichert werden, wurde auch der Trie mit diesen Lösungen befüllt. Grundsätzlich ist er so auch zu guten Ergebnissen gekommen, allerdings wurde in dem Algorithmus auch ein Verfahren zur Optimierung nach dem Pop-Schema inkludiert. Gemäß seinen Tests ergänzen sich der Trie und diese Optimierung nicht gut, sondern der Trie erzielt mit dieser Optimierung teilweise sogar schlechtere Lösungen. Daher schlug er als mögliche weiterführende Arbeit vor, auch ein Archiv auch für Lösungen, die nach den globalen Kanten kodiert sind, zu entwickeln, und diese dann nach Möglichkeit zu kombinieren, was das Thema der vorliegenden Arbeit ist.

1.4 Heuristische Optimierungsverfahren

Da es für viele schwierige Probleme der Informatik nicht möglich ist, einen exakten Algorithmus zu finden, der das Problem für größere Instanzen in akzeptabler Zeit löst, sind verschiedenste heuristische Verfahren vorgeschlagen worden, um Lösungen von akzeptabler Güte zu finden. Ein grundlegendes Verfahren ist das der lokalen Suche, d.h. es wird versucht, für ein Problem bzw. eine Lösung eine Nachbarschaft zu definieren, diese zu durchsuchen und die Lösung mit dem besten Wert zu übernehmen. Eine solche Nachbarschaft besteht aus aus Lösungen, die mit einer definierten (geringfügigen) Änderung an der Ursprungslösung zu erreichen sind. Diese müssen an das jeweilige Optimierungsproblem angepasst werden. Da dieses Verfahren zu einem lokalen Optimum führt, aber in der Regel nicht zu einem globalen, sind diverse Metaheuristiken erdacht wurden, mit denen diese überwunden werden sollen. Eine Metaheuristik ist ein allgemeines, vom konkreten Optimierungsproblem unabhängiges Verfahren, das definiert, welche Vorgehensweise grundsätzlich verwendet wird. Es werden abstrakte Operatoren definiert, die dann an das konkrete Problem angepasst werden müssen. Beispiele dafür sind Simulated Annealing, die Tabusuche oder eben der in der vorliegenden Arbeit verwendeten Evolutionäre Algorithmus. Oftmals ist es nicht möglich, Gütegarantien abzugeben, wie gut die Lösung approximiert werden kann. Dennoch wurden in der Praxis gute Resultate erzielt.

1.5 Evolutionäre Algorithmen

Grundsätzlich wird versucht, die natürliche Evolution bzw. die Erkenntnisse der Evolutionstheorie mit ihren Mechanismen zu modellieren. Es werden jedoch meist nur die grundlegenden Konzepte wie das Vorhandensein einer Population von Lösungen sowie Selektion, Mutation und Rekombination verwendet.

In einem Evolutionären Algorithmus (EA) bzw. einem Genetischen Algorithmus [SP94] gibt es, wie bei einigen anderen heuristischen Optimierungsverfahren, eine Population von Lösungen für das Problem. Die kodierte Lösung wird als Genotyp bezeichnet, einzelne Teile als Gene. Eine solche Lösung kann etwa als Bitstring kodiert werden. Da die Bewertung des Genotyps oft nicht direkt aus der Darstellung berechnet werden kann, ist eine Dekodierung erforderlich. Diese dekodierte Lösung wird dann als Phänotyp bezeichnet. Dann wird eine Fitnessfunktion definiert, die den Lösungen einen Wert zuordnet. Bei einem Minimierungsproblem wie dem GMST-Problem haben somit Individuen mit einem kleineren Funktionswert eine höhere Fitness. Mittels der Mutation wird analog zum biologischen Vorbild neues genetisches Material erzeugt. Dieses soll der Überwindung von lokalen Optima dienen. Bei der Mutation wird also eine Lösung zufällig verändert. Die Mutation wird in der Regel nur mit einer gewissen (kleinen) Wahrscheinlichkeit aufgerufen, da der Algorithmus andernfalls zu einer Zufallssuche werden würde. Die Rekombination dient dazu, aus zwei Elternlösungen Kindslösungen zu generieren, die möglichst viel genetisches Material übernehmen sollen. Es wird somit versucht, zwei Lösungen zu einer neuen zu verschmelzen. Die Selektion dient dazu, aus der Population Lösungen auszuwählen, die überleben, indem sie in die nächste Generation des Algo-

rithmus übernommen werden. Sie setzt dabei an dem Funktionswert der Lösungen an; Lösungen mit einem besseren Wert haben eine höhere Chance, selektiert zu werden. Da die prinzipielle Funktionsweise des evolutionären Algorithmus sehr allgemein sind, müssen die Operatoren an das spezifische Problem angepasst werden, um gute Ergebnisse zu liefern. Dabei ist es oft schwierig, geeignete Rekombinationsoperatoren zu finden, die einerseits beide Elternlösungen berücksichtigen und andererseits die Qualität der Lösungen bewahren können. In eingeschränktem Sinn gilt das auch für Mutationoperatoren.

In einem sogenannten Steady-State-EA, wie er in dieser Arbeit verwendet wird, wird pro Generation nur eine Lösung durch eine neue ersetzt. Dagegen wird in einem sogenannten generationellen EA in jeder Generation die Ausgangspopulation ersetzt, was eher dem biologischem Vorbild entspricht.

1.6 Lösungsarchiv

Ein Lösungsarchiv wird dazu verwendet, Lösungen, die von einem heuristischen Algorithmus generiert werden, zu speichern, um deren Bewertung nicht mehr als einmal berechnen zu müssen. Da dessen Berechnung aus dem Genotypen oft recht aufwendig ist, kann sich der zusätzliche Aufwand für das Speichern und Suchen der Lösungen rentieren. Ein weiteres Ziel, das auch im Rahmen dieser Arbeit verfolgt wird ist, die Lösungen nicht nur zu speichern, sondern auch durch das Modifizieren eines Duplikats eine neue Lösung zu bekommen, die noch nicht untersucht wurde. Dieses Verfahren wurde bereits von Zaubzer [Zau08], Šramko [Sra09] und Wolf [Wol09] für verschiedene Optimierungsprobleme verwendet. Grundsätzlich sind dafür verschiedene Datenstrukturen, wie eine Hashtabelle, Bäume oder Tries denkbar. Zaubzer [Zau08] und Šramko haben diese Datenstrukturen analysiert und haben den Trie für die am besten geeignete befunden. Eine Hashtabelle kann zwar Lösungen speichern und es kann in $O(l)$, (wobei l die Länge des Strings ist) festgestellt werden, ob eine Lösung bereits enthalten ist, jedoch ist es nicht möglich, schnell eine Ersatzlösung zu finden, da dies im schlechtesten Fall 2^l Schritte brauchen kann. Ein binärer Baum könnte die relevanten Operationen zwar in $O(l * log(n))$ (wobei n die Anzahl der Knoten im Baum ist) durchführen, jedoch muss jede Lösung extra in einem eigenen Knoten gespeichert werden, was einen hohen Speicherverbrauch verursacht. Daher ist es eine bessere Lösung, einen Trie zu verwenden, der die Operationen in $O(l)$ durchführen kann. Da diese Analyse für binäre Tries durchgeführt wurde, und wie in der Arbeit von Wolf [Wol09] in der vorliegenden Arbeit ein anders strukturierter Trie verwendet wird, dessen Speicherverbrauch deutlich höher ist, muss dies gesondert betrachtet werden. Inwieweit dies ein Problem darstellt, wird in der Beschreibung des Tries und den Tests analysiert.

1.7 Tries

Ein Trie ist eine Datenstruktur, in der Wörter gespeichert und gesucht werden können. Somit stellen sie eine Lösung für das Wörterbuchproblem dar. Für das vorliegende Problem ist es jedoch nicht erforderlich, Wörter wieder zu entfernen, da das Lösungsarchiv

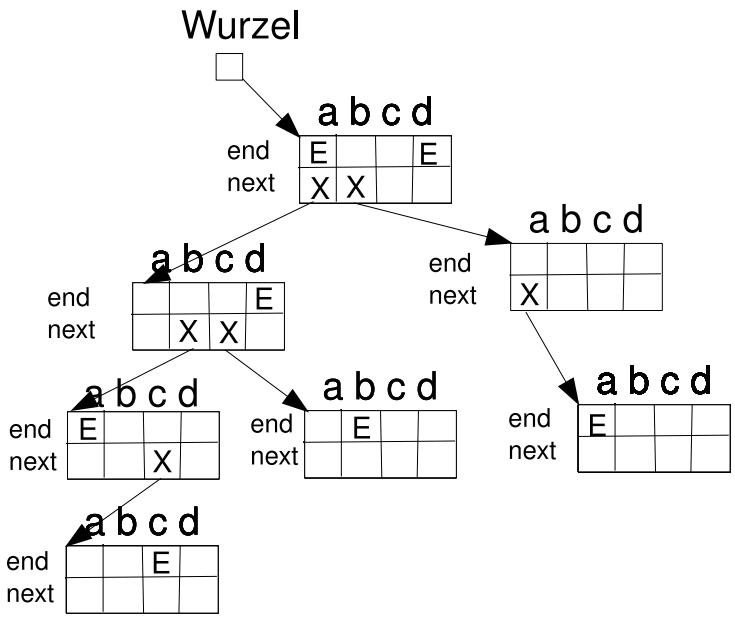

Abbildung 1.2: Ein Indexed Trie mit den Wörtern {a, aba, abcc, acb, ad , baa, d }

diese Funktionalität nicht benötigt. Im Vergleich etwa zu einem binären Baum werden die Lösungen im Trie nicht in den Knoten gespeichert, sondern in den Kanten. Dabei wird jeder Kante ein Buchstabe zugeordnet, während die Knoten grundsätzlich keine Lösungen speichern. Die Lösung wird Konkatenation der jeweiligen Buchstaben der Kanten auf dem Weg von der Wurzel zur Lösung berechnet. Es ist allerdings erforderlich, das Ende der Zeichenfolgen in den Knoten zu markieren.

Im Indexed Trie können grundsätzlich beliebig lange Wörter über einem gegebenen Alphabet A definiert werden. Für jeden Knoten in dem Trie gilt, dass sie denselben Präfix teilen. Ein Knoten enthält die Information, ob bzw. welche Wörter mit welchem Buchstaben an diesem Knoten enden, d.h. $|A|$ Boolsche Variablen. Dieser ist in Abbildung 1.2 als end bezeichnet, konkrete Wörter sind mit E markiert. Außerdem gibt es die Referenzen auf potenziell $|A|$ weitere Knoten, wobei der Knoten dem Präfix einen weiteren Buchstaben hinzufügt. In der Abbildung wird dieses als next bezeichnet, wobei Buchstaben, die zu weiteren Knoten führen mit X markiert sind.

KAPITEL 2

Ablauf des Algorithmus

2.1 Zwei Lösungskodierungen für das GMST-Problem

Im Zusammenhang mit dem GMST-Problem und der vorliegenden Arbeit sind die zwei Vorgehensweisen von grundlegender Bedeutung. Sie gehen auf die Autoren Pop [Pop02] und Ghosh [Gho03] zurück, die diese verwendet haben.

Die Lösungskodierung nach Ghosh

Ghosh hat die Vorgangsweise definiert, die Knoten in den Cluster festzulegen. Für die Lösung S, den Subgraphen von G, $S = (P, T)$ werden also die Knoten $P = \{p_1, p_2, ..., p_n\}$ festgelegt. Nach diesem Schritt ist die Lösung für das Problem, aus diesen Knoten die optimale Lösung zu finden, also die Kantenmenge T zu bestimmen, gleich dem klassischen MST, das leicht gelöst werden kann. Im Kontext des EA wird somit der Genotyp definiert, d.h. die Lösungen werden auf diese Weise codiert. Dies kann dann etwa in einem Array erfolgen, das für jeden Cluster den auswählten Knoten speichert. Nachdem die optimale Lösung für diese Knotenbelegung einfach berechnet werden kann, reicht dies für die eindeutige Beschreibung der Lösung aus. Dieser Ansatz wird in Abbildung 2.1 illustriert.

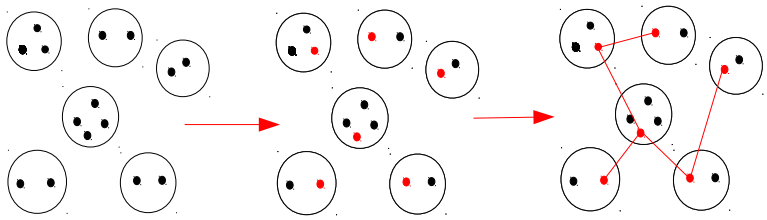

Abbildung 2.1: Der Ansatz nach Pop. Mittels der gegebenen Knoten kann die optimale Auswahl der Kanten berechnet werden.

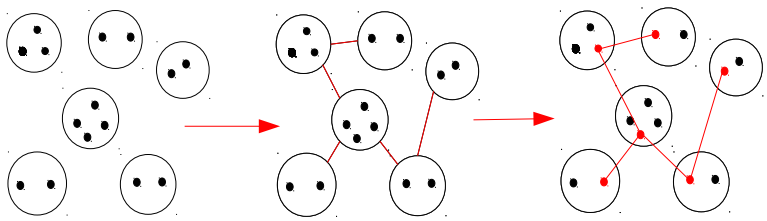

Abbildung 2.2: Der Ansatz nach Pop. Mittels der gegebenen globalen Kanten kann die optimale Auswahl der Knoten berechnet werden.

Die Lösungskodierung nach Pop

Im Gegensatz dazu hat Pop die umgekehrte Vorgehensweise definiert, nämlich die Kanten zwischen den Clustern festzulegen, und nach dieser Festlegung die optimale Knotenbelegung innerhalb der Cluster zu berechnen. Genauer ausgedrückt, wird ein sogenannter globaler Graph definiert [Lei06], dessen Knoten die Cluster des ursprünglichen Problems sind: $G^g = (V^g, E^g)$ und $V^g = \{V_1, V_2, ..., V_n\}$. Die Kantenmenge entspricht den Verbindungen zwischen den Knoten: $E^g = V^g \times V^g$ Diese Kanten werden auch als globale Kanten bezeichnet. Der Graph ist vollständig, d.h. es gibt zwischen jeweils zwei Knoten immer eine Kante. Auf diesem globalen Graphen wird dann der globale Spannbaum definiert: $S^g = (V^g, T^g)$ mit $T^g \subseteq E^g$. Dieser Spannbaum repräsentiert alle Lösungen für das ursprüngliche GMST-Problem, für die gilt, dass sie für jede Kante $(V_a, V_b) \in T^g$ eine Kante $(u, v) \in E$ und $u \in V_a \wedge v \in V_b \wedge a \neq b$. Wenn die globalen Kanten festgelegt ist, ist die optimale Knotenbelegung des ursprünglichen Graphen P für diesen eindeutig und kann mittels dynamischer Programmierung effizient berechnet werden. In der Terminologie des EA wird somit der Phänotyp berechnet. Analog zur ersten Darstellung können Lösungen mittels dieses globalen Spannbaums T^g kodiert werden, da die optimale Lösung bereits eindeutig berechenbar ist. Diese Kodierung wird in Abbildung 2.2 illustriert.

Zur einfacheren Benennung wird das Verfahren, die Knoten in den Cluster festzulegen bzw. die dazugehörige Codierung in weiterer Folge Ghosh-Verfahren bzw. Ghosh-Kodierung oder Ghosh-Darstellung genannt, das Verfahren, die Kanten zwischen den Clustern festzulegen, hingegen Pop-Verfahren bzw. Pop-Kodierung oder Pop-Darstellung. Das Lösungsarchiv wird dann ebenfalls Ghosh-Archiv bzw. Pop-Archiv genannt.

2.2 Aufbau des EAs

Es wird ein steady-state-EA verwendet, der um die Funktionalität des Lösungsarchivs ergänzt wird. Dessen grundsätzlicher Ablauf ist in Algorithmus 1 zu finden. Die Aufgabe der vorliegenden Arbeit ist es, ein Lösungsarchiv auf Basis der Pop-Darstellung zu implementieren. Darüber hinaus wurde das auf der Ghosh-Darstellung basierende Archiv von Wolf [Wol09] übernommen, so dass der Algorithmus sowohl nur mit dem Pop-Archiv, als auch mit beiden laufen kann. Er hat auch eine Variante mit mehreren Tries implementiert. In dieser Arbeit wird aber ausschließlich die Variante mit einem Trie verwendet.

Algorithmus 1 steady-state EA mit Lösungsarchiv

1: Generiere zufällige Population *pop* und füge sie in Archiv ein
2: **while** Solange Abbruchbedingung nicht erfüllt **do**
3: $elternteil1$ <- selektion(*pop*)
4: $elternteil2$ <- selektion(*pop*)
5: sol_{neu} <- rekombination($elternteil1$, $elternteil2$)
6: mutation(sol_{neu})
7: lokaleVerbesserung(sol_{neu})
8:
9: **if** sol_{neu} in Archiv enthalten **then**
10: wandle sol_{neu} zu neuer Lösung um
11: **end if**
12: Füge sol_{neu} in Archiv ein
13: Ersetze eine Lösung in *pop* durch sol_{neu}
14: **end while**

2.3 Repräsentierung von Lösungen

Grundsätzlich arbeitet der Algorithmus auf Basis der Ghosh-Kodierung, d.h. die Lösungen werden in dieser Kodierung gespeichert. Die Lösung wird, wie beschrieben, durch die Knotenmenge $P = \{p_1, p_2, ..., p_n\}$ definiert. Da es keine ungültigen Lösungen gibt, kann diese Menge einfach mit einem Array bzw. Vektor kodiert werden. Die Operatoren auf Basis der Ghosh-Darstellung basieren somit auf dieser Kodierung. Der zweite Ansatz basiert auf der Pop-Darstellung: Es werden somit die globalen Kanten T^g zwischen den Clustern kodiert. Hierfür wird die sogenannte Predecessor-Darstellung verwendet. Konkret wird dafür zunächst ein Wurzelcluster festgelegt. Da es sich um einen Spannbaum handelt, gibt es für jeden Cluster genau einen Weg zum Wurzelknoten. Der Vorgänger eines Clusters ist dann jener, mit dem er auf diesem Weg direkt verbunden ist. Diese Lösungsrepräsentierung kann mittels eines Vektors erfolgen, der Vorgänger des n-ten Clusters wird in dem Vektor auf Position n gespeichert. Dies wird in Abbildung 2.3 illustriert. Ein Nachteil dieser Kodierung ist, dass nicht jeder mögliche Kodierung einen gültigen Spannbaum beschreibt. Daher ist es bei den Funktionen, die eine Lösung auf Basis dieser Codierung veränderen, notwendig zu beachten, dass sie nur Spannbäume generieren. Da der EA intern die Ghosh-Darstellung verwendet, ist es notwendig, vor der Verwendung der Pop-Operatoren bzw. des entsprechenden Archivs die Lösung in die Predecessor-Darstellung zu transformieren. Danach muss die Lösung wieder zurücktransformiert werden. Es wird außerdem die sogenannte Prüfer-Codierung verwendet, die aber nicht für den EA oder die Optimierung selber verwendet wird, sondern ausschließlich für die Speicherung der Lösungen im Trie. Auch hier müssen die Lösungen von der Predecessor-Darstellung in die Prüfer-Darstellung transformiert werden. Diese Algorithmen werden im Abschnitt 2.10 dargestellt.

