FAKULTÄT FÜR !NFORMATIK

Ein Lösungsarchiv-unterstützter evolutionärer Algorithmus für das GMST-Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Markus Wolf
Matrikelnummer 0325879

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 07.07.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Markus Wolf
Hadikgasse 122/5, 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, 07.07.2009

Kurzfassung

Das Generalized Minimum Spanning Tree (GMST)-Problem ist ein Optimierungsproblem. Gegeben ist ein Graph, dessen Knoten in Cluster eingeteilt sind, und gesucht wird ein minimaler Spannbaum, wobei aus jedem Cluster genau ein Knoten gewählt werden muss.

In dieser Arbeit wird ein evolutionärer Algorithmus verwendet, um dieses Problem zu behandeln. Ein Problem von derartigen Algorithmen sind die häufig mehrfach auftretenden Lösungen. Die gleiche Lösung mehrmals zu evaluieren, kostet unnötig Zeit. Deshalb wird zur Verbesserung des Algorithmus ein Lösungsarchiv basierend auf einer Trie-Struktur verwendet.

Alle generierten Lösungen werden in dieses Archiv gespeichert. Stellt man dabei fest, dass die Lösung schon vorhanden ist, kann sie auf effiziente Weise in eine neue, garantiert noch nicht untersuchte Lösung umgewandelt werden. Auf diese Art können in der gleichen Laufzeit mehr unterschiedliche Lösungen untersucht werden.

Derartige Lösungsarchive wurden bislang nur bei Problemen angewendet, deren Lösungen als Bitstrings repräsentiert wurden. Das ist beim GMSTP nicht möglich, weswegen der Trie eine andere Struktur hat, die zusätzliche Überlegungen bezüglich des Speicherplatzes nötig macht. Unter anderem wird eine Variante vorgestellt, bei der mehrere Tries verwendet werden, von denen jeder nur einen Teil der Lösung enthält.

Es zeigt sich, dass mit Hilfe dieses Archivs bei vielen Instanzen bessere Lösungen gefunden werden können. Weiter verbessert werden können die Ergebnisse durch eine Optimierung, die auf einer alternativen Repräsentation von Lösungen basiert.

Abstract

The Generalized Minimum Spanning Tree (GMST) problem is an optimization problem. Given a graph whose node set is partitioned into clusters, the goal is to find a minimum spanning tree that contains exactly one node from each cluster.

In this work an evolutionary algorithm is used to solve the GMSTP. One problem with this kind of algorithms is the frequently appearing duplicate solutions. Evaluating the same solution multiple times wastes time while offering no new information. For this reason, a solution archive based on a trie structure is used to improve the algorithm.

Every newly generated solution is stored in this archive. If the solution is already present, it can be efficiently converted into one that is guaranteed to be new and not yet evaluated. This allows the algorithm to evaluate more unique solutions using the same runtime.

Until now, solution archives of this type have only been applied to problems whose solutions are represented as bit strings. Since this is not possible for the GMSTP, it leads to a different structure for the trie and requires additional considerations concerning memory usage. A variant using multiple tries is also introduced, where each individual trie stores only partial solutions.

The results show that this archive can lead to better solutions for many instances. Additionally, a different kind of optimization based on an alternate representation of solutions can further improve solutions.

Inhaltsverzeichnis

1 Einleitung

Bei dem Generalized Minimum Spanning Tree (GMST)-Problem handelt es sich um ein Optimierungsproblem, das auf dem klassischen Minimum Spanning Tree (MST)-Problem basiert.
Das MST-Problem ist wie folgt definiert:
Gegeben ist ein verbundener Graph G, dessen Kanten Gewichte zugewiesen sind. Gesucht ist ein minimaler Spannbaum. Hierbei handelt es sich um einen kreisfreien verbundenen Teilgraph von G, bei dem die Summe der Kantengewichte minimal ist. Dieses Problem ist einfach (d.h. in polynomieller Zeit) lösbar, z.B. mit dem Algorithmus von Kruskal [19].

1.1 Das GMST-Problem

Beim GMST-Problem sind die Knoten des Graphen G in Cluster partitioniert. Gesucht ist nun ein minimaler Spannbaum, der aus jedem Cluster genau einen Knoten enthält.
Eine formelle Definition des GMSTP lautet wie folgt [12]:

Gegeben sei ein vollständiger gewichteter Graph $G = (V, E, c)$ mit Knotenmenge V, Kantenmenge E und Kostenfunktion $c : E \rightarrow \mathbb{R}^+$, wobei V in paarweise disjunkte Teilmengen $V_1, V_2, \ldots, V_r$, genannt Cluster, mit jeweils $d_1, d_2, \ldots, d_r$ Knoten partitioniert ist. Eine Lösung für das GMST-Problem ist ein Subgraph $S = (P, T)$, wobei $P = \{p_1, p_2, \ldots p_r\}$ aus jedem Cluster genau einen Knoten enthält, d.h. $p_i \in V_i$, $1 \leq i \leq r$, und $T \subseteq P \times P \subseteq E$ ein Spannbaum ist. Gesucht ist eine Lösung mit minimalen Kosten $C(T)$, wobei die Kosten einer Lösung der Summe der Kosten aller Kanten des Spannbaums entsprechen.

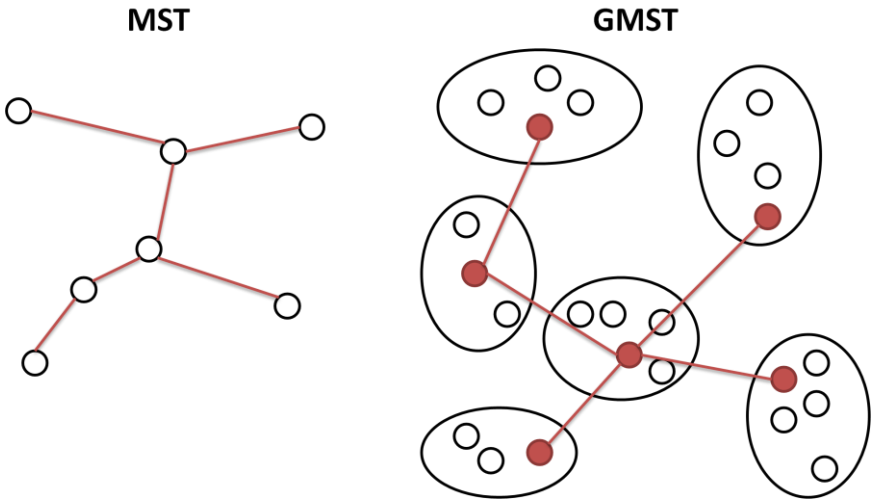

Abb. 1: Vergleich von MST und GMST (in rot jeweils die Lösung)

Durch die Erweiterung auf Cluster wird dieses Problem im allgemeinen Fall NP-schwierig, der Beweis ist in [3] zu finden.

Eine praktische Anwendung dieses Problems ist das Verbinden mehrerer lokaler Netzwerke. Hierbei entspricht ein Cluster jeweils einem Netzwerk, und der aus dem Cluster gewählte Knoten entspricht einem Backbone. Diese werden dann miteinander auf möglichst effiziente Weise verbunden [1]. Eine andere Variante wäre die möglichst optimale Verteilung von Filialen eines Unternehmens in einem Gebiet, die dann anschließend effizient vernetzt werden sollen [2].

Eine Variante des Problems ist das „At-Least GMST“, bei dem nicht genau ein Knoten pro Cluster, sondern mindestens ein Knoten pro Cluster gewählt werden muss (siehe [9], [10]). Diese Variante soll in dieser Arbeit allerdings nicht weiter untersucht werden.

1.2 Bisherige Arbeiten zum GMST-Problem

Schwierigkeit

Myung, Lee und Tcha führten das GMSTP 1995 ein [3] und bewiesen dabei die NP-Schwierigkeit des Problems auf allgemeinen Graphen. Pop [2] zeigte, dass die NP-Schwierigkeit auch bestehen bleibt, wenn der gegebene Graph ein Baum ist. Pop gab außerdem in [4] Spezialfälle an, in denen das Problem doch in polynomieller Zeit lösbar ist, nämlich im Fall einer fixen Anzahl von Clustern und im Fall eines Baums mit einer beschränkten Anzahl von Blättern, sowie dem trivialen Fall mit nur einem Knoten pro Cluster (in diesem Fall handelt es sich um das klassische MST-Problem).