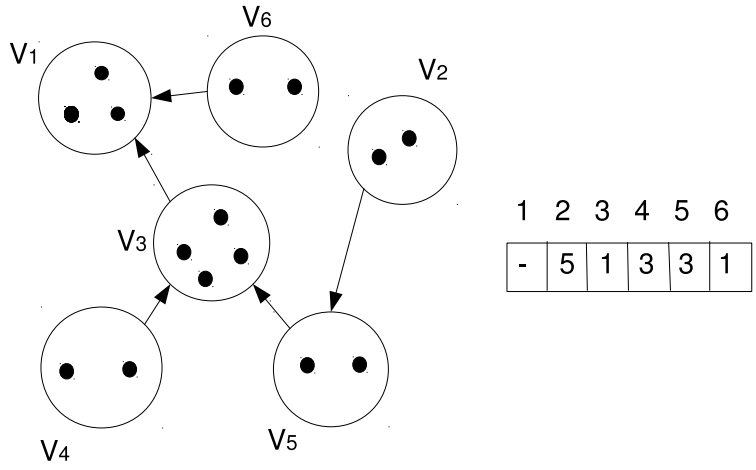

Abbildung 2.3: Die Predecessor-Kodierung. V_1 ist der Wurzelcluster

2.4 Die Operationen des Evolutionären Algorithmus

Damit beide Darstellungen in allen Bereichen des Algorithmus verwendet werden, werden für beide Darstellungen - die Repräsentation der Knoten der Cluster als auch über globale Kanten - eigene Rekombinations- und Mutationsoperatoren verwendet. Grundsätzlich wird bei jedem Aufruf zufällig eine der beiden Varianten ausgewählt. Es kann aber über einen Parameter gesteuert werden, ob welche Operatoren verwendet werden sollen - entweder nur die Ghosh-Operatoren, nur die Pop-Operatoren oder bei jedem Aufruf zufällig eine der Varianten.

Für die Selektion wird Tournament Selection verwendet, d.h. es werden zufällig eine bestimmte Anzahl von Lösungen ausgewählt, und die beste aus diesen wird dann tatsächlich ausgewählt. In dieser Arbeit wird die Standardeinstellung von EALIB (siehe 2.13) verwendet: Es werden zwei Lösungen ausgewählt, und die bessere davon selektiert.

Die Operationen für die Ghosh-Darstellung

Da es bei dieser Darstellung keine Abhängigkeiten gibt, können einfache Operatoren für Arrays verwendet werden.
Die Rekombination erfolgt somit mit Uniform Crossover, d.h. dass für jedes Feld des Arrays zufällig der Wert einer der beiden Elternlösungen übernommen wird. Gegeben sind also zwei Lösungen in der Ghosh-Darstellung: $P = \{p_1, p_2, ..., p_n\}$ $Q = \{q_1, q_2, ..., q_n\}$. Dann wird die neue Lösung R wie folgt generiert: $\forall i : r_i$ wird zufällig, also mit jeweils Wahrscheinlichkeit $p = 0,5$ aus der Menge $\{p_i, q_i\}$ ausgewählt. Die Rekombination wird in 2.4 illustriert.

Bei der Mutation wird hingegen ein zufällig gewähltes Feld des Arrays verändert. In der Lösung $P = \{p_1, p_2, ..., p_n\}$ wird somit ein Element zufällig, also jeweils mit

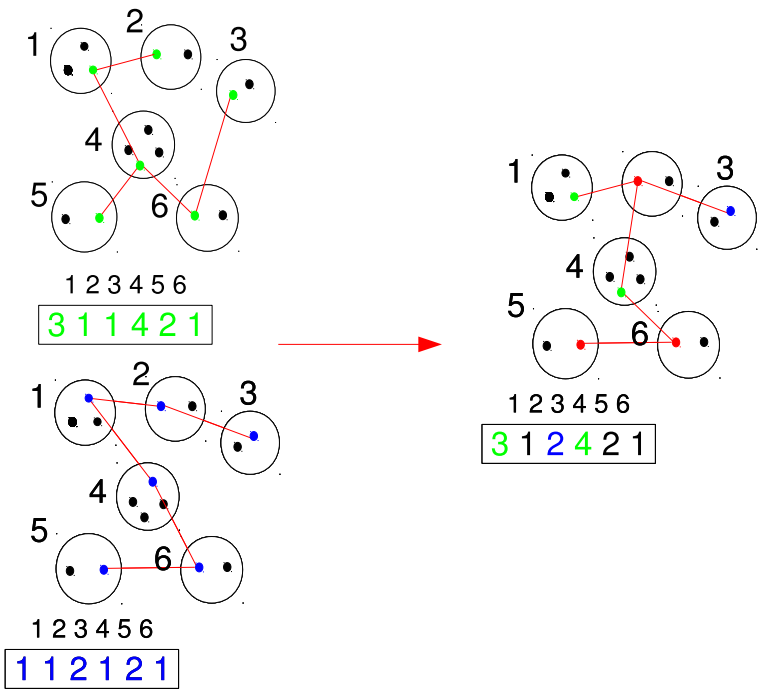

Abbildung 2.4: Die Rekombination auf Basis der Ghosh-Kodierung

Wahrscheinlichkeit $p = \frac{1}{n}$ p_i ausgewählt. Dieser Knoten ist Element eines Clusters $p_i \in V_i$. Dieser Knoten wird dann zufällig ($p = \frac{1}{|V_i^*|}$) durch ein anderes Element des Clusters ersetzt, d.h. aus der Menge $V_i^* = V_i \backslash \{p_i\}$.

Die Operationen für die Predecessor-Darstellung

Da diese Codierung die Lösungen als Spannbäume darstellt, und diese in der Predecessor-Darstellung vorliegen, muss darauf geachtet werden, dass keine ungültigen Lösungen erzeugt werden. Die Operatoren wurden aus der Arbeit von Raidl und Drexel [RD00] übernommen, in der sie die Operatoren für ein ähnliches Problem, das Capacitated Minimum Spanning Tree Problem, definieren. Diese Operatoren können analog für das GMST verwendet werden, es muss nur die zusätzliche Überprüfung, ob die Kapazitäten nicht überschritten werden, entfernt werden.

Die Rekombination wird wie folgt durchgeführt: Ausgehend von den beiden Elternlösungen, werden die Kanten, die bei beiden identisch ist, also bei denen der Predecessor für den jeweiligen Cluster derselbe ist, auf jeden Fall in die Kindslösung übernommen. Sind sie nicht identisch, so wird zufällig der Wert einer der Elternlösungen übernommen, wenn sich dabei kein Zyklus ergibt. Die Einhaltung dieser Bedingung wird mit einer Union-Find-Datenstruktur überprüft. Würde ein Zyklus entstehen, so wird die andere

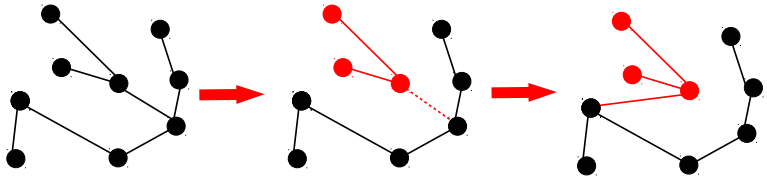

Abbildung 2.5: Die Mutation auf Basis der Pop-Kodierung

Lösung verwendet. Ist auch dies nicht möglich, so wird der Cluster in eine Liste von noch nicht bearbeiteten Clustern gespeichert. Die Cluster, die übrig geblieben sind, werden dann eine zufällige Stelle im Baum gehängt.

Die Mutation wird hingegen wie folgt ausgeführt: Es wird zufällig ein Cluster im Spannbaum ausgewählt, dessen Vorgänger verändert werden soll. Kandidaten für den neuen Vorgänger sind dann alle Cluster, dessen direkter oder indirekter Vorgänger nicht der Cluster ist, dessen Vorgänger verändert werden soll, da dadurch ein Zyklus entstehen würde. Die Mutation wird in Abbildung 2.5 illustriert.

2.5 Die Pop-Optimierung

Die Pop-Optimierung [Wol09] besteht darin, für eine gegebene Lösung in der Ghosh-Darstellung die Kanten zwischen den Clustern zu verwenden und für diese Kanten dann mittels dynamischer Programmierung die optimalen Knoten zu berechnen. Bei der Verwendung eines Pop-Tries wird diese auch implizit verwendet, da die Lösungen ohnehin zwischen der Ghosh-Darstellung und der Predecessor-Kodierung umgewandelt werden müssen. Es ist also nicht erforderlich, diese dann explizit aufzurufen. Die Optimierung ist also nur dann eine sinnvolle Option, wenn kein Pop-Archiv verwendet wird.

2.6 Lokale Verbesserung

Auch bei der lokalen Verbesserung werden wieder verschiedene Nachbarschaftsstrukturen benutzt, die auf den beiden Darstellungsarten basieren. Die Definition wurde aus der Arbeit von Leitner übernommen [Lei06]. Eine Nachbarschaftsstruktur ist eine Funktion, die jeder Lösung eine Menge von Lösungen zuweist, die die Nachbarschaft bilden. Formal also eine Funktion $N : X \rightarrow 2^X$, der jeder Lösung $x \in X$ eine Menge von Nachbarn $N(X) \subseteq X$ zuweist. Analog zur Ghosh-Darstellung wird eine Ghosh-Nachbarschaft bzw. Node Exchange Neighborhood (NEN) [Gho03] verwendet, die darin besteht, in einem Cluster einen Knoten verändern. Für eine Lösung $P = \{p_1, p_2, ..., p_n\}$, wobei p_i der Knoten ist, der aus Cluster V_i verbunden wird die Nachbarschaft wie folgt definiert: Sie besteht aus allen Vektoren, die dadurch erreicht werden können, dass in einem Cluster V_i genau ein Knoten p_i durch einen Knoten aus demselben Cluster ersetzt wird. Außerdem wird noch eine zweite, größere Ghosh-Nachbarschaft, die darin besteht, jeweils einen

Knoten in zwei Clustern auszutauschen. Diese wird aufgrund ihrer Größe nur teilweise durchsucht, d.h. es wird ein Zeitlimit verwendet, und die bis dahin beste Lösung, die in dieser Nachbarschaft gefunden werden konnte, übernommen. Analog zur Pop-Darstellung wird eine ursprünglich von Pop definierte Nachbarschaft [Pop02] verwendet, die darin besteht, den globalen Graphen T_g zu verändern. Zu dieser Nachbarschaft gehören alle gültigen Spannbäume, die sich in genau einer Kante von der ursprünglichen Lösung unterscheiden. Um den Aufwand in einem vertretbaren Ausmaß zu halten, wird die Bewertung für diese Lösungen inkrementell [Lei06] berechnet. Die beiden Optimierungsarten werden grundsätzlich nacheinander aufgerufen, bis keine Verbesserung mehr erzielt werden kann. Alternativ dazu kann auch nur entweder die Verbesserung basierend auf Ghosh bzw. Pop verwendet werden, was über einen Parameter eingestellt werden kann.

2.7 Der Trie als Datenstruktur für das Lösungsarchiv

In der vorliegenden Arbeit werden immer Integer-Vektoren im Trie gespeichert. Da diese immer dieselbe Länge haben, können Lösungen grundsätzlich nur in den Blättern des Tries markiert werden. Wenn eine Lösungkodierung aus n Integern besteht, so haben die Trieknoten in den ersten n-1 Ebenen lediglich Referenzen auf weitere Trieknoten, die Lösungen werden dann in der n-ten Ebene durch mit der Markierung *Vollständig* gespeichert. Daher ist es nicht nötig, in einem Knoten für einen gegeben Buchstaben sowohl eine Lösung also auch eine Referenz auf weitere Knoten zu speichern. Aufgrund der Funktion, vollständig besuchte Knoten zu löschen und sie durch die Markierung *Vollständig* zu ersetzen, können die Wörter aber in Ausnahmefällen dennoch unterschiedlich lang sein. Dies ist aber kein Problem, da in diesem Fall keine Referenz auf einen Unterknoten erforderlich sind, da alle Lösungen, die durch so einen Knoten beschrieben werden, bereits besucht wurden und dies mit der Markierung *Vollständig* kodiert wird. Die Funktion zum Löschen von vollständigen Knoten wird in der Darstellung der Einfügen-Funktion beschrieben. Ein solcher Trie wird in Abbildung 2.6 illustriert.

Mögliche Zustände der Referenzen

Eine Referenz, also der Nachfolger dieser Knoten, kann grundsätzlich drei Zustände haben. Der Zustand *Null* bedeutet, dass noch keine Lösung mit dem entsprechenden Vorgänger aufgenommen wurde. Der Zustand *Vollständig* bedeutet dagegen, dass alle Lösungen, die von dieser Referenz ausgehen können, bereits vorgekommen sind. Der Zustand *Unvollständig* bedeutet, dass mindestens eine Lösung aufgenommen wurde, aber es auch noch mindestens eine noch nicht besuchte Lösung gibt. Für die Predecessor-Darstellung gibt es darüber hinaus noch den Zustand *Verboten*, der ungültige Lösungen markiert. In den Illustrationen wird ein *Null* als leeres Feld dargestellt, *Vollständig* als **C**, *Verboten* als **F** und *Unvollständig* als **X**.

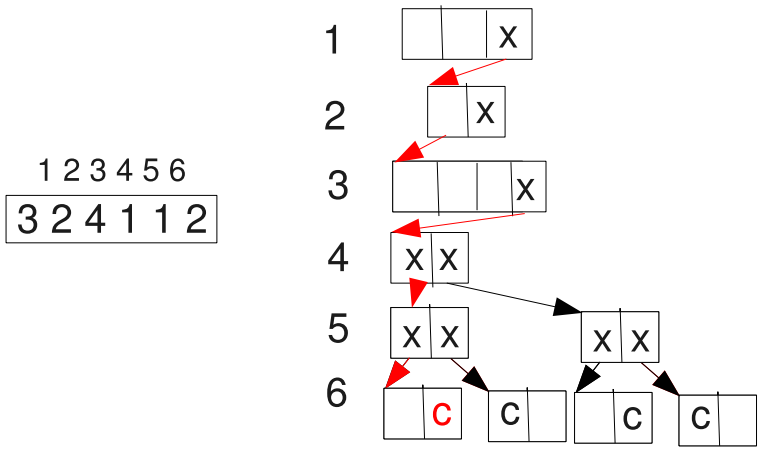

Abbildung 2.6: Ein Trie als Archiv für Lösungen, die als Integer-Vektor kodiert werden.

2.8 Kodierung des Ghosh-Tries

Für die Kodierung nach den Knoten wird ein einfacher Integer-Vektor verwendet, bei dem das n-te Element den n-ten Cluster repräsentiert. Im Trie entspricht die n-te Ebene dem n-ten Cluster. Die Größe der Trieknoten bzw. die Anzahl der nötigen Referenzen hängt von der Anzahl der Knoten im entsprechenden Cluster ab. Der Trie wird in Abbildung 2.7 illustriert. Die Operationen des Tries sind im Wesentlichen analog zu dem des Pop-Tries. Die Anzahl der Trie-Ebenen und die Anzahl der Knoten in den Trieknoten sind zwar unterschiedlich, und es sind keine Überprüfungen notwendig, ob die konvertierten Lösungen gültig sind, dennoch sind die grundlegenden Operationen wie Einfügen und Konvertieren analog zu denen des Pop-Tries, die im nächsten Abschnitt beschrieben werden.

2.9 Kodierung des Pop-Tries

Die Darstellung nach globalen Kanten entspricht einem Spannbaum. Daher stellt sich die Frage, wie der Baum im Trie kodiert werden soll. Es gibt hierfür eine Vielzahl von Darstellungsmöglichkeiten [PK94, RJ03]. Der EA verwendet für die Funktionen, die auf der Pop-Darstellung basieren, die Predecessor-Darstellung, daher bietet es sich an, diese auch im Trie zu verwenden. Ein Problem dieser Darstellung ist jedoch, dass nicht jeder Graph, der hiermit dargestellt werden kann, auch wirklich ein Baum ist. Als Alternative dazu wird die Prüferkodierung verwendet. Diese hat der Vorteil, dass jeder darstellbare Code ein gültiger Baum ist und somit klassische Rekombination- und Mutationsoperatoren auf Arrays ausgeführt werden können. Als Nachteil wird jedoch angeführt, dass deren Lokalität gering ist [PK94], d.h. eine geringe Änderung, wie z.B. die Veränderung nur einer Stelle der Lösung kann sie stark verändern. Da sich diese Untersuchungen jedoch auf klassische EA-Operatoren wie Mutation und Rekombination beziehen, nicht

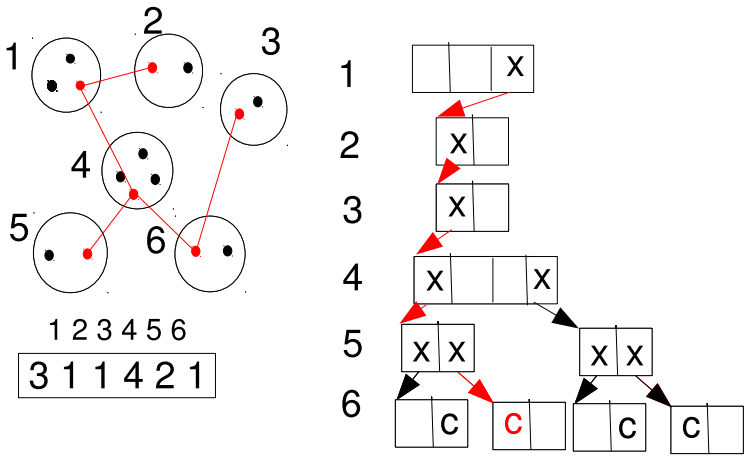

Abbildung 2.7: Eine Lösung in Ghosh-Darstellung. Im Trie ist diese Lösung mit roten Pfeilen markiert.

jedoch auf den Anwendungsfall eines Lösungsarchives, soll hier auch untersucht werden, inwieweit diese Kodierung geeignet ist.

Der Prüfer-Code wird wie folgt berechnet: Es wird im Baum jenes Blatt gesucht, dessen Nummer die kleinste ist. Die Nummer des Knotens, mit dem das Blatt verbunden ist, wird dem Code hinzugefügt. Das Blatt wird gelöscht und dieser Vorgang so lange wiederholt, bis nur mehr zwei Knoten übrig sind. Der in Abbildung 2.3 dargestellte Spannbaum hat dann den Prüfercode *5331* . Der Algorithmus zu dessen Generierung bzw. zu der Rückkonvertierung wird weiter unten beschrieben. Da der EA für die Operationen auf Basis der Pop-Darstellung die Predecessor-Kodierung verwendet, muss die Lösung zuerst in die Prüfer-Kodierung umgewandelt werden, wenn der Prüfer-Trie verwendet wird. Da alle diese Codierungen wie die Ghosh-Darstellung auf Integer-Arrays basieren, wird immer auch ein ähnlicher Trie verwendet. Sie unterscheiden sich nur in der Tiefe bzw. der Größe der einzelnen Trie-Knoten. Bei Verwendung der Predecessor-Darstellung ist der Trie bei n Clustern $n-1$ Ebenen tief, da der Wurzelknoten keinen Vorgänger besitzt. Ein Knoten des Tries hat dann höchstens $n-1$ Referenzen auf Nachfolger, da der Knoten nicht Vorgänger seiner selbst sein kann.

In Abbildung 2.8 ist ein Beispiel mit sechs Clustern zu sehen. Der Trie hat fünf Ebenen und jeder Knoten hat fünf Referenzen. Die Blätter enthalten Lösungen, die anderen Knoten Referenzen zu den Ebenen tiefer im Trie.

Der Prüfer-Trie ist bei n Clustern $n-2$ Ebenen tief, da auch der Prüfer-Code $n-2$ Stellen hat. Ein Knoten hat dann n Referenzen. In Abbildung 2.9 ist dieselbe Lösung in der Prüfer-Darstellung zu sehen. Die Instanz hat sechs Cluster, der Trie vier Ebenen zu jeweils sechs Knoten.

Der Trie hat somit $O(n)$ Ebenen mit Knoten, die bis zu jeweils $O(n)$ Referenzen auf weitere Knoten haben. Die n-te Ebene könnte dann theoretisch aus $O(n^n)$ Referenzen bestehen, was aber für die Praxis keine Bedeutung hat, da die Anzahl der Lösungen durch

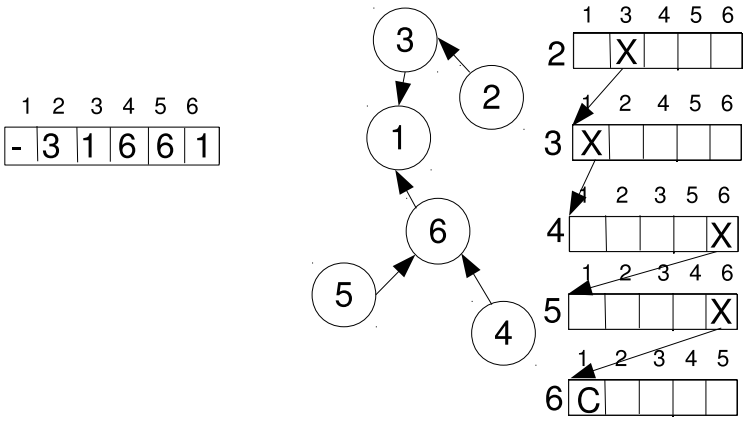

Abbildung 2.8: Eine Lösung in Predecessor-Darstellung und dessen Darstellung im Trie

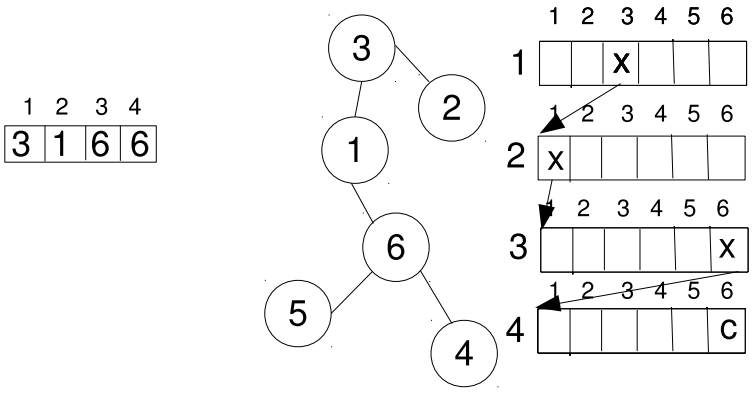

Abbildung 2.9: Eine Lösung in Prüfer-Darstellung und dessen Darstellung im Trie

die Laufzeit des EA begrenzt ist. Der zusätzliche Speicherverbrauch für das Einfügen einer einzelnen neuen Lösung kann im schlechtesten Fall n^2 sein, da es $O(n)$ Ebenen und $O(n)$ Referenzen pro Ebene gibt. Bei dieser Datenstruktur entstehen somit in jedem neuen Knoten $O(n)$ Null-Zeiger. Bei der Einfügung der ersten Elemente entstehen somit $O(n^2)$ Null-Zeiger, was einem recht hohen Speicherverbrauch entspricht. Da dann jedoch neue Lösungen, die sich in nur in den unteren Stellen von einer bereits vorhandenen Lösung unterscheiden, mit wenig Speicherverbrauch (im besten Fall muss nur ein Null-Zeiger auf *Vollständig* geändert werden, um eine neue Lösung zu markieren) hinzugefügt werden können, ist zu erwarten, dass der Speicherverbrauch nicht explodieren wird. Wolf [Wol09] hat in seiner Arbeit festgestellt, dass der theoretische Speicherverbrauch für den Ghosh-Trie für größere Instanzen zwar Hunderte von Terabyte betragen könnte, dies jedoch durch die Laufzeit des Algorithmus begrenzt wird. In der Praxis erreichte er Werte von einigen Dutzend MB. Da sein Trie jedoch $O(n)$ Ebenen und eine Ebene nur

$O(n/m)$ (wobei m die durchschnittliche Zahl der Knoten pro Cluster ist) Knoten hat, die Knoten des Trie in der Spannbaum-Darstellung jedoch $O(n)$ Referenzen haben, ist ein Speicherverbrauch von mehreren Hundert MB durchaus möglich. Der Speicherverbrauch wird dann im Zuge der Tests beobachtet.

2.10 Die Operationen des Tries

Es gibt grundsätzlich nur zwei Operationen, Einfügen und Konvertieren. Die Funktion der Einfügen-Operation besteht darin, die neue Lösung einzufügen, wenn diese noch nicht im Trie existiert. Die Konvertierungs-Funktion erstellt ausgehend von einer bereits besuchten Lösung eine neue, garantiert unbekannte. Eine eigene Funktion für das Suchen einer Lösung bzw. Prüfung, ob eine Lösung im Trie bereits vorhanden ist, ist nicht notwendig, da dies implizit über die Funktion zum Einfügen geschieht. Bei den Trie-Funktionen für die Predecessor-Darstellung muss berücksichtigt werden, dass es im Trie nicht erforderlich ist, für einen Knoten, der die möglichen Vorgänger speichert, eine Referenz auf sich selbst zu enthalten. Dieser Umstand macht den Programmcode zwar etwas unübersichtlicher, ist aber für die grundsätzliche Idee nicht wesentlich.

Die Einfügen-Funktion für die Predecessor-Codierung

Das Einfügen einer Lösung in den Trie mit Predecessor-Kodierung wird in Algorithmus 2 dargestellt. Zunächst wird überprüft, ob die Lösung bereits im Trie enthalten ist. Dazu wird in jedem Trieknoten dem Zeiger gefolgt, der der kodierten Lösung im Trie entspricht. Stößt die Suche dabei auf einen Nullzeiger, ist die Lösung nicht enthalten und sie wird neu eingefügt, wobei gegebenenfalls fehlende Trieknoten ebenfalls eingefügt werden. Stößt die Suche dagegen auf einen *Vollständig*-Pointer, so ist die Lösung bereits vorhanden.

In der Predecessor-Kodierung wird für jeden Cluster sein Vorgänger gespeichert. Da der Graph, der die Verbindungen zwischen den einzelnen Clustern darstellt, ein Baum ist und somit keine Zyklen enthalten darf, ist dies zu berücksichtigen. Es ist mit dieser Darstellung somit möglich, Lösungen zu kodieren, die keine Bäume sind und die Zyklen enthalten. Diese ungültigen Lösungen müssen jedoch als solche erkannt werden bzw. gar nicht erst generiert werden. Es gibt somit prinzipiell mehrere Stellen, an denen dies erfolgen kann. Einerseits könnten beim Einfügen einer neuen Lösung alle ungültigen Lösungen in den Knoten der Tries markiert werden, oder aber es könnte in der Konvertierungs-Methode darauf geachtet werden, keine ungültigen Lösungen zu generieren. Da es aber nicht möglich ist, alle ungültigen Lösungen gleich in der Einfügen-Methode zu markieren, muss auch bei der Konvertierungs-Methode darauf geachtet werden, dass keine ungültigen Lösungen entstehen. Nach dem Einfügen der neuen Lösung wird noch überprüft, ob dadurch Trieknoten vollständig geworden sind, d.h. ob sie nur Knoten mit dem Zustand *Vollständig* bzw. *Verboten* aufweisen. Ist dies der Fall, so kann der Knoten gelöscht werden und die auf diesen Knoten verweisende Referenz im Knoten darüber als *Vollständig* markiert werden. Da dadurch auch der darüber liegende Knoten vollständig werden kann, muss diese Funktion rekursiv aufgerufen werden. Der Trie kann somit grund-

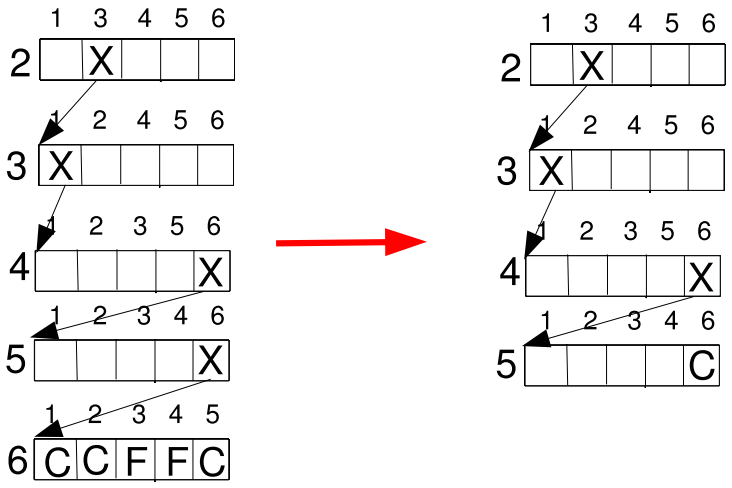

Abbildung 2.10: Ein vollständiger Trieknoten wird gelöscht

sätzlich wieder Speicherplatz freigeben, wenn zusätzliche Lösungen eingefügt werden. Angesichts des großen Lösungsraumes ist diese Funktion jedoch wohl nur bedingt dazu geeignet, in der Praxis Speicherplatz einzusparen. Der Vorgang des Zusammenfassen eine vollständigen Trieknotens wird in Abbildung 2.10 illustriert.

Berechnung der Verboten-Felder

Die Berechnung der *Verboten*-Felder erfolgt rekursiv. Der Pseudocode ist in Algorithmus 2.10 angegeben. Damit in dem Graph kein Zyklus entsteht, darf ein Cluster keinen Vorgänger haben, der wiederum diesen als Vorgänger hat. Wenn die *Verboten*-Felder eines Clusters berechnet werden sollen, sind alle Cluster, die einen kleineren Index haben als der gegenwärtige, Kandidaten für verbotene Cluster. Dann wird überprüft, ob es unter diesen einen bereits hinzugefügten Cluster gibt, dessen Nachfolger der gegenwärtige Cluster ist. Ist dies der Fall, so würde eine Lösung mit einem Zyklus entstehen und dieser wird als *Verboten* markiert. Wird ein *Verboten*-Feld markiert, so wird weiter rekursiv untersucht, ob es bereits hinzugefügte Cluster gibt, dessen Nachfolger der nun verbotene Cluster ist. Solche wären ebenfalls ungültige Lösungen, die markiert werden müssen. In Abbildung 2.11 wird gezeigt, dass es von der Reihenfolge der Cluster abhängt, ob und wieviele Felder als *Verboten* markiert werden können. Damit ein Feld in einem Cluster als *Verboten* markiert werden kann, müssen dessen Vorgänger bereits vorher im Trie gespeichert worden sein. Im linken Beispiel können keine Cluster als *Verboten* markiert werden, weil die Cluster, die Blätter sind, einen höheren Index als ihre Vorgänger haben, im rechten ist es dagegen umgekehrt. Weiterhin kann man an diesem Beispiel sehen, dass wenn man an der linken Lösung im Trieknoten für den Cluster 2 die Lösung von 1 auf 3 verändert, sich ein Zyklus ergeben würde, der von dieser Funktion nicht verhindert werden kann. Daher ist es nötig, dass die Prüfung bei der Konvertierungs-Funktion

Algorithmus 2 Einfügen mit Markierung von ungültigen Lösungen

Eingabe: Lösung $sol : S^g =< V^G, T^G >$
Ausgabe: boolean: Einfügen erfolgreich?

```
1: for (i = 1; i < sol.size(); i++) do
2:     numberOfPredecessors[sol[i]]++ // Anzahl der Vorgänger für jeden Cluster,
       wird für die Berechnung der Verboten-Felder gebraucht
3: end for
4: if root = null then
5:     generiere neuen root und füge diesen ein
6: end if
7: curr = root
8: for (i = 1; i < sol.size() − 1; i++) do
9:     pos = sol[i];
10:    if curr.next[pos] == complete then
11:        return false
12:    else if curr.next[pos]==null then
13:        if i == anzahlDerTrieEbenen then
14:            // wenn die letzte Ebene erreicht wird
15:            curr.next[pos] = complete
16:        else
17:            curr.next[pos] = neuer Trieknoten
18:            for (j = 0; j < sol.size(); j++) do
19:                checkPredecessors(sol, numberOfPredecessors, curr.next[pos],
                   j)
20:                //für den Trieknoten Verboten-Felder berechnen
21:            end for
22:        end if
23:    end if
24:    curr = curr.next[pos]
25: end for
26: überprüfe, ob ein Trie-Knoten vollständig geworden ist
27: return true
```

erfolgt.

Die Konvertierungs-Funktion

Die Konvertierungs-Funktion dient dazu, eine Lösung zur verändern und somit zu einer neuen Lösung zu kommen. Es werden zwei verschiedene Konvertierungsfunktionen verwendet, die randomisierte Konvertierung und die Konvertierung von unten.

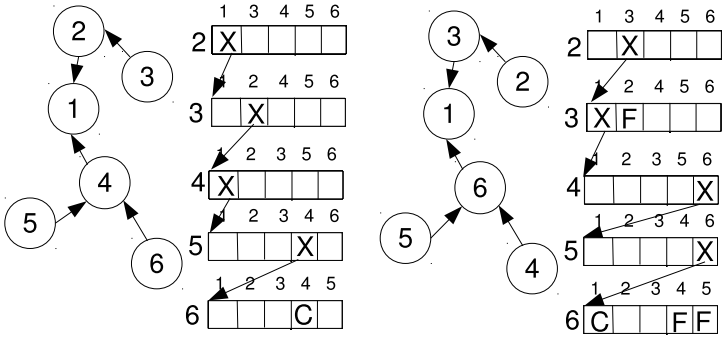

Abbildung 2.11: Die Markierung von Verboten-Feldern hängt von der Reihenfolge ab.

Algorithmus 3 checkPredecessors(Lösung *sol*, int[] $numberOfPredecessors$, Trienode n, int *node*): Berechnung von ungültigen Lösungen

```
1: if numberofpredecessors[node]==0 then
2:     return;
3: end if
4: if if (sol[pred] == node ∧ pred bereits oben im Trie hinzugefügt) then
5:     n.next[pred] = verboten
6:     for (int i = 0; i < sol.size(); i++) do
7:         checkPredecessors(sol, n, numberOfPredecessors, i);
8:     end for
9: end if
```

Die randomisierte Konvertierung

Der Trie wird somit entlang der Lösung durchsucht, während an einer geeigneten Stelle eine Änderung vorgenommen wird. Der Pseudocode zu dieser Funktion ist in Algorithmus 4 angegeben. Wie aber aufgrund der anderen Speicherstruktur der Predecessor-Darstellung schon beim Einfügen Abhängigkeiten auftraten, die es zu behandeln galt, so ist dies auch in der Konvertierungs-Funktion erforderlich. Es ist daher nicht möglich, einfach eine Lösung beliebig zu verändern, ohne zu überprüfen, ob das Ergebnis wieder eine gültige Lösung ist. Bei der Erkennung eines Zyklus wird der Vorgänger des Clusters, dessen neuer Nachfolger in den Zyklus führen würde, auf den alten Nachfolger zu verweisen. Der Pseudocode für diese Funktion ist in Algorithmus 5 zu finden. Allerdings wird so die Lösung an mehreren Stellen verändert, was somit zu Folge haben kann, dass die neue Lösung wieder ein bereits im Archiv enthaltendes Duplikat sein könnte. Daher kann nicht garantiert werden, dass es sich tatsächlich um eine unbesuchte Lösung handelt. Für den Fall, dass die Korrektur-Funktion ein Duplikat erstellt, wird die Konvertierungs-Funktion erneut aufgerufen, um mit diesem Duplikat als Ausgangspunkt eine neue Lösung zu generieren. Gegebenfalls kann dies auch mehrmals hintereinander

Algorithmus 4 randomisiertesKonvertieren(): Berechnung einer neuen Lösung durch Veränderung einer bereits enthaltenen

Eingabe: Lösung $sol : S^g =< V^G, T^G >$
Ausgabe: veränderte Lösung *sol*

```
1: berechne alle Startpunkte, d.h. Trieknoten entlang der Lösung, wenn diese nicht
   Vollständig sind
2: trienode curr = ein zufälliger Startpunkt;
3: while curr != null do
4:     suche in diesem Knoten einen Null-Zeiger
5:     if Null-Zeiger gefunden then
6:         Lösung gefunden, Funktion kann abgebrochen werden
7:         überprüfe, ob kein Zyklus entstanden ist
8:     else
9:         if die Referenz entlang der Lösung sol nicht vollständig ist then
10:            folge der Lösung
11:        else
12:            suche einen beliebigen noch nicht vollständigen Zeiger
13:            überprüfe, ob kein Zyklus entstanden ist
14:        end if
15:    end if
16: end while
```

auftreten; dann wird die Konvertierungs-Funktion solange aufgerufen, bis eine neue Lösung gefunden wird. Allerdings kann die Lösung dann dementsprechend stark im Vergleich zur Ursprungslösung verändert werden. In Abbildung 2.13 wird die Korrektur eines Zyklus illustriert. In diesem Fall, wo der Cluster 4 der Vorgänger des Clusters 3 werden soll, würde ein solcher unzulässiger Zyklus entstehen. Um dies zu verhindern, wird der Vorgänger von 3, der sich im Zyklus befindet, also der Cluster 5, an den Nachfolger 4 angebunden. Somit entsteht wieder eine gültige Lösung. Allerdings wurde die Lösung somit an zwei Stellen verändert.