Im allgemeinen Fall gibt es für das Problem keinen Approximationsalgorithmus mit konstanter Gütegarantie [3]. Im speziellen Fall beschränkter Clustergrößen ist es allerdings möglich, wie Pop, Still und Kern in [5] zeigten (bei maximaler Clustergröße k ist eine Approximation mit Gütegarantie 2k möglich).

Exakte Algorithmen

In der ursprünglichen Formulierung [3] stellten Myung et al 4 Integer Linear Programming (ILP)-Formulierungen vor. Weitere ILP-Formulierungen wurden von Feremans, Labbe und Laporte [6] eingeführt, die diese und die ursprünglichen genauer untersuchten. Pop [4] stellte eine effiziente Mixed Integer Programming

(MIP)-Formulierung vor. Einen Branch-and-Cut-Algorithmus gibt es von Feremans, Labbe und Laporte [7].

Bis zu einer Größenordnung von ca. 200-300 Knoten (abhängig von der Anzahl der Cluster) lassen sich Instanzen vom GMSTP derzeit exakt lösen. Für größere Instanzen werden metaheuristische Verfahren verwendet.

Metaheuristiken

Pop wendete in [4] einen Simulated Annealing-Ansatz an. Deutlich bessere Ergebnisse erzielte Ghosh [8], der verschiedene Ansätze basierend auf Tabusuche, Variable Neighborhood Descent und Variable Neighborhood Search (VNS) auf Instanzen mit 100-400 Clustern und bis zu 2000 Knoten verglich. Als günstig für mittelgroße Instanzen (in diesem Fall 100 Cluster) erwies sich dabei die Tabusuche mit recency based und frequency based memory (TS2), für große Instanzen lieferte der Variable Neighborhood Decomposition Search (VNDS)-Ansatz die besten Ergebnisse.
Golden, Raghavan und Stanojevic [11] stellten einen evolutionären Algorithmus (EA) vor, sowie eine lokale Suche, die mit oberen und unteren Schranken arbeitet, und Konstruktionsheuristiken, die auf Verfahren für das klassische MST-Problem basieren. Sie erzielten dabei gute Ergebnisse.
Hu, Leitner und Raidl [12] wendeten einen VNS-Ansatz mit Nachbarschaften basierend auf den Ansätzen von Ghosh [8] und Pop [4] an. In [1] erweiterten sie diesen Ansatz um eine weitere Nachbarschaft, die Teillösungen mittels MIP optimiert.

In dieser Arbeit soll das GMSTP mittels eines EA gelöst werden. Für zusätzliche lokale Verbesserungen kommen dabei die in [12] erwähnten Nachbarschaften basierend auf Ghosh und Pop zum Einsatz. Insbesondere wird untersucht, inwiefern die Verwendung eines Lösungsarchivs zu besseren Lösungen führen kann.

2 Grundlagen

2.1 Evolutionäre Algorithmen

Evolutionäre Algorithmen (EA) sind metaheuritische Verfahren, die den natürlichen Auslese- und Evolutionsprozess simulieren. In der Natur gibt es eine große Anzahl Lebewesen, die miteinander um Ressourcen und Fortpflanzungsmöglichkeiten konkurrieren. Die stärksten / angepasstesten setzen sich eher durch und vermehren sich stärker, wodurch die gesamte Population mit der Zeit stärker wird.

Dieses Prinzip wird hier zur Lösung von Optimierungsproblemen eingesetzt. Ausgehend von einer größeren Menge an Kandidatenlösungen werden anhand der Bewertungen gute Lösungen ausgewählt (Selektion) und daraus neue Lösungen gebildet (Rekombination / Crossover). In Anlehnung an die Natur nennt man die interne Repräsentation einer Lösung auch Chromosom, und ihre einzelnen Bestandteile heißen Gene. Die Idee ist, dass die so gebildeten neuen Lösungen die guten Aspekte der zusammengesetzten Lösung vereinen sollen und so noch bessere Ergebnisse liefern. Als zusätzliche Operation gibt es noch die Mutation, die zufällig einzelne Gene verändert.