Grundsätzlich werden die Knoten entlang der Lösung analog zu Wolfs Lösung [Wol09] gespeichert, und eine davon als Startpunkt zufällig ausgewählt. Dies geschieht darum, um den sogenannten Bias zu verringern, also um die Wahrscheinlichkeit zu erhöhen, dass die Lösung an einem zufälligen Knoten verändert wird. Ausgehend von diesen Punkt wird ein Null-Zeiger gesucht. Kann ein solcher nicht gefunden werden, wird die Lösung weiter nach unten verfolgt, es wird also der nächste Cluster der Lösung verwendet, wo wieder ein Null-Zeiger gesucht wird. Ist es nicht mehr möglich, diese Lösung weiterzuverfolgen, weil sie bereits vollständig ist, so wird ein zufälliger Cluster, der noch nicht vollständig ist, ausgewählt. Immer, wenn die Lösung verändert wird, muss überprüft werden, ob ein Zyklus entstanden ist, also wenn ein anderer, noch nicht vollständiger Knoten verfolgt wird, oder aber ein Null-Zeiger gefunden wurde, der die neue Lösung darstellt. Diese Konvertierungsart wird randomisierte Konvertierung genannt.

Algorithmus 5 zyklusÜberprüfen(): überprüft, ob ein Zyklus vorhanden ist und korrigiert ihn gegebenenfalls

Eingabe: Lösung $sol : S^g =< V^G, T^G >$, int $veraenderterCluster$, int $alterVorgaenger$
Ausgabe: ggf. veränderte Lösung $sol : S^g =< V^G, T^G >$

```
1: int pred;
2: while pred != wurzelcluster ∧ pred != veraenderterCluster do
3:     pred = sol[ veraenderterCluster ]
4: end while
5: if pred == wurzelcluster then
6:     // wenn der Wurzelknoten erreicht wurde:
7:     // kein Zyklus, die Lösung wird nicht verändert
8:     return sol
9: else
10:    // Zyklus wurde entdeckt
11:    bestimme Cluster c, der im Zyklus der Vorgänger von veraenderterCluster ist
12:    Vorgänger von c = alterVorgaenger
13: end if
```

Die Konvertierung von unten

Eine Variante zu dieser Konvertierungs-Funktion besteht darin, den Startpunkt nicht zufällig, sondern so weit unten wie möglich zu wählen. Diese wird dann Konvertierung von unten genannt. Dadurch ist es möglich, Speicherplatz zu sparen, da die Lösungen im Durchschnitt dann längere gemeinsame Präfixe aufweisen. Als Nebeneffekt können vollständig durchsuchte Trieknoten früher entfernt werden, was auch ein wenig Speicherplatz einsparen kann. Allerdings werden denn bestimmte Cluster - die weiter unten im Trie - öfter verändert als die andere - die weiter oben. Ob sich dadurch die Qualität der gefundenen Lösungen verschlechtert, wird in den Tests untersucht werden.

Der Aufwand der beiden Konvertierungsvarianten

Der Aufwand für die randomisierte Konvertierungs-Funktion ist für den Prüfer-Trie im schlechtesten Fall in $O(n^2)$, wobei n der Anzahl der Cluster entspricht. Ausgehend vom Wurzelknoten muss höchstens n-mal bis in die letzte Trieebene hinuntergegangen werden, während die Suche nach Null-Zeigern bzw. Zeigern, die *Unvollständig* sind, in einem Vektor einen Aufwand von $O(n)$ verursacht. Das entspricht einem Gesamtaufwand von $O(n^2)$.

Bei der Konvertierung von unten könnte, wenn der Trie relativ voll ist, im schlechtesten Fall ebenfalls ein quadratischer Aufwand entstehen. Allerdings kommt das in der Praxis kaum vor. Nachdem die Lösung möglichst weit unten verändert wird, und die Suche solange entlang der Lösung verfolgt wird, bis auf einen *Vollständig*-Zeiger gestoßen wird, muss im einfachen Fall der Trie n mal hinunterverfolgt werden, während das Finden des Null-Zeigers ebenfalls einen Aufwand von $O(n)$ verursacht. Wenn vollständige

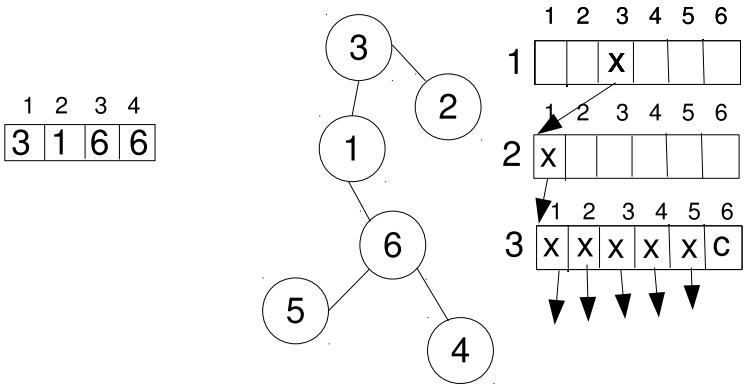

Abbildung 2.12: Bei Konvertierung von unten kann es auch einen größeren Aufwand erfordern, eine neue Lösung zu finden.

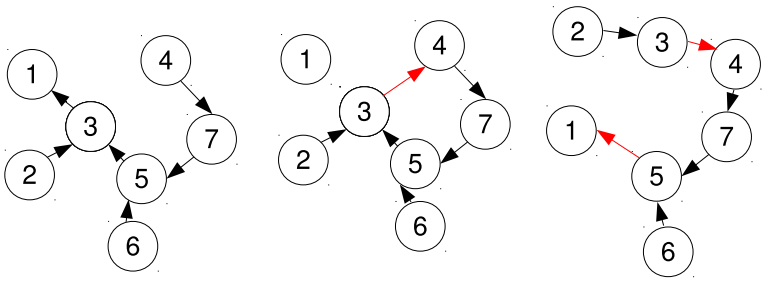

Abbildung 2.13: Die Konvertierungs-Funktion verhindert das Auftreten von Zyklen

Knoten aber zusammengefasst werden, so ist es auch möglich, dass *Vollständig*-Zeiger nicht nur in der letzten Trieebene auftreten. Wenn nun in so einem Trieknoten ein unvollständiger Zeiger gesucht werden muss, so verursacht dies einen zusätzlichen Aufwand in der Ordnung $O(n)$. Diese Situation wird in Abbildung 2.12 illustriert. In einem Prüfer-Trie soll die Lösung *3166* konvertiert werden. Es sind jedoch bereits alle Lösungen mit dem Präfix *316* eingefügt wurden, was zur Folge hat, dass der Trieknoten auf der vierten Ebene durch einen *Vollständig*-Zeiger in der dritten Ebene ersetzt wurde. Daher stößt die Konvertierungs-Funktion bereits in der dritten Ebene auf einen *Vollständig*-Zeiger. Nachdem auch kein Null-Zeiger in dem Trieknoten vorhanden ist, muss ein zufälliger *Unvollständig*-Zeiger weiterverfolgt werden. Wenn die Konvertierungs-Funktion theoretisch bereits in der ersten Ebene auf einen *Vollständig*-Zeiger stoßen würde, und sie im Zuge der Suche in jeder Ebene auf einem *Vollständig*-Zeiger stoßen würde, so wäre der Aufwand für diese Funktion quadratisch.

Zur Terminierung der Konvertierungs-Funktion im Predecessor-Trie

Aufgrund der Auftretens von Zyklen kann für den Predecessor-Trie nicht garantiert werden kann, dass die Konvertierungs-Funktion beim ersten Aufruf eine neue, noch nicht

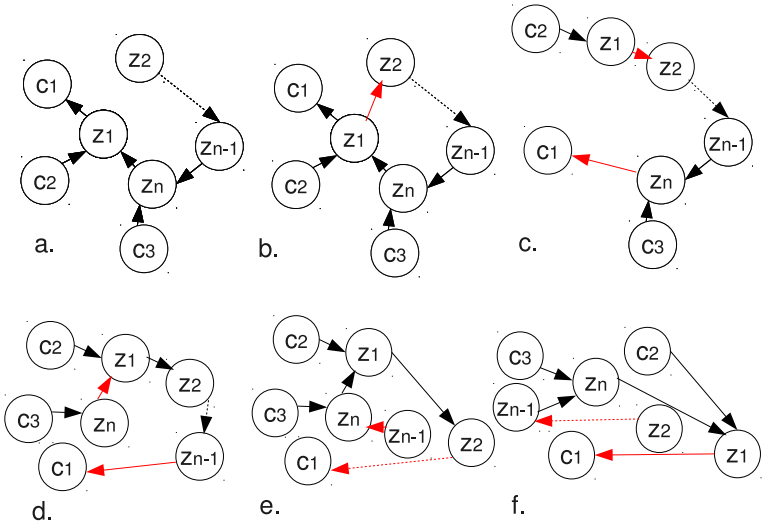

Abbildung 2.14: Zur Terminierung der Konvertierungs-Funktion im Predecessor-Trie

besuchte Lösung erstellt. Die Lösung für diesen Fall besteht darin, die Funktion erneut aufzurufen. Bei den Tests (siehe Abschnitt 3.10) hat die Funktion zwar immer spätestens nach fünf Reparaturen eine unbesuchte Lösung gefunden. Dennoch stellt sich die Frage, ob es Fälle gibt, in keine neue Lösung gefunden werden kann, wenn die Funktion zu einer Lösung springt, die bereits im Zuge der Konvertierungs-Funktion vorkommen ist. Damit die Funktion zur selben Lösung springt, muss die Konvertierungs-Funktion in einen Zyklus geraten.

In Abbildung 2.14 wird dieser Fall illustriert. Der Vorgänger eines Clusters wird hier wie folgt beschrieben: $Pred(c_1) = c_2$, der Vorgänger von c_1 ist c_2. Vor der Konvertierung gilt $Pred(z_1) = c_1$ (Abb. 2.14a). Die Konvertierungs-Funktion verändert die Lösung nun so, dass gilt: $Pred(z_1) = z_2$ (Abb. 2.14b). Dann wird der alte Vorgänger wie folgt referenziert: $Pred_{old}(z_1) = c_1$. Es ergibt sich ein Zyklus $z_1, z_2, ..., z_n$. Die Reparatur-Funktion verändert die Lösung so, dass gilt: $Pred(z_n) = Pred_{old}(z_1) = c_1$ (Abb. 2.14c). Die Funktion könnte nur dann zu der alten Lösung kommen, wenn die Konvertierungs-Funktion danach nacheinander $n-1$ Mal wie folgt aufgerufen wird, dass bei ersten Aufruf die Lösung verändert wird, dass gilt: $Pred(z_n) = z_1$; ab dem zweiten Aufruf die Lösung beim i-ten Aufruf verändert wird, dass gilt: $Pred(z_{n-i+1}) = z_{n-1}$. Es wird also immer der Vorgänger von c_1 verändert. Dadurch entsteht immer wieder ein Zyklus, was zur Folge hat, dass die Reparatur-Funktion die Lösung $n-1$ Mal verändert, wobei sie beim i-ten Aufruf die Lösung repariert, dass gilt: $Pred(z_{n-i}) = c_1$ (Abb. 2.14d-f). Somit würde die Funktion nach insgesamt n Konvertierungen zu der ursprünglichen Lösung kommen. Jedoch wird diese Folge an Reparaturen bei dem Cluster mit dem höchsten Label-Index im Zyklus, also $z_h : Label(z_h) = max(Label(z_1, ..., z_n))$ unterbrochen. Es gilt für diesen Cluster z_h, dass $Label(z_h) \geqslant Label(z_i) \forall z_i$; es kann ein Zyklus nur für solche Cluster entstehen, deren Label-Index geringer ist als der von z_h. Da die Funktion zur Markierung

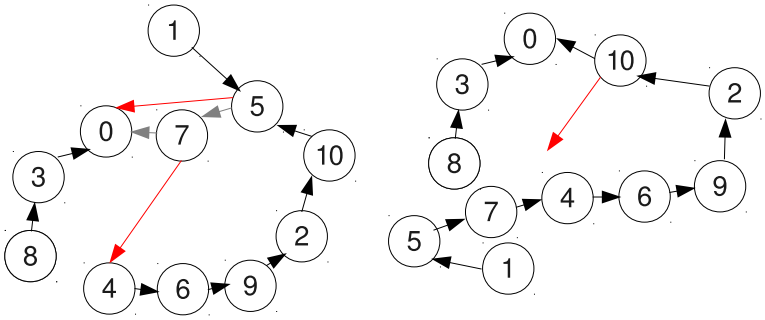

Abbildung 2.15: Zum Abbruch der Folge an Konvertierungen bei dem Cluster mit dem höchsten Wert

von *Verboten*-Feldern beim Einfügen dieser Lösung in den Trie in diesem Fall alle diese Cluster als *Verboten* markieren kann, kann die Konvertierungs-Funktion nicht in diesen Zyklus gelangen. Dieser Fall wird in Abbildung 2.15 illustriert. Im ersten Fall kann für den Cluster 7 nur ein *Verboten*-Feld markiert werden, nämlich für den Cluster 5, für die restlichen jedoch nicht, da der Cluster 10 erst weiter unten im Trie vorkommt. Im zweiten Fall können für den Cluster mit dem höchsten Wert 10 alle Cluster, deren direkter oder indirekter Vorgänger 10 ist, als *Verboten* markiert werden, da alle einen niedrigeren Wert haben und somit bereits weiter oben im Trie vorgekommen sind.

Obwohl für die Praxis nicht relevant, ist es möglich, einen Worst-Case-Fall zu konstruieren, bei dem $O(n^2)$ Konvertierungen nötig sind. Dieser tritt auf, wenn der Baum nur aus einem Weg besteht, der ausgehend von der Wurzel aufsteigend sortierte Label-Indizes aufweist. Dieser Fall wird in Abbildung 2.16 illustriert. Ausgehend von Abb. 2.16a wird der Vorgänger von 2 auf n geändert, danach der Vorgänger von 3 auf 2 usw. Der Baum nach $n-2$ Konvertierungen, wenn die Funktion an dem Cluster mit dem höchsten Label-Index abbricht, ist in Abb. 2.16b zu sehen. Nun ist es aber wieder möglich, den Vorgänger von 2 auf $n-1$ zu setzen, was im schlechtesten Fall $n-3$ Konvertierungen erforderlich macht. Der Baum nach diesen Konvertierungen ist in Abb. 2.16c zu sehen. Dieser Vorgang kann insgesamt $n-2$ Mal auftreten, die Anzahl der Konvertierungen verringert sich jeweils immer um eins, also $\sum_{i=1}^{n-2} i$, was einem quadratischen Aufwand entspricht.

2.11 Die Operationen für die Prüfer-Darstellung

Da die Prüfer-Zahlen, wie auch die Predecessor-Darstellung, aus einem Integer-Array bestehen, ist das Einfügen und das Konvertieren der Lösungen bis auf die geringfügig unterschiedliche Tiefe des Tries und die Länge des Arrays gleich. Außerdem ist es bei der Prüfer-Darstellung nicht erforderlich, ungültige Lösungen zu markieren bzw. nach der Konvertierung einer Lösung zu überprüfen, ob ein Zyklus entstanden ist. Davon abgesehen ist die prinzipielle Funktionsweise aber gleich. Da der EA jedoch grundsätzlich mit der Predecessor-Darstellung arbeitet, ist eine Konvertierung erforderlich. Diese Opera-

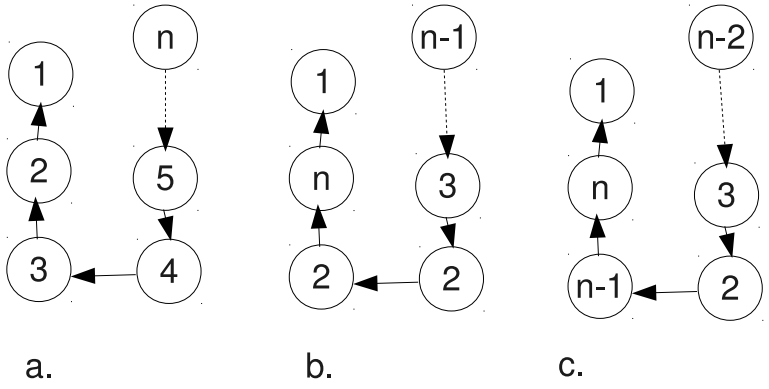

Abbildung 2.16: Zum Worst-Case

tionen werden nun im Folgenden dargestellt.

Konvertierung der Predecessor-Darstellung in die Prüfer-Kodierung

Es wird grundsätzlich der klassische Algorithmus [PK94] verwendet. Der Pseudocode ist in Algorithmus 6 angegeben. Zunächst werden die Kanten in Form einer Adjazenzliste aus der Predecessor-Darstellung berechnet, und der Grad der Clusters bestimmt und diese in einen Vektor gespeichert. Danach wird laufend der Cluster mit der kleinsten Nummerierung gesucht, dessen Grad 1 beträgt. Der Cluster, mit dem das Blatt noch verbunden ist, wird in den Prüfercode übernommen. Um dies effizient zu gestalten, wird der Vektor nur einmal durchlaufen, da der nächste Cluster mit Grad 1 grundsätzlich nur auf einem Feld mit einem höheren Index sein kann. Da die Kanten allerdings entfernt werden, wenn ein Cluster in die Prüferliste aufgenommen wird, ist es auch möglich, dass dadurch ein neuer Cluster Grad 1 erhält. Falls also ein Cluster, dessen Index kleiner als der zuletzt verwendete Cluster ist, in die Auswahl kommt, wird der Index direkt auf diese Stelle gesetzt, was ein erneutes Iterieren überflüssig macht. Der Aufwand ist $O(d_{max} * n)$, wobei d_{max} dem Cluster mit dem maximalen Grad entspricht.

Konvertierung der Prüfer-Kodierung in die Predecessor-Darstellung

Die umgekehrte Richtung, also die Umrechnung in die Predecessor-Darstellung, kann im Wesentlichen analog berechnet werden. Algorithmus 7 generiert zuerst eine Liste von Kanten in Struktur einer Adjazenzliste, aus denen dann in weiterer Folge die eigentliche Predecessor-Darstellung gebildet wird. Auch der Index wird analog zu dem vorigen Algorithmus immer nur erhöht, um die Berechnung zu beschleunigen. Zuerst wird der Kandidaten-Vektor berechnet, der angibt, wie oft ein Cluster im Prüfercode vorkommt. Kandidaten sind zu Beginn solche Cluster, die gar nicht im Code vorkommen. Eine Kante zwischen dem Kandidaten und dem Cluster aus dem Prüfercode (im ersten Schritt wird der erste Cluster der Prüfercodes verwendet, im n-ten Schritt der n-te) kann dann

Algorithmus 6 konvertierePredecessorZuPruefer(): Konvertierung in Prüfer-Kodierung

Eingabe: Lösung $sol_{pred} : S^g =< V^G, T^G >$in Predecessor-Darstellung
Ausgabe: Lösung $sol_{pruefer} : S^g =< V^G, T^G >$ in Prüfer-Darstellung

```
1: int[][] adjazenzliste: berechne diese Liste aus der Lösung sol
2: int[] grad: speichert den Grad der Cluster
3: int j = 0; int realindex = 0; int saveindex=-1
4: for (int i = 0; i < sol_pred.size() − 2; i++) do
5:     while j < sol_pred.size() do
6:         if saveindex ==-1 then
7:             realindex = j
8:         else
9:             realindex = saveindex
10:        end if
11:        if degree[realindex]==1 then
12:            Blatt mit kleinstem Label ( adjazenzliste[realindex][0] ) gefunden,
               füge sie in sol_pruefer ein
13:            Kante aus adjazenzliste entfernen und grad aktualisieren
14:            if grad[pos] == 1 ∧ pos < j then
15:                saveindex=pos;
16:            else
17:                saveindex = -1;
18:            end if
19:        else
20:            j++
21:        end if
22:    end while
23: end for
```

hinzugefügt werden. Der Prüfercode wird dann Schritt für Schritt abgearbeitet, im Verlauf der Berechnung werden dann auch solche Cluster zu den Kandidaten hinzugefügt, die nicht mehr in dem Code vorkommen. Wenn die Adjazenzliste erst mal vorliegt, kann die Berechnung der Predecessor-Darstellung effizient mittels der Tiefensuche durchgeführt werden. Insgesamt beträgt der Aufwand für diese Funktion $O(n)$.

2.12 Kombinierung der beiden Sichtweisen

Da Ghosh- und die Pop-Kodierung komplementär sind, werden diese gemeinsam eingesetzt, mit der Absicht, deren Optimierungsmöglichkeiten zu kombinieren und somit die Lösungen weiter verbessern zu können. Dies haben Hu, Leitner und Raidl [HLR08] in ihrem VNS-Ansatz mit Nachbarschaftsstrukturen, die auf den beiden Sichtweisen basieren, getan und gute Resultate erhalten. Daraus ergibt sich die Motivation, dies analog für die Lösungsarchive zu tun.

Algorithmus 7 konvertierePrueferZuPredecessor(): Konvertierung in Predecessor-Kodierung

Eingabe: Lösung $sol_{pruefer} : S^g =< V^G, T^G >$ in Prüfer-Kodierung
Ausgabe: Lösung $sol_{pred} : S^g =< V^G, T^G >$ Predecessor-Kodierung

```
1: berechne Kandidaten-Vektor: candidate
2: int j = 0; int realindex = 0; int saveindex=-1
3: int[][]adjazenzliste
4: for (int i = 0; i < sol.size() − 2; i++) do
5:     while j < candidate.size() do
6:         if saveindex ==-1 then
7:             realindex = j
8:         else
9:             realindex = saveindex;
10:        end if
11:        if candidate[realindex]==0 then
12:            Kandidat mit kleinstem Label gefunden, füge sie mit dem gegenwärtigen Cluster aus der Prüfernummer in adjazenzliste ein
13:            candidate aktualisieren
14:            if candidate[pos] == 0 ∧ pos < j then
15:                saveindex=pos;
16:            else
17:                saveindex = -1;
18:            end if
19:        else
20:            j++
21:        end if
22:    end while
23: end for
24: füge eine Kante zwischen denn übrig gebliebenen Kandidaten ein
25: berechne aus adjazenzliste den Predecessor-Vektor sol_pred
```

Wenn beide Archive verwendet werden, so ist die einfachste Variante, zuerst das Pop-Archiv verwendet, danach das Ghosh-Archiv. Für diesen Fall besteht allerdings die Möglichkeit, dass das zweite Archiv die Lösung so verändert, dass diese wieder ein Duplikat für das erste ist. Daher gibt es die Option, wenn sowohl das Ghosh- als auch das Pop-Archiv verwendet wird, diese solange laufen zu lassen, bis eine Lösung gefunden ist, die in beiden Archiven neu ist und sie somit sowohl in Bezug auf das Ghosh- als auch die Pop-Archiv neu ist. Durch diese Kombinierung der beiden Archive wird erwartet, dass die Lösungen weiter verbessert werden können. Der Pseudocode für die Kombination der beiden Archive ist in Algorithmus 8 angegeben. Die Schleife wird nach einer bestimmten Anzahl von Versuchen abgebrochen, wenn bis dahin nicht gelungen ist, eine Lösung zu finden, die für beide Archive neu ist.

Algorithmus 8 Kombination der beiden Archive

Eingabe: Lösung $sol : S^g =< V^G, T^G >$ (generiert durch den EA), int $maxTries$
Ausgabe: Lösung $sol : S^g =< V^G, T^G >$, neu für beide Archive, wenn diese gefunden wurde, bevor die maximale Anzahl an Versuchen erreicht ist

```
1: popkonvertiert = true
2: ghoshkonvertiert = true
3: i = 0
4: while (ghoshkonvertiert == true ∨ popkonvertiert == true) ∧ i < maxTries do
5:     füge sol in Ghosh-Trie ein, ggf. konvertieren
6:     if keine Konvertierung erforderlich then
7:         ghoshkonvertiert = false
8:     end if
9:     füge sol in Pop-Trie ein, ggf. konvertieren
10:    if keine Konvertierung erforderlich then
11:        popkonvertiert = false
12:    end if
13:    i++
14: end while
```

Ob diese Option die Lösung verbessern kann, wird in den Tests untersucht.

2.13 Implementierung

Der Algorithmus wurde in C++ unter Verwendung der EALIB-Bibliothek implementiert. Dies ist eine Software-Bibliothek, die ein Framework für die Implementierung von heuristischen Optimierungsverfahren bietet. Somit ist es nicht erforderlich, den ganzen Algorithmus selber zu implementieren, sondern nur die problemspezifischen Teile wie die Codierung der Lösungen, die Operatoren, die lokale Optimierung und in diesem Fall auch das Lösungsarchiv. Dazu wurden entsprechende Teile des Codes einerseits aus der Arbeit von Wolf [Wol09] übernommen, der wiederum Teile aus der Arbeit von Hu, Leitner und Raidl [HLR08] verwendet hat. Konkret wurde das Ghosh-Archiv, die Repräsentierung der Lösung in der Ghosh-Darstellung, die Pop-Optimierung und die lokale Optimierung im Wesentlichen übernommen. Es wurden die Klassen Poparchive, Prüfertrie, Prüfernode, Poptrie und Popnode hinzugefügt, die die Funktionalität des Pop-Archives implementieren. Weiters wurden die Konvertierungsfunktionen für die Prüferdarstellung und die Mutation und Rekombination basierend auf der Pop-Codierung in Poparchive implementiert.

KAPITEL 3

Tests und Ergebnisse

3.1 Methodik

Es werden für die Tests die TSPLib-Instanzen[1] verwendet, die u.a. von Hu, Leitner und Raidl [HLR08], aber auch von Wolf [Wol09] benutzt wurden. Dabei handelt es sich um Instanzen, die für das Traveling Salesman Problem (TSP) erstellt wurden, aber insofern verändert wurden, als die Knoten noch Clustern zugeordnet wurden. Diese Zuordnung wurde mit geographischem Clustering durchgeführt [Fer01]. Die verwendeten Instanzen haben 137-442 Knoten sowie 28-89 Cluster. Durchschnittlich wurden somit einem Cluster ca. fünf Knoten zugeordnet. Sie werden in Tabelle 3.1 kurz beschrieben. Es wird die Anzahl der Knoten, die der Cluster sowie die verwendete Laufzeit bei Verwendung einer fixierten Laufzeit angegeben.

Außerdem wurden für diese Tests ähnliche Bedingungen wie in der Arbeit von Wolf verwendet, wodurch ein Vergleich der erzielten Lösungen möglich ist. Die Tests wurden auf dem Grid des Instituts für Computergraphik und Algorithmen der TU Wien laufen gelassen. Da der EA ein randomisierter Algorithmus ist, wurden für jeden Test dreißig Läufe durchgeführt und der Mittelwert $\overline{C(T)}$ sowie die Standardabweichung s_n wie folgt berechnet.

$$\overline{C(T)} = \frac{1}{n}\sum_{i=1}^{n} C(T_i)\ ,\ s_n = \sqrt{\frac{1}{n-1}(\sum_{i=1}^{n} C(T_i) - \overline{C(T)})^2}$$

Grundsätzlich werden zwei verschiedene Optionen zur Terminierung des Algorithmus verwendet: Es wird entweder eine gegebene Anzahl von Generationen ($tgen$) oder eine gegebene Laufzeit in Sekunden ($ttime$) angegeben.

Da der EA mit den beiden Tries und verschiedensten Parametern eine unübersichtlich große Zahl von Optionen zulässt, ist das erste Ziel, ungeeignete Konfigurationen aus-

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

Tabelle 3.1: Übersicht über die Instanzen

Instanz	Anzahl Knoten	Anzahl Cluster	Laufzeit (s)
gr137	137	28	150
kroa150	150	30	150
d198	198	40	300
krob200	200	40	300
gr202	202	41	300
ts225	225	45	300
pr226	226	46	300
gil262	262	53	450
pr264	264	54	450
pr299	299	60	450
lin318	318	64	600
rd400	400	80	600
fl417	417	84	600
gr431	431	87	600
pr439	439	88	600
pcb442	442	89	600

zuschließen. Dazu wurden Tests durchgeführt, deren Ergebnisse in den folgenden Abschnitten dargestellt werden. Danach werden die eigentlichen Tests mit den jeweiligen Konfigurationen durchgeführt und die Ergebnisse vorgestellt bzw. kommentiert.

3.2 Konvertierung von unten und randomisierte Konvertierung

Als Erstes wurde untersucht, ob es Unterschiede zwischen den beiden Konvertierungsarten gibt. Dafür werden hier zwei Varianten verwendet: Der Pop-Trie alleine und Pop- und Ghosh in einer Schleife. Es wird eine fixierte Laufzeit verwendet, die Laufzeit für die Instanzen ist in Tabelle 3.1 angegeben. Die Ergebnisse sind in Tabelle 3.2 zu finden. Diese ist in vier Spalten für die verschiedenen Konfigurationen unterteilt: *prüfer, unten* bedeutet, dass der Prüfer-Trie mit Konvertierung von unten verwendet wird, *prüfer, rand*, dass der Prüfer-Trie mit randomisierter Konvertierung verwendet wird, *pred, unten*, dass der Predecessor-Trie mit Konvertierung von unten verwendet wird und *pred, rand*, dass der Predecessor-Trie mit randomisierter Konvertierung verwendet wird. $\overline{C(T)}$ gibt den Durchschnittswert über alle Läufe an, der Wert darunter in Klammern die Standardabweichung, *Anzahl Gen* die Anzahl der Generationen und *Mem(MB)* den Speicherverbrauch in MB. Es zeigt sich, dass es bei den kleineren Instanzen nur geringfügige Unterschied gibt, der Prüfer-Trie mit dieser Methode kann die besten Ergebnisse erzielen. Bei den größeren Instanzen *PR299* und *PCB442* sind die Ergebnisse für die ran-

domisierte Konvertierung deutlich besser, bei den anderen größeren Instanzen sind die Ergebnisse nicht so eindeutig. Deswegen wird bei den weiteren Tests die randomisierte Konvertierungsmethode verwendet, auch wenn deren Speicherverbrauch ein Mehrfaches höher ist. Die Ergebnisse für die Version in der Schleife sind in Tabelle 3.3 zu finden. Die Beschriftung ist dieselbe wie in der vorigen Tabelle. Es ergibt sich ein ähnliches Bild. Die Unterschiede sind sehr klein, nur in der Instanz *LIN318* hat die randomisierte Konvertierung einen deutlichen Vorteil. Für die einfacheren Instanzen wird die beste endgültige Lösung immer von allen Versionen gefunden.

3.3 Zusammenspiel von Pop- und Ghosh-Trie

Der Pop- und der Ghosh-Trie können auch in Kombination verwendet werden. Es gibt einerseits die einfache Variante, die Lösung nacheinander in den Trie einzufügen, oder aber eine Schleife, in der die Lösung solange konvertiert wird, bis in beiden Tries eine neue Lösung gefunden wird. In Tabelle 3.2 sind die Ergebnisse für die Version ohne Schleife zu finden, in Tabelle 3.3 die für die Version mit Schleife.

Aus dem Vergleich ergibt sich, dass die Ergebnisse sind für die Verwendung der Schleife durchgehend besser sind, es wird somit für die weiteren Tests nur diese Methode verwendet. Der Speicherverbrauch ist aber nochmal deutlich höher als die Variante ohne Verwendung der Schleife.

3.4 Lokale Optimierung

Als nächstes wird verglichen, ob sich die lokale Optimierung positiv auswirkt. Dieser Test wurde mit ebenfalls einer fixierten Laufzeit durchgeführt. Diese sind in Tabelle 3.4 zu finden. Hier wird nur mehr die randomisierte Konvertierung verwendet. Die Spalten sind wie folgt beschriftet: *Prüfer-Trie* bedeutet, dass der Prüfer-Trie ohne lokale Optimierung verwendet wird, *Prüfer-Trie mit lok. Opt.*, dass der Prüfer-Trie mit lokaler Optimierung verwendet wird, *Pred-Trie* bedeutet, dass der Predecessor-Trie ohne lokale Optimierung verwendet wird, *Pred-Trie mit lok. Opt.*, dass der Predecessor-Trie mit lokaler Optimierung verwendet wird. Die restlichen Angaben, also der Unterspalten, sind unverändert.

Wie an den Ergebnissen zu sehen ist, bringt sie nur manchmal etwas bessere Ergebnisse, liefert teilweise schlechtere Ergebnisse und verringert aber die Anzahl der Generationen, die in derselben Zeit durchgeführt werden können, deutlich. Daher wird die lokale Optimierung im Folgenden nicht verwendet werden.

3.5 Andere Optionen

Es wurde zwischen dem Predecessor-Trie und dem Prüfer-Trie kein bedeutender Unterschied gefunden, daher werden im Folgenden beide Varianten verwendet. Die Mutation wurde auf 0.10 eingestellt, da sich damit die besten Resultate ergaben.