Grundsätzlich gibt es zwei Varianten, das Ersetzen alter Lösungen durch neue zu behandeln. In einem „steady-state“ EA bleibt die Population generell erhalten, ersetzt wird immer nur jeweils eine Lösung (üblicherweise die schlechteste) durch eine neu generierte. In einem „generationalen“ EA dagegen wird jeweils eine gesamte neue Generation an Lösungen erzeugt, die die alte ersetzt. Allerdings wird oftmals zumindest ein Teil der Lösungen unverändert in die nächste Generation übernommen, insbesondere die beste bislang gefundene Lösung (dieses Prinzip nennt man Elitismus).

Algorithmus 1: steady-state EA

```
Generiere zufällige Population p
Solange Abbruchkriterium nicht erfüllt
{       elternteil1 ← selektion(p)
        elternteil2 ← selektion(p)
        neu ← crossover(elternteil1, elternteil2)
        mutation(neu)
        lokaleVerbesserung(neu)
        Ersetze eine Lösung in p durch neu
}
```

Idealerweise verbessern sich im Verlauf eines EA die Lösungen in der Population, während die Diversität abnimmt (der Algorithmus konvergiert).

Repräsentation der Lösung

Die Lösungen des Problems müssen in entsprechender Weise codiert werden, um die EA-Operatoren (Crossover und Mutation) einsetzen zu können. Die so entstandene codierte Repräsentation heißt Genotyp, und die daraus berechnete konkrete Lösung des Problems heißt Phenotyp. Eine simple und häufig angewendete Variante ist die Codierung als Bitstring (Folge von 0 und 1), worauf sich die EA-Operatoren in intuitiver Weise anwenden lassen.
Gute Lösungen werden von schlechten anhand ihrer „Fitness" unterschieden, die durch eine Bewertungsfunktion (fitness function) berechnet werden. Oftmals ist es genau diese Bewertung, die während der Laufzeit des Algorithmus am meisten Rechenaufwand erfordert, weswegen eine effiziente Codierung besonders wichtig ist.

Rekombination

Um neue Lösungen zu erzeugen, werden (meistens zwei) „Eltern"-Lösungen ausgewählt, die dann auf bestimmte Weise kombiniert werden. Für die Auswahl der Eltern gibt es verschiedene Strategien, grundsätzlich gilt aber immer, dass Lösungen mit besseren Fitnesswerten mit höherer Wahrscheinlichkeit gewählt werden.
Für die eigentliche Kombination (Crossover) gibt es ebenfalls mehrere Varianten. Im Allgemeinen werden Gene der Elternlösungen zufällig gewählt, die dann zusammengesetzt werden. Häufige Varianten sind:

- 1-point crossover
  Vom ersten Gen bis zu einem zufälligen Punkt werden die Gene des ersten Elternteils übernommen, ab dann die des zweiten.

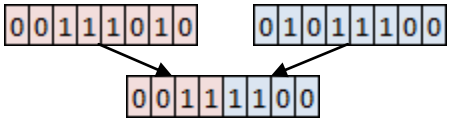

Abb. 2: 1-point crossover

- 2-point crossover
  Vom ersten Gen bis zu einem zufälligen Punkt werden die Gene des ersten Elternteils übernommen, dann die des zweiten bis zu einem weiteren zufälligen Punkt, dann wieder die des ersten.

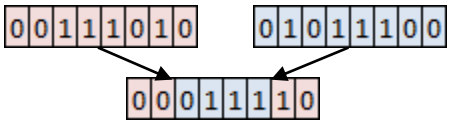

Abb. 3: 2-point crossover

- uniform crossover
  Jedes einzelne Gen wird mit einer gewissen Wahrscheinlichkeit (meistens 50%) vom ersten Elternteil übernommen, sonst vom zweiten.