Tabelle 3.2: Vergleich von Konvertierung von unten und randomisierter Konvertierung, nur mit Pop-Trie

	prüfer, unten			prüfer, rand			pred, unten			pred, rand		
Instanz	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)
gr137	**329,0** (0,0)	50308,0	5,0	**329,0** (0,0)	48278,3	33,1	**329,0** (0,0)	74648,1	4,1	**329,0** (0,0)	75099,1	69,5
kroa150	**9815,0** (0,0)	48038,2	5,3	**9815,0** (0,0)	45724,0	26,7	**9815,0** (0,0)	68420,8	3,9	**9815,0** (0,0)	69076,1	80,0
d198	**7044,0** (0,0)	48641,0	11,1	7045,8 (3,7)	47221,0	75,5	7051,4 (6,9)	65999,2	3,3	7044,6 (2,3)	65719,9	139,8
krob200	**11244,0** (0,0)	50977,4	33,1	11245,7 (6,6)	48913,1	137,7	11247,0 (8,1)	75468,6	8,7	11246,7 (6,9)	74891,6	170,8
gr202	**242,0** (0,0)	49033,6	23,0	**242,0** (0,0)	46071,3	123,7	**242,0** (0,0)	65921,1	9,7	242,1 (0,3)	67529,4	168,2
ts225	62269,3 (4,7)	43895,7	86,3	**62268,4** (0,5)	40046,2	182,6	**62268,4** (0,6)	61393,7	40,2	62268,5 (0,5)	59021,4	203,2
pr226	**55515,0** (0,0)	36895,1	2,6	**55515,0** (0,0)	35465,2	45,3	**55515,0** (0,0)	53254,1	1,3	**55515,0** (0,0)	53040,3	174,8
gil262	942,4 (2,0)	44020,0	75,1	**942,0** (0,0)	40509,4	193,6	942,6 (2,3)	60077,2	45,9	**942,0** (0,0)	61466,9	278,7
pr264	21886,7 (2,1)	42865,8	39,2	**21886,0** (0,0)	39475,0	189,0	21886,9 (2,4)	58290,7	31,0	**21886,0** (0,0)	55910,6	252,6
pr299	20335,3 (22,1)	32254,7	33,5	20326,3 (13,7)	31203,0	215,2	20321,2 (13,2)	46630,6	18,9	**20318,1** (11,3)	45916,7	272,7
lin318	18524,6 (12,3)	40449,6	39,5	**18520,4** (15,9)	35749,6	196,5	18525,2 (14,5)	54921,0	29,0	18527,8 (13,9)	55591,5	379,2
rd400	**5945,4** (7,1)	26708,1	87,1	5947,4 (13,4)	24585,7	470,0	5945,9 (11,2)	35589,3	31,8	5946,5 (10,0)	34903,0	390,7
fl417	**7982,0** (0,0)	23557,4	9,1	**7982,0** (0,0)	23146,9	227,7	**7982,0** (0,0)	32782,1	7,6	**7982,0** (0,0)	32286,4	391,9
gr431	**1033,1** (0,5)	21751,6	57,0	**1033,1** (0,4)	20238,8	416,3	1033,3 (0,7)	30066,6	28,5	1033,3 (0,8)	29948,4	403,4
pr439	51801,2 (18,5)	20066,6	83,9	51801,3 (25,8)	19290,8	422,9	**51800,0** (20,9)	25324,3	47,3	51804,4 (22,2)	25214,1	363,6
pcb442	19635,7 (38,9)	21258,6	142,1	19629,4 (19,0)	20717,3	486,9	19633,8 (21,0)	27819,6	81,6	**19625,4** (18,9)	27951,7	439,1

Tabelle 3.3: Vergleich von Konvertierung von unten und randomisierter Konvertierung, Kombination von Pop- und Ghosh-Trie in der Schleife

	prüfer, unten			prüfer, rand			pred, unten			pred, rand		
Instanz	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)
gr137	**329,0** (0,0)	20096,4	38,8	**329,0** (0,0)	22105,7	116,6	**329,0** (0,0)	16172,8	26,0	**329,0** (0,0)	22454,2	154,0
kroa150	**9815,0** (0,0)	21361,2	48,2	**9815,0** (0,0)	24104,4	128,8	**9815,0** (0,0)	16533,1	27,1	**9815,0** (0,0)	22245,1	159,0
d198	**7044,0** (0,0)	20781,0	101,7	**7044,0** (0,0)	21136,4	311,9	**7044,0** (0,0)	13801,1	39,0	**7044,0** (0,0)	20473,0	286,0
krob200	**11244,0** (0,0)	31160,0	198,1	**11244,0** (0,0)	30516,2	415,0	**11244,0** (0,0)	26639,8	71,9	**11244,0** (0,0)	32017,9	353,9
gr202	**242,0** (0,0)	28461,1	176,1	**242,0** (0,0)	28120,4	331,3	**242,0** (0,0)	21803,1	121,0	**242,0** (0,0)	29401,9	315,8
ts225	62268,6 (0,6)	26873,6	167,5	**62268,4** (0,5)	22876,1	322,6	62268,5 (0,5)	18583,1	94,4	62268,5 (0,5)	24291,0	427,5
pr226	**55515,0** (0,0)	2919,4	11,0	**55515,0** (0,0)	2941,3	244,6	**55515,0** (0,0)	4502,9	5,8	**55515,0** (0,0)	4873,3	381,5
gil262	**942,0** (0,0)	30562,2	304,6	**942,0** (0,0)	28982,2	512,0	**942,0** (0,0)	23876,1	152,0	**942,0** (0,0)	29349,2	530,1
pr264	**21886,0** (0,0)	24853,8	206,4	**21886,0** (0,0)	26552,0	437,3	**21886,0** (0,0)	17230,0	115,8	**21886,0** (0,0)	22897,3	530,6
pr299	20318,2 (11,3)	22869,7	244,3	**20316,0** (0,0)	22668,2	443,8	20317,4 (7,7)	18890,0	115,8	20317,4 (7,7)	22964,5	500,3
lin318	18516,1 (11,9)	26236,5	264,6	18515,5 (10,1)	25362,4	532,4	18522,1 (12,4)	21188,3	172,1	**18506,1** (9,5)	24379,5	806,6
rd400	5939,1 (8,8)	18832,9	586,4	5940,5 (6,4)	18301,1	955,6	5940,5 (6,4)	16815,8	243,7	**5938,8** (6,7)	18581,9	792,4
fl417	**7982,0** (0,0)	2609,1	51,6	**7982,0** (0,0)	2553,0	775,1	**7982,0** (0,0)	3205,0	31,0	**7982,0** (0,0)	3468,0	852,3
gr431	1033,1 (0,5)	14443,8	338,5	**1033,0** (0,0)	14444,7	717,7	**1033,0** (0,0)	11254,2	142,9	**1033,0** (0,0)	13695,8	661,4
pr439	**51793,5** (9,6)	12592,2	339,3	**51791,0** (0,0)	13883,8	721,3	**51791,0** (0,0)	11269,6	188,4	**51791,0** (0,0)	12965,2	712,1
pcb442	19626,7 (18,1)	16446,1	579,6	19628,9 (21,1)	15867,2	835,9	**19624,3** (22,5)	15047,2	374,7	19626,8 (19,7)	16708,7	748,8

Tabelle 3.4: Vergleich von der Version mit und ohne lokale Optimierung, in der Schleife

	Prüfer-Trie			Prüfer-Trie mit lok. Opt.			Pred-Trie			Pred-Trie mit lok. Opt.		
Instanz	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)	$\overline{C(T)}$	Anzahl Gen	Mem(MB)
gr137	**329,0** (0,0)	22105,7	116,6	**329,0** (0,0)	21507,4	112,2	**329,0** (0,0)	22454,2	154,0	**329,0** (0,0)	21517,2	147,8
kroa150	**9815,0** (0,0)	24104,4	128,8	**9815,0** (0,0)	22626,8	120,8	**9815,0** (0,0)	22245,1	159,0	**9815,0** (0,0)	21291,6	150,6
d198	**7044,0** (0,0)	21136,4	311,9	**7044,0** (0,0)	19763,2	290,8	**7044,0** (0,0)	20473,0	286,0	**7044,0** (0,0)	19270,0	263,2
krob200	**11244,0** (0,0)	30516,2	415,0	**11244,0** (0,0)	27590,5	374,2	**11244,0** (0,0)	32017,9	353,9	**11244,0** (0,0)	29266,0	312,5
gr202	**242,0** (0,0)	28120,4	331,3	**242,0** (0,0)	25976,7	308,2	**242,0** (0,0)	29401,9	315,8	**242,0** (0,0)	27332,9	289,1
ts225	**62268,4** (0,5)	22876,1	322,6	**62268,4** (0,5)	20649,7	274,1	62268,5 (0,5)	24291,0	427,5	**62268,4** (0,5)	21777,6	352,8
pr226	**55515,0** (0,0)	2941,3	244,6	**55515,0** (0,0)	2893,3	234,4	**55515,0** (0,0)	4873,3	381,5	**55515,0** (0,0)	4710,5	371,8
gil262	**942,0** (0,0)	28982,2	512,0	**942,0** (0,0)	24645,2	432,2	**942,0** (0,0)	29349,2	530,1	**942,0** (0,0)	25171,2	449,4
pr264	**21886,0** (0,0)	26552,0	437,3	**21886,0** (0,0)	23092,3	376,5	**21886,0** (0,0)	22897,3	530,6	**21886,0** (0,0)	20428,9	472,1
pr299	**20316,0** (0,0)	22668,2	443,8	20316,7 (3,8)	18025,4	348,2	20317,4 (7,7)	22964,5	500,3	20318,1 (11,3)	18077,3	384,3
lin318	18515,5 (10,1)	25362,4	532,4	18514,3 (12,7)	21603,4	452,6	**18506,1** (9,5)	24379,5	806,6	18509,0 (9,3)	21569,5	686,8
rd400	5940,5 (6,4)	18301,1	955,6	5942,4 (5,9)	14337,3	733,7	5938,8 (6,7)	18581,9	792,4	**5937,4** (6,1)	14374,8	592,5
fl417	**7982,0** (0,0)	2553,0	775,1	**7982,0** (0,0)	2494,4	745,4	**7982,0** (0,0)	3468,0	852,3	**7982,0** (0,0)	3315,3	809,3
gr431	**1033,0** (0,0)	14444,7	717,7	1033,1 (0,5)	10151,1	493,4	**1033,0** (0,0)	13695,8	661,4	**1033,0** (0,0)	10187,5	466,4
pr439	**51791,0** (0,0)	13883,8	721,3	**51791,0** (0,0)	9502,6	471,6	**51791,0** (0,0)	12965,2	712,1	**51791,0** (0,0)	9279,5	479,2
pcb442	19628,9 (21,1)	15867,2	835,9	19629,5 (21,3)	11466,9	583,7	**19626,8** (19,7)	16708,7	748,8	19627,6 (26,8)	11637,1	503,0

3.6 Die verwendeten Konfigurationen

Es werden folgende Konfigurationen verglichen: Erstens der EA ohne Trie, um ein Bild von der grundsätzlichen Effizienz des Algorithmus zu bekommen. Zweitens der EA mit der Pop-Optimierung, um zu sehen, ob der Aufwand der Tries mit ihrem hohen Speicherverbrauch einen Vorteil gegenüber dieser simplen Variante hat. Drittens der EA mit Ghosh-Trie, viertens der EA mit Prüfer-Trie alleine sowie fünftens der EA mit Predecessor-Trie alleine, um die Ergebnise der Tries alleine zu sehen. Sechstens wird der EA mit einer Schleife von Prüfer-Trie und Ghosh-Trie verwendet, siebtens der EA mit einer Schleife von Predecessor-Trie und Ghosh-Trie. Mit diesen beiden Varianten sollten die besten Ergebnisse gefunden werden.

3.7 Ergebnisse für die verschiedenen Varianten mit fixer Anzahl von Generationen

Um den Aufwand der verschiedenen Verfahren zu vergleichen, werden die verschiedenen Varianten mit einer fixierten Anzahl von Generationen getestet. Dabei hat sich herausgestellt, dass es bei der Verwendung von kleinen Instanzen mit der Schleife von Ghosh- und Pop-Trie vorkommen kann, dass die Erzeugung einer Lösung, die für beide Tries neu ist, unverhältnismäßig viel Zeit in Anspruch nimmt. Daher wurde ein zusätzlicher Parameter eingebaut, der die maximale Anzahl von Konvertierungsversuchen in der Schleife begrenzt und dieser Wert auf 25 gesetzt, d.h. es wird höchstens 25 Mal versucht, eine Lösung zu konvertieren. Für den Test wurden 10000 Generationen verwendet. In Tabelle 3.6 sind die Ergebnisse zu finden. Die Beschriftung erfolgt im Wesentlichen analog zu den vorigen Tabellen; der einzige Unterschied ist, dass bei dieser Variante nicht die Anzahl der Generationen angegeben wird, sondern die Laufzeit in Sekunden. Diese wird in der Spalte mit der Überschrift *Zeit(s)* angegeben.

Den Ergebnissen kann man entnehmen, dass die Varianten mit beiden Tries in der Schleife die besten Ergebnisse erzielen, sie können gegenüber den Versionen ohne Schleife einen deutlichen Gewinn erzielen. Aber auch der Pop-Trie alleine kann gegenüber den einfachen EA und dem EA mit Ghosh-Trie deutliche Gewinne erzielen. Außerdem ergibt sich für die Instanzen *PR226* und *FL417* eine besonders lange Laufzeit, es können hier offenbar keine Lösungen gefunden werden, die für beide Tries neu sind. Das heißt, dass die Kanten zwischen den Clustern der Lösungen, die bei der Konvertierung im Ghosh-Archiv generiert werden, im Pop-Archiv bereits vorhanden sind und umgekehrt, dass für die neuen Lösungen im Pop-Archiv die entsprechende Knotenbelegung bereits im Ghosh-Archiv vorhanden ist. Was die Laufzeit angeht, ist zu sehen, dass die Pop-Tries eine längere Laufzeit aufweisen als der Ghosh-Trie. Außerdem hat der Prüfer-Trie eine längere Laufzeit als der Predecessor-Trie. Daraus kann man folgern, dass die Konvertierung der Lösung in die Prüfer-Kodierung und wieder zurück mehr Zeit in Anspruch nimmt, als das Suchen einer neuen Lösung bzw. die Reparatur beim Auftreten eines Zyklus. Der Version mit den beiden Tries in der Schleife nimmt ebenfalls deutlich mehr Zeit in Anspruch als die einfachere Version. In dieser Variante gibt bei der Laufzeit zwischen der Version mit

Tabelle 3.5: Vergleich der verschiedenen Varianten mit 10000 Generationen

	EA		Pop-opt		Ghosh-Trie		Prüfer-Trie			Pred-Trie			Ghosh,Prüfer-Trie			Ghosh,Pred-Trie		
Instanz	$\overline{C(T)}$	Zeit(s)	$\overline{C(T)}$	Zeit(s)	$\overline{C(T)}$	Zeit(s)	$\overline{C(T)}$	Zeit(s)	Mem(MB)	$\overline{C(T)}$	Zeit(s)	Mem(MB)	$\overline{C(T)}$	Zeit(s)	Mem(MB)	$\overline{C(T)}$	Zeit(s)	Mem(MB)
gr137	329,2	16,4	**329,0**	19,0	329,3	20,1	**329,0**	28,0	11,8	**329,0**	19,9	11,7	**329,0**	58,1	48,5	**329,0**	55,0	59,2
	(0,4)		(0,0)		(0,5)		(0,0)			(0,0)			(0,0)			(0,0)		
kroa150	9817,8	17,5	**9815,0**	21,6	9822,0	22,1	**9815,0**	30,7	12,4	**9815,0**	21,7	14,5	**9815,0**	58,6	50,7	**9815,0**	59,5	64,5
	(12,9)		(0,0)		(21,4)		(0,0)			(0,0)			(0,0)			(0,0)		
d198	7057,4	37,3	7050,6	44,3	7057,2	47,3	7047,6	61,5	35,0	7048,2	45,0	25,2	**7044,0**	126,5	140,1	**7044,0**	126,3	123,9
	(9,0)		(4,6)		(8,9)		(4,5)			(4,6)			(0,0)			(0,0)		
krob200	11274,9	32,4	11250,6	39,8	11262,6	40,8	11248,6	57,5	48,4	11245,8	41,1	28,6	**11244,0**	86,7	126,3	**11244,0**	84,5	99,5
	(43,8)		(8,2)		(24,7)		(8,7)			(6,2)			(0,0)			(0,0)		
gr202	242,3	37,0	**242,0**	43,7	242,2	46,2	242,1	61,2	39,8	242,1	44,3	29,7	**242,0**	100,2	115,4	**242,0**	88,1	96,3
	(0,5)		(0,0)		(0,4)		(0,3)			(0,3)			(0,0)			(0,0)		
ts225	62284,6	42,3	62268,5	48,1	62297,5	48,3	**62268,3**	68,2	60,1	62270,0	49,0	40,0	62268,6	110,1	129,9	**62268,4**	105,4	153,4
	(39,3)		(0,6)		(35,2)		(0,5)			(8,7)			(0,5)			(0,5)		
pr226	**55515,0**	47,3	**55515,0**	56,5	**55515,0**	63,8	**55515,0**	80,3	27,2	**55515,0**	56,4	38,4	**55515,0**	1140,0	650,5	**55515,0**	677,8	772,4
	(0,0)		(0,0)		(0,0)		(0,0)			(0,0)			(0,0)			(0,0)		
gil262	951,6	61,0	942,2	71,9	944,9	74,7	942,4	104,3	71,2	943,1	72,4	55,0	**942,0**	141,5	163,4	**942,0**	136,7	165,5
	(4,8)		(1,1)		(4,0)		(2,0)			(3,4)			(0,0)			(0,0)		
pr264	21898,6	66,3	21887,6	76,5	21897,6	82,2	21888,8	108,8	64,7	**21886,0**	78,6	53,8	**21886,0**	157,2	154,0	**21886,0**	166,5	206,1
	(12,0)		(3,0)		(19,0)		(3,5)			(0,0)			(0,0)			(0,0)		
pr299	20347,2	85,8	20337,2	99,3	20346,2	107,1	20331,9	136,2	78,7	20326,0	98,8	67,9	**20317,4**	180,0	180,1	20319,5	179,5	199,1
	(27,0)		(24,7)		(31,6)		(21,9)			(18,8)			(5,3)			(13,5)		
lin318	18547,8	91,2	18536,9	109,8	18538,7	118,8	18528,2	159,5	81,4	18524,2	108,0	81,0	**18517,4**	218,1	194,3	18517,6	220,8	294,0
	(21,7)		(57,3)		(20,5)		(32,8)			(11,4)			(8,8)			(10,2)		
rd400	5959,9	139,0	5946,9	164,6	5961,4	174,7	5946,7	228,2	212,8	5942,6	165,0	130,8	5940,9	319,9	497,0	**5940,3**	305,3	414,2
	(13,4)		(12,0)		(20,6)		(12,8)			(9,1)			(6,0)			(6,1)		
fl417	**7982,0**	159,3	**7982,0**	182,1	**7982,0**	201,1	**7982,0**	264,7	183,5	**7982,0**	194,9	132,7	**7982,0**	2924,0	3172,4	**7982,0**	1929,5	2508,5
	(0,0)		(0,0)		(0,0)		(0,0)			(0,0)			(0,0)			(0,0)		
gr431	1034,4	168,2	1033,7	206,9	1034,2	216,2	1033,3	273,9	213,6	1033,2	205,7	145,6	**1033,0**	401,4	483,6	**1033,0**	419,7	453,9
	(1,1)		(0,9)		(1,3)		(0,8)			(0,5)			(0,0)			(0,0)		
pr439	51935,1	208,0	51804,6	237,1	51930,7	256,9	51805,0	312,5	225,7	51802,4	230,0	156,5	**51791,0**	432,9	495,1	**51791,0**	460,8	524,0
	(72,5)		(20,5)		(60,4)		(20,1)			(17,4)			(0,0)			(0,0)		
pcb442	19722,8	183,3	19641,6	212,6	19671,7	225,8	19635,7	283,9	249,5	19632,5	215,0	172,0	19630,8	350,9	498,9	**19629,3**	339,8	411,8
	(75,7)		(25,8)		(51,3)		(23,8)			(24,9)			(20,7)			(22,5)		

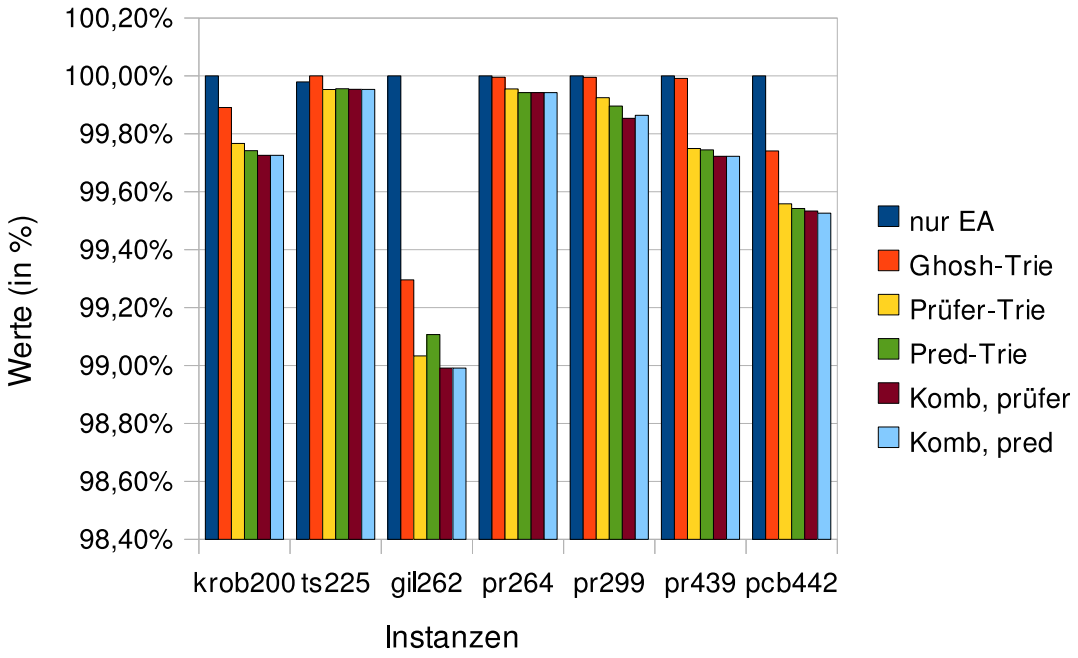

Abbildung 3.1: Ergebnisse für einige Instanzen bei Verwendung einer fixen Anzahl von Generationen

dem Predecessor-Trie und der mit dem Prüfer-Trie keine signifikanten Unterschiede. Dies lässt darauf schließen, dass der Prüfer-Trie aufgrund der niedrigeren Lokalität schneller eine für beide Tries neue Lösung findet, was den Zeitverlust aufgrund der Konvertierung in die Prüfer-Darstellung kompensiert. Zur Illustration wird das Ergebnis auch grafisch in einem Balkendiagramm in Abbildung 3.1 dargestellt. Es wurden nicht alle Instanzen verwendet, um die Grafik nicht zu überfrachten. Um die Werte für verschiedene Instanzen vergleichen zu können, werden die Ergebnisse der verschiedenen EA-Varianten relativ zueinander angegeben. Die Variante mit den schlechtesten Lösungswert entspricht dabei 100%, die Ergebnisse der anderen Varianten werden dabei in Relation zu diesem Wert gesetzt.

Was den Speicherverbrauch angeht, so ist zu sehen, dass bei der Varianten mit dem Pop-Archiv allein das Prüfer-Archiv mehr Speicherverbrauch hat. Da die Anzahl der Generationen gleich ist und in beiden Archiven somit gleich viele Lösungen gespeichert sind, kann daraus geschlossen werden, dass das Prüfer-Archiv aufgrund der geringeren Lokalität mehr Speicher verbraucht. Da die Lösungen, die der EA generiert, bei in der Prüfer-Darstellung schon bei kleineren Veränderungen stark auswirken können, sind die gemeinsamen Präfixe der Lösungen kleiner, was den größeren Speicherverbrauch erklärt. Bei Verwendung beider Archive hat das Prüfer-Archiv bei manchen Instanzen einen geringeren Speicherverbrauch. Dies kann damit erklärt werden, dass die geringere Lokalität

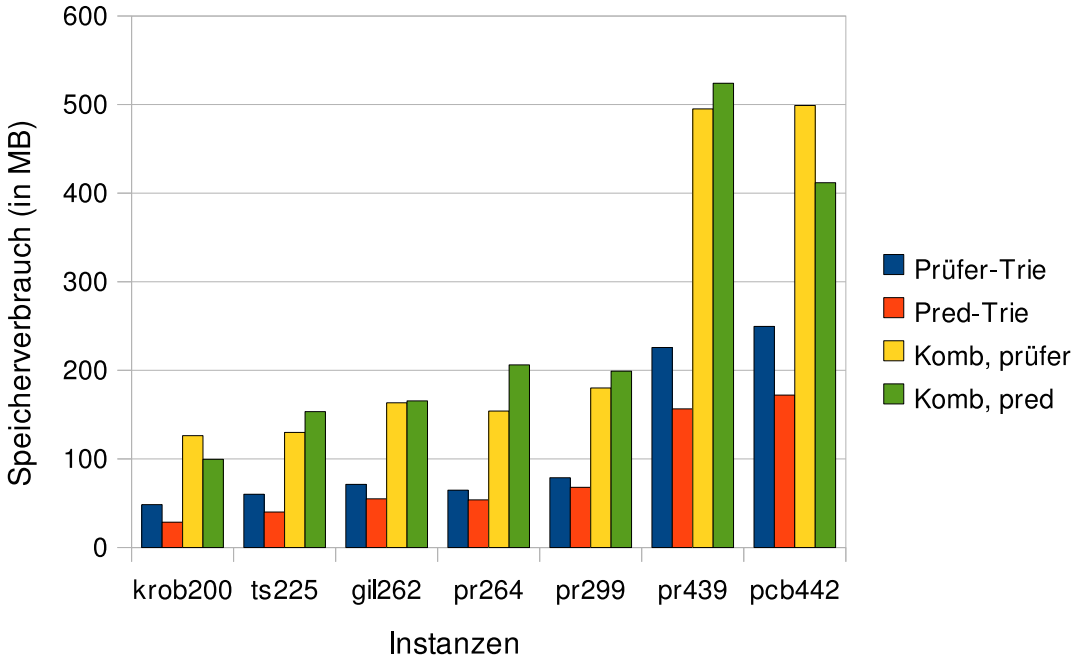

Abbildung 3.2: Speicherverbrauch für einige Instanzen bei Verwendung einer fixen Anzahl von Generationen

des Prüfer-Tries es ermöglicht, schneller eine Lösung zu finden, die für beide Archive neu ist, was auch im Abschnitt über die Anzahl der Konvertierungen für beide Archive gesehen werden kann. Zur besseren Übersicht wird der Speicherverbrauch in Abbildung 3.2 in einem Balkendiagramm dargestellt.

3.8 Ergebnisse für die verschiedenen Varianten mit fixer Laufzeit

Hier werden die verschiedenen Varianten mit einer fixen Laufzeit getestet. Die Ergebnisse sind in Tabelle 3.6 dargestellt. Die Beschriftung erfolgt analog zu den vorigen Tabellen, die ebenfalls eine fixierte Laufzeit verwenden. Aus Platzgründen wird hier die Spalte, die die Anzahl der Generationen angibt, mit *Gen* beschriftet. Die absoluten Werte kann man jedoch nicht direkt vergleichen mit denen aus dem vorigen Abschnitt vergleichen, da die Laufzeit hier höher ist als bei den Tests vorher.

Es ergibt sich dasselbe Bild: Die Versionen mit den beiden Tries in der Schleife sind den Versionen mit nur einem Trie überlegen, wobei der Pop-Trie ebenfalls deutlich besser als der Ghosh-Trie abschneidet. Der EA mit Pop-Optimierung erzielt dabei ungefähr die Werte wie der EA mit Pop-Trie, ist aber besser als die Version nur mit Ghosh-Trie, ist aber dem EA mit beiden Tries deutlich unterlegen.

Auch hier ist wieder zu erkennen, dass die Anzahl der Generationen bei der Verwendung beider Tries bei den Instanzen *GR226* und *FL417*, deutlich niedriger ist, weil diese Instanzen offenbar zu einfach sind. Für diese Instanzen hat nämlich auch der einfache EA dieselbe Lösung wie die anderen Varianten gefunden.

Die Ergebnisse für einige der Instanzen werden grafisch in Abbildung 3.3 dargestellt. Dieses ist analog zum Diagramm aus dem vorigen Abschnitt. In Bezug auf den Speicherverbrauch kann gesagt werden, dass das Predecessor-Archiv bei manchen Instanzen - im Gegensatz zu den Tests mit fixer Anzahl von Generationen - hier mehr Speicherplatz verbraucht als das Prüfer-Archiv, da diese in der selben Zeit mehr Generationen durchläuft und somit auch mehr Lösungen im Archiv zu speichern hat. Das entsprechende Diagramm ist in Abbildung 3.4 zu sehen.

3.9 Gelöschte Knoten im Trie

Der Trie inkludiert die Funktion, vollständige Knoten, also Knoten, deren Referenzen alle *Vollständig* sind, zu löschen. Jedoch bringt dies keine relevante Speicherplatzersparnis, da bei Verwendung der randomisierten Konvertierungsmethode die Anzahl der gelöschten Knoten bei 0-300 liegt, was einer Ersparnis von wenigen KB entspricht. Wie zu erwarten war, ist die Anzahl der gelöschten Knoten bei der Verwendung von der Konvertierung von unten etwas größer, jedoch bringt dies auch keine relevante Speicherersparnis. Dieser Wert ist bei ungefähr 1000.

3.10 Anzahl der Konvertierungen im Predecessor-Trie

Tabelle 3.7 stellt die Anzahl der notwendigen Konvertierungen bei Verwendung des kombinierten Lösungsarchivs dar, d.h. es wird die Anzahl der Konvertierungen angegeben, bis eine für beide Archive neue Lösung gefunden werden kann. Für diesen Test wird der Predecessor-Trie verwendet, der Test aus dem vorigen Abschnitt mit der fixen Laufzeit dient als Basis für diese Tabelle. Für eine bessere Darstellung wurden die Werte zusammengefasst. Die Spalten beschreiben die Anzahl der Konvertierungen: 0 (keine Konvertierung), 1-5, 6-10, 11-15, 16-20, 21-24, 25 ist der Maximalwert und zeigt an, wieoft keine neue Lösung gefunden werden konnte. Wie zu sehen ist, sind für die kleineren Instanzen durchschnittlich mehr Konvertierungen erforderlich. Bei den größeren Instanzen kann in mehr als 90 % der Fälle nach 10 Versuchen eine Lösung gefunden werden, die in beiden Tries neu ist. Für die zwei Instanzen *PR226* und *FL417* wird die maximale Zahl an Konvertierungsversuchen für den Großteil der Generationen erreicht.

Anzahl der Reparaturen im Predecessor-Trie

Da der Predecessor-Trie bei der Konvertierung auch ungültige Lösungen generieren kann, wenn ein Zyklus auftritt, müssen diese korrigiert werden. Allerdings kann die Lösung nach dieser Reparatur wieder ein Duplikat werden, und die Lösung muss erneut konvertiert

Tabelle 3.6: Vergleich der verschiedenen Varianten mit fixierter Laufzeit

		EA		Pop-opt		Ghosh-Trie		Prüfer-Trie			Pred-Trie			Ghosh,Prüfer-Trie			Ghosh,Pred-Trie		
Instanz	Zeit	$\overline{C(T)}$	Gen	$\overline{C(T)}$	Gen	$\overline{C(T)}$	Gen	$\overline{C(T)}$	Gen	Mem(MB)	$\overline{C(T)}$	Gen	Mem(MB)	$\overline{C(T)}$	Gen	Mem(MB)	$\overline{C(T)}$	Gen	Mem(MB)
gr137	150	329,4 (0,5)	92088,6	**329,0** (0,0)	73559,2	329,3 (0,5)	68285,5	**329,0** (0,0)	51786,4	34,8	**329,0** (0,0)	74217,7	69,0	**329,0** (0,0)	21920,6	117,8	**329,0** (0,0)	22492,2	155,5
kroa150	150	9830,6 (31,4)	85207,8	**9815,0** (0,0)	68318,5	9831,3 (30,1)	60032,8	**9815,0** (0,0)	47376,6	27,4	**9815,0** (0,0)	68786,3	79,5	**9815,0** (0,0)	24363,6	128,8	**9815,0** (0,0)	22307,1	158,9
d198	300	7055,1 (8,7)	80573,6	**7044,0** (0,0)	67885,8	7059,6 (9,0)	61261,6	7047,3 (4,4)	45053,6	72,1	7044,6 (2,3)	66463,3	141,1	**7044,0** (0,0)	21085,4	308,7	**7044,0** (0,0)	20725,3	287,6
krob200	300	11275,0 (45,6)	88482,8	11244,7 (3,7)	73274,6	11248,9 (7,5)	70410,6	11244,7 (3,7)	50637,6	142,9	**11244,0** (0,0)	74346,1	169,3	**11244,0** (0,0)	30027,5	411,6	**11244,0** (0,0)	31391,4	354,9
gr202	300	242,1 (0,3)	82079,8	**242,0** (0,0)	69617,2	242,2 (0,4)	60553,7	**242,0** (0,0)	46119,8	117,6	**242,0** (0,2)	64455,8	161,5	**242,0** (0,0)	28987,4	340,6	**242,0** (0,0)	29320,0	320,8
ts225	300	62290,8 (40,4)	75250,6	62268,6 (0,5)	63075,2	62299,1 (50,9)	58674,1	62268,5 (0,6)	39070,8	187,1	62268,6 (0,5)	61816,3	212,3	**62268,4** (0,5)	25579,2	331,3	**62268,4** (0,5)	25097,3	415,4
pr226	300	**55515,0** (0,0)	61452,2	**55515,0** (0,0)	52228,4	**55515,0** (0,0)	48535,1	**55515,0** (0,0)	36368,6	45,8	**55515,0** (0,0)	52915,2	174,6	**55515,0** (0,0)	2833,4	229,4	**55515,0** (0,0)	4902,5	385,2
gil262	450	945,5 (4,0)	74319,0	**942,0** (0,0)	63085,7	945,0 (3,7)	55321,4	942,2 (1,3)	41070,2	195,9	942,4 (2,0)	58507,7	265,7	**942,0** (0,0)	29062,2	515,1	**942,0** (0,0)	29515,5	533,2
pr264	450	21893,2 (7,7)	70628,1	21886,2 (1,3)	59856,0	21898,4 (20,9)	53267,6	**21886,0** (0,0)	38061,3	186,2	**21886,0** (0,0)	58053,8	262,5	**21886,0** (0,0)	26353,8	433,7	**21886,0** (0,0)	22453,1	523,9
pr299	450	20352,1 (37,4)	53161,7	20325,2 (15,1)	44973,2	20349,7 (24,9)	42395,6	20325,8 (13,2)	31146,3	214,5	20318,5 (11,3)	46510,0	277,2	20325,0 (21,5)	21871,0	426,8	**20318,1** (11,3)	22770,8	501,3
lin318	600	18545,9 (29,2)	65893,4	18520,9 (15,0)	56310,4	18547,3 (25,6)	47832,0	18523,6 (13,2)	37039,1	206,2	18525,8 (12,4)	55163,1	376,4	18513,1 (12,1)	25722,4	544,1	**18511,0** (10,8)	24639,0	805,1
rd400	600	5953,0 (15,4)	43338,2	5947,6 (10,9)	37084,8	5959,4 (20,2)	33302,7	5945,3 (7,6)	24997,3	472,8	5946,4 (10,8)	34719,4	388,4	**5938,0** (6,3)	18656,6	970,1	5940,2 (6,5)	17891,4	772,1
fl417	600	**7982,0** (0,0)	39010,8	**7982,0** (0,0)	32938,4	**7982,0** (0,0)	29715,1	**7982,0** (0,0)	22032,8	224,4	**7982,0** (0,0)	32678,6	395,9	**7982,0** (0,0)	2539,4	765,8	**7982,0** (0,0)	3573,4	876,5
gr431	600	1034,1 (1,4)	34104,4	1033,4 (0,7)	28988,0	1033,4 (0,9)	26429,4	1033,1 (0,5)	21057,7	433,0	1033,3 (0,7)	29858,9	403,3	1033,1 (0,4)	14623,2	724,3	**1033,0** (0,0)	13591,0	649,5
pr439	600	51921,4 (60,7)	30605,0	51798,6 (15,1)	26509,3	51888,5 (56,3)	23722,0	51795,8 (11,6)	19630,4	430,6	51810,5 (26,5)	24953,9	363,7	**51791,0** (0,0)	13930,6	720,6	**51791,0** (0,0)	12063,3	660,5
pcb442	600	19717,0 (59,5)	33185,1	19633,6 (29,0)	28486,4	19708,1 (70,2)	25830,4	19634,8 (21,0)	19847,6	469,6	19632,6 (21,1)	28147,9	439,7	**19620,9** (16,6)	15681,6	829,3	19623,7 (15,9)	16547,3	754,8

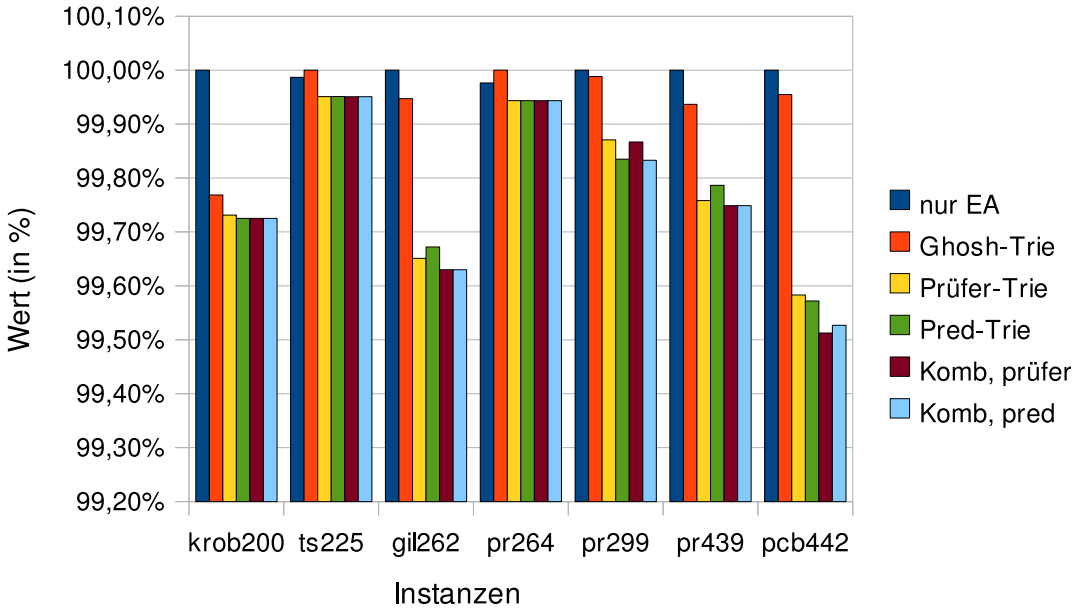

Abbildung 3.3: Ergebnisse für einige Instanzen bei Verwendung fixer Laufzeit

werden. Dies kann auch mehrmals hintereinander auftreten. Tabelle 3.8 zeigt, wie oft die Lösungen repariert mussten, bzw. wie oft sie mehrmals repariert werden mussten.