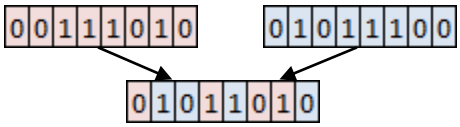

Abb. 4: uniform crossover

Mutation

Der Mutationsoperator verändert zufällig einzelne Gene der neuen Lösung. Das hat den Sinn, neue oder in früheren Generationen verloren gegangene Elemente in die Lösungen einzubringen. Auf diese Art bleibt die Diversität in der Population länger erhalten und es besteht eine höhere Chance, aus lokalen Optima zu entkommen.
Je nach Problem und konkreter Implementation des EA kann die Mutation ein unterschiedlich großer Faktor sein. Manchmal wird sie auch gar nicht verwendet. In jedem Fall sollte aber die Mutationswahrscheinlichkeit nicht zu hoch gewählt werden, da der Algorithmus sonst zu einer reinen Zufallssuche entartet.

2.2 Lösungsarchive

EAs setzen neue Lösungen aus schon bekannten Lösungen zusammen. Zudem ähneln sich die Lösungen in der aktuellen Population im Verlauf des Algorithmus immer mehr.
Diese beiden Faktoren ergeben ein Problem: Die Chance, eine „neue“ Lösung zu generieren, die in einer früheren Generation bereits untersucht wurde (ein Duplikat), ist relativ hoch und steigt mit Verlauf des Programms immer mehr an. Duplikate erneut zu evaluieren hilft nicht bei der Findung des Optimums und verbraucht unnötig Laufzeit.
Ein simpler Ansatz zur Lösung dieses Problems ist es, jede neu generierte Lösung jeweils mit den vorhandenen Lösungen in der aktuellen Population zu

vergleichen. Diese Methode kann bereits helfen, hat aber das offensichtliche Problem, dass die aktuelle Population ja nur eine gewisse Menge an Lösungen enthält und nicht alle bereits bekannten.

Hier kommt die Idee ins Spiel, bereits evaluierte Lösungen in geeigneter Form abzuspeichern. Damit kann bei neu generierten Lösungen schnell festgestellt werden, ob es sich um ein Duplikat handelt. Wenn ja, muss es nicht extra bewertet werden und kann entweder sofort verworfen oder, wenn möglich, zu einer wirklich „neuen" Lösung umgewandelt werden.

Je nach gewählter Codierung kann noch ein weiteres Problem auftreten: Lösungen, deren Genotypen sich unterscheiden, können dennoch „identisch" sein – d.h. gleiche Phenotypen haben. Trifft das auf das jeweilige Problem zu, kann man mit einer eventuellen Duplikatselimination bessere Ergebnisse erzielen, wenn man diesen Effekt berücksichtigt. Raidl und Gottlieb haben das in [13] genauer untersucht. Durch die Eigenschaften der gewählten Codierung für das GMST-Problem tritt dieser Effekt hierbei allerdings nicht auf und wird deshalb in dieser Arbeit nicht weiter behandelt.

Bisherige Arbeiten

Ansätze zur Verbesserung von EAs durch das spezielle Behandeln von Duplikaten gibt es einige. Ursprünglich konzentrierten sich die Arbeiten dabei hauptsächlich auf das Einsparen von Zeit für die Evaluierung.
Roland untersuchte dieses Problem in [15] und [16] und stellte eine „hash tagging"-Methode vor, mit der sehr effizient Lösungen erkannt werden können, die schon in der aktuellen Population enthalten sind. Povinelli und Xeng [17] verwendeten eine Hashtabelle, um die Fitnesswerte aller schon evaluierten Lösungen abzuspeichern.

Durch die Wahl einer geeigneten Datenstruktur für das Lösungsarchiv gibt es neben der eingesparten Evaluierungszeit aber noch weitere Vorteile. Ein Trie-basiertes „Complete Solution Archive", wie es in dieser Arbeit verwendet wird, wurde zunächst von Zaubzer [14] vorgestellt, der es als Ergänzung eines EA zur Lösung des mehrdimensionalen Knapsack-Problems verwendet. Neben der Zeitersparnis bei der Evaluierung bietet seine Implementierung auch die Möglichkeit, Duplikate zu neuen gültigen Lösungen umzuwandeln. Zudem können durch Schranken Teilbereiche des Suchraums ausgeschlossen werden, die keine