Wie man sieht, kann für die meisten Instanzen fast immer nach spätestens zwei Konvertierungen eine neue Lösung gefunden werden, die Anzahl der notwendigen Aufrufe hält sich also im Rahmen. Für kleinere Instanzen sind Reparaturen öfter notwendig als bei den größeren.

3.11 Anzahl der Konvertierungen für den Prüfer-Trie

Als nächstes wird angegeben, wieviele Konvertierungen erforderlich sind, wenn das kombinierte Lösungsarchiv mit dem Prüfer-Trie verwendet wird. Die Beschriftung der Tabelle ist analog zu 3.7. Wie man Tabelle 3.9 entnehmen kann, sind im Vegleich zum Predecessor-Trie durchschnittlich deutlich weniger Konvertierungen erforderlich. Dies liegt wohl daran, dass beim Prüfer-Trie die Lokalität kleiner ist, und schneller eine für beide Tries neue Lösung gefunden werden kann. Wie jedoch im Abschnitt zu den Tests festgestellt wurde, ist die Laufzeit der Version mit dem Prüfer-Trie dennoch im Durchschnitt länger als die mit Predecessor-Trie, da das Hin- und Zurückkonvertieren der Lösungen in die Prüfer-Darstellung selber mehr Aufwand verursacht.

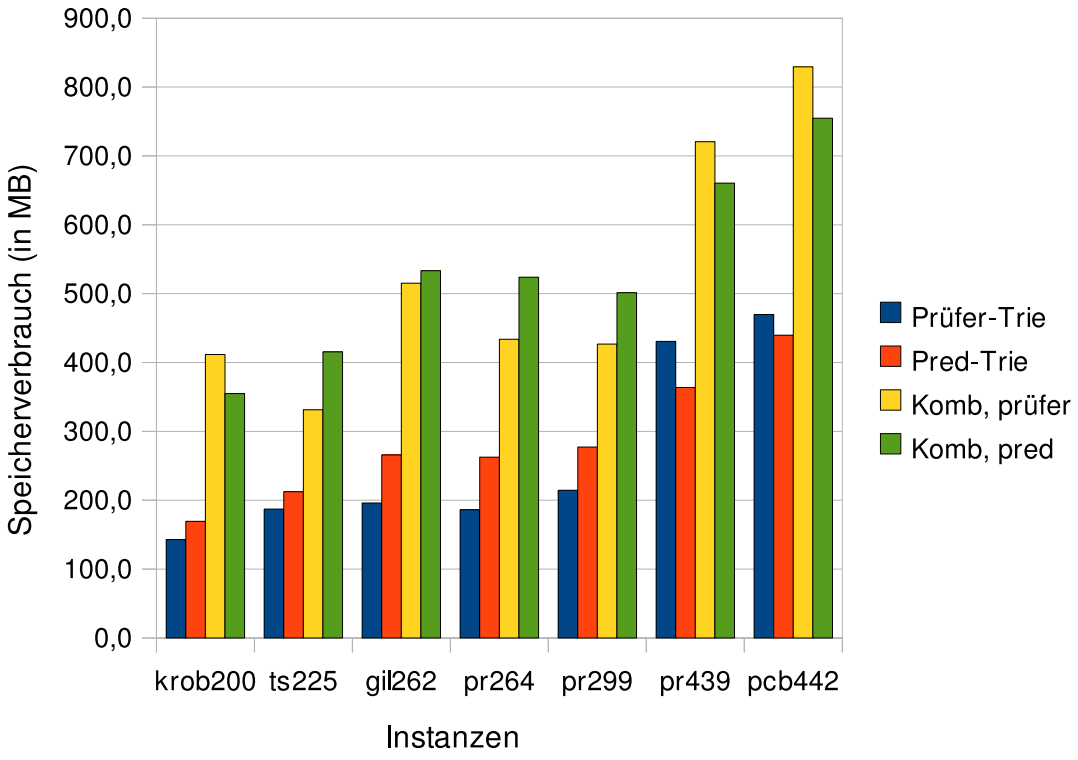

Abbildung 3.4: Speicherverbrauch für einige Instanzen bei Verwendung einer fixen Anzahl von Generationen

3.12 Test mit längerer Laufzeit

Um den Algorithmus bzw. dessen Verhalten für längere Laufzeiten zu überprüfen, wurde auch ein Test mit einer Laufzeit von 30000 Generationen durchgeführt. Die Ergebnisse sind in Tabelle 3.12 zu finden, die Beschriftung ist analog zu Tabelle 3.6. Die Instanzen *PR226* und *FL417* wurden hier ausgelassen.

Die Variante mit der randomisierten Konvertierungsmethode ist meistens besser, und manchmal ist der Unterschied sehr deutlich, wie bei den Instanzen *PCB442* oder *LIN318*. Bei den anderen Varianten ergibt sich allerdings kein größerer Unterschied, teilweise sind auch die Versionen mit der Konvertierungsmethode von unten geringfügig besser. Bei *RD400* und *PR299* sind die Resultate etwas besser. Es zeigt sich hier auch, dass der Predecessor-Trie bei den größeren Instanzen Vorteile hat, was sich in den Tests mit kürzerer Laufzeit nicht so gezeigt hat.

Bei der Version mit dem Predecessor-Trie ist zu beachten, dass die randomisierte Konvertierung deutlich schneller ist als die andere Variante. Dies liegt wohl daran, dass die Lösung bei der Konvertierung von unten öfter weiter unten verändert wird, was zur Folge hat, dass es länger dauert, bis eine Lösung gefunden werden kann, die in beiden Tries

Tabelle 3.7: Anzahl der Konvertierungen für den Predecessor-Trie

Instanz	0	1-5	6-10	11-15	16-20	21-24	25
gr137	949,5	12333,1	5520,7	2182,2	867,3	313,0	326,4
kroa150	1221,2	12646,4	5621,2	1876,9	627,5	179,9	133,8
d198	703,5	11139,2	5692,4	2085,1	719,1	217,8	167,9
krob200	2369,0	21710,2	5938,8	1127,0	204,0	31,9	10,6
gr202	1997,3	20932,8	5201,7	969,8	176,7	30,0	11,4
ts225	8820,8	10210,2	3656,8	1413,5	571,2	201,4	223,4
pr226	87,0	95,1	93,1	98,3	98,0	80,1	4351,3
gil262	4543,8	19975,4	4344,2	577,4	67,1	6,2	1,5
pr264	3057,0	13056,6	4622,2	1266,7	331,4	79,8	39,2
pr299	2368,2	16734,7	3127,5	461,6	66,6	9,9	2,5
lin318	2494,5	16698,1	4394,0	853,8	163,7	25,4	9,5
rd400	2037,8	13241,2	2241,0	320,2	44,0	6,0	1,2
fl417	151,5	242,2	221,6	229,4	231,6	176,3	2320,7
gr431	1551,4	9403,0	2172,7	384,5	66,5	9,7	3,1
pr439	2329,6	7558,0	1694,6	374,9	84,6	14,2	7,4
pcb442	3460,2	11977,6	1034,0	70,5	4,8	0,1	0,1

Tabelle 3.8: Anzahl der Reparaturen für Schleife von Predecessor-Trie und randomisierter Konvertierung

Instanz	0	1	2	3	4
gr137	12690,6	34805,6	5860,0	157,5	0,8
kroa150	12209,5	32382,8	6336,7	284,7	0,6
d198	11642,8	36887,1	6838,1	90,0	0,2
krob200	11674,5	23924,3	938,8	0,4	0,0
gr202	14734,1	20028,3	362,5	0,0	0,0
ts225	10846,5	26682,9	2613,7	26,1	0,0
pr226	1824,3	111552,0	113169,6	23798,8	151,3
gil262	12628,0	19980,5	172,1	0,0	0,0
pr264	11632,3	28576,0	978,7	0,1	0,0
pr299	11241,6	21486,8	529,2	0,1	0,0
lin318	11509,2	26070,7	177,7	0,0	0,0
rd400	11634,9	21545,9	101,3	0,0	0,0
fl417	4946,2	140589,0	78751,8	2776,1	0,2
gr431	13087,9	23309,2	162,1	0,0	0,0
pr439	10943,0	25309,7	22,1	0,0	0,0
pcb442	11699,0	12365,6	4,2	0,0	0,0

Tabelle 3.9: Anzahl der Konvertierungen für den Prüfer-Trie

Instanz	0	1-5	6-10	11-15	16-20	21-24	25
gr137	1079,3	18153,7	2320,2	311,8	46,5	10,7	2,4
kroa150	1331,8	22070,2	923,1	36,9	1,4	0,1	0,0
d198	825,5	18348,7	1769,4	131,7	9,3	1,0	0,1
krob200	2625,7	27017,2	380,4	4,1	0,0	0,0	0,0
gr202	2225,6	26072,3	672,7	16,3	0,7	0,1	0,0
ts225	9987,6	14772,4	767,7	48,3	2,9	0,6	0,0
pr226	107,2	198,7	142,5	129,2	121,2	113,9	2044,7
gil262	4878,7	24016,0	166,6	0,9	0,0	0,0	0,0
pr264	4073,4	22043,9	233,0	3,3	0,0	0,0	0,0
pr299	2639,8	19147,2	83,5	0,4	0,0	0,0	0,0
lin318	2488,0	23017,0	215,8	1,6	0,0	0,0	0,0
rd400	2326,1	16272,3	58,0	0,1	0,0	0,0	0,0
fl417	186,3	541,9	295,1	233,7	193,9	168,5	955,3
gr431	1863,3	12607,9	149,6	2,2	0,0	0,0	0,0
pr439	2798,8	11048,7	82,4	0,5	0,0	0,0	0,0
pcb442	3659,7	11998,4	23,3	0,1	0,0	0,0	0,0

neu ist. Der Speicherverbrauch ist allerdings weiter deutlich geringer. Bei der Version mit dem Prüfer-Trie kann dieser Effekt nicht beobachtet werden, was darin liegt, dass die Lokalität in diesem Trie geringer ist und somit die Veränderung der Lösung weiter unten nach der Konvertierung in der Predecessor-Darstellung größer ist.

Im Vergleich zu dem Lauf mit nur 10000 Generationen zeigt sich, dass die Lösungen bei den größeren Instanzen meistens geringfügig verbessert werden kann. Bei den kleineren Instanzen zeigt sich kein Unterschied, da dieselbe Lösung von allen Varianten durchgehend gefunden wird.

Tabelle 3.10: Vergleich der verschiedenen Varianten mit 30000 Generationen

	prüfer, unten			prüfer, rand			pred, unten			pred, rand		
Instanz	$\overline{C(T)}$	Zeit(s)	Mem(MB)	$\overline{C(T)}$	Zeit(s)	Mem(MB)	$\overline{C(T)}$	Zeit(s)	Mem(MB)	$\overline{C(T)}$	Zeit(s)	Mem(MB)
gr137	**329,0** (0,0)	256,6	55,2	**329,0** (0,0)	210,7	162,3	**329,0** (0,0)	293,0	50,3	**329,0** (0,0)	207,7	214,1
kroa150	**9815,0** (0,0)	215,2	66,2	**9815,0** (0,0)	188,9	161,5	**9815,0** (0,0)	276,9	49,3	**9815,0** (0,0)	209,2	216,9
d198	**7044,0** (0,0)	453,9	144,7	**7044,0** (0,0)	433,7	454,3	**7044,0** (0,0)	768,2	81,3	**7044,0** (0,0)	478,5	447,1
krob200	11244,7 (3,7)	282,0	189,8	**11244,0** (0,0)	285,2	400,1	11244,7 (3,7)	324,5	80,9	**11244,0** (0,0)	274,0	327,9
gr202	**242,0** (0,0)	315,0	183,2	**242,0** (0,0)	312,5	355,3	**242,0** (0,0)	433,8	158,3	**242,0** (0,0)	301,4	322,8
ts225	62268,4 (0,5)	372,7	183,3	62268,4 (0,5)	370,6	400,0	**62268,2** (0,4)	515,3	142,0	62268,4 (0,5)	401,1	581,4
gil262	**942,0** (0,0)	444,4	297,5	**942,0** (0,0)	474,6	534,3	**942,0** (0,0)	609,7	190,4	**942,0** (0,0)	453,1	539,7
pr264	**21886,0** (0,0)	550,7	248,7	**21886,0** (0,0)	513,7	497,9	**21886,0** (0,0)	853,5	199,9	**21886,0** (0,0)	602,4	728,5
pr299	20318,1 (11,3)	591,3	317,7	20318,8 (11,8)	615,9	596,4	20318,1 (11,3)	780,9	179,9	**20316,0** (0,0)	624,7	683,7
lin318	18518,8 (10,4)	706,2	299,3	18514,8 (9,2)	732,2	631,6	18516,7 (12,4)	869,5	251,7	**18504,0** (6,8)	739,8	1017,7
rd400	5942,6 (10,3)	1029,5	920,0	5938,5 (6,5)	1007,5	1600,5	5938,5 (6,2)	1168,3	421,8	**5937,5** (6,4)	1012,6	1314,4
gr431	**1033,0** (0,0)	1277,0	683,3	**1033,0** (0,0)	1319,6	1558,1	**1033,0** (0,0)	1786,9	364,9	**1033,0** (0,0)	1470,9	1553,0
pr439	**51791,0** (0,0)	1558,9	811,4	**51791,0** (0,0)	1453,9	1718,1	**51791,0** (0,0)	2002,8	481,6	**51791,0** (0,0)	1627,7	1898,4
pcb442	19630,1 (24,9)	1183,3	1061,7	19626,3 (21,8)	1181,6	1645,6	19630,1 (20,4)	1393,5	743,9	**19622,3** (19,5)	1111,0	1424,4

KAPITEL 4

Zusammenfassung

In der vorliegenden Arbeit wurde ein neues Lösungsarchiv für einen evolutionären Algorithmus für das Generalized Minimum Spanning Tree-Problem entwickelt. Ausgehend von der Arbeit von Wolf, in der er ein Archiv auf Basis der Ghosh-Darstellung erstellt hat, wurde hier ein Archiv auf Basis der Pop-Darstellung hinzugefügt.

Die Funktionsweise des Archivs besteht darin, dass es die Lösungen, die der evolutionäre Algorithmus generiert, speichert, und für bereits vorhandene Lösungen eine neue, unbesuchte Lösung generiert. Dabei wurden zwei verschiedene Methoden, die Lösungen umzuwandeln, implementiert. Die erste versucht, die Lösung möglichst weit unten im Trie zu verändern und verbraucht somit weniger Speicherplatz, erzielt aber auch schlechtere Lösungen als die zweite Variante, die die Lösung an einer zufälligen Stelle verändert.

Der Pop-Trie wurde mit zwei verschiedenen Kodierungen implementiert, der Predecessor- und der Prüfer-Kodierung. Zwischen diesen beiden Varianten ergab sich allerdings kein großer Unterschied in der Qualität der Lösungen.

Die verschiedenen Varianten wurden dann getestet. Der Pop-Trie kann bessere Ergebnisse als der Ghosh-Trie erzielen oder der evolutionäre Algorithmus alleine. Durch Kombinierung der beiden Archive konnten die Ergebnisse im Vergleich zu den anderen Varianten deutlich verbessert werden. Dies wurde dadurch erreicht, dass eine neue Lösung erst dann akzeptiert wird, wenn sie in beiden Archiven neu ist. Ist dies nicht der Fall, wird die Lösung solange konvertiert, bis eine solche gefunden werden kann. Allerdings weist diese Version eine längere Laufzeit sowie einen größeren Speicherverbrauch auf.

Literaturverzeichnis

[Fer01] Feremans, C.: *Generalized Spanning Trees and Extensions*, Universite Libre de Bruxelles, Diss., 2001

[Gho03] Ghosh, D.: Solving Medium to Large Sized Euclidean Generalized Minimum Spanning Tree Problems / Indian Institute of Management, Research and Publication Department, Ahmedabad, India. 2003. – Forschungsbericht

[HLR08] Hu, B. ; Leitner, M. ; Raidl, G.: 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

[Lei06] Leitner, M.: *Solving Two Generalized Network Design Problems with Exact and Heuristic Methods.*, Technische Universität Wien, Diplomarbeit, 2006

[MLT95] Myung, Y.S. ; Lee, C.H. ; Tcha, D.W.: On the Generalized Minimum Spanning Tree Problem. In: *NETWORKS* 26 (1995), S. 231–241

[OW02] Ottmann, T. ; Widmayer, P.: *Algorithmen und Datenstrukturen.* 4. Spektrum Akademischer Verlag, 2002

[PK94] Palmer, C.C. ; Kershenbaum, A.: Representing Trees in Genetic Algorithms. In: *Proceedings of the First IEEE Conference on Evolutionary Computation* (1994), S. 379–384

[Pop02] Pop, P.C.: *The Generalized Minimum Spanning Tree Problem*, University of Twente, Diss., 2002

[Pop05] Pop, P.C.: On Some Polynomial Solvable Cases Of the Generalized Minimum Spanning Tree Problem. In: *Proceedings of the International Conference on Theory and Application of Mathematics and Informatics ICTAMI 2005* (2005)

[RD00] Raidl, G.R. ; Drexel, C.: A Predecessor Coding in an Evolutionary Algorithm for the Capacitated Minimum Spanning Tree Problem. In: *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference* (2000), S. 309–316

[RJ03] Raidl, G.R. ; Julstrom, B.A.: Edge Sets: An Effective Evolutionary Coding Of Spanning Trees. In: *IEEE Transactions on Evolutionary Computation* 7 (2003), Nr. 3, S. 225–239

[SP94] Srinivas, M. ; Patnaik, L.M.: Genetic algorithms: A survey. In: *Computer* 27 (1994), Nr. 6, S. 17–26

[Sra09] Sramko, A.: *Enhancing a Genetic Algorithm by a Complete Solution Archive Based on a Trie Data Structure.*, Technische Universität Wien, Diplomarbeit, 2009

[Wol09] Wolf, M.: *Ein Lösungsarchiv-unterstützter evolutionärer Algorithmus für das Generalized Minimum Spanning Tree Problem*, Technische Universität Wien, Diplomarbeit, 2009

[Zau08] Zaubzer, S.: *A Complete Archive Genetic Algorithm for the Multidimensional Knapsack Problem.*, Technische Universität Wien, Diplomarbeit, 2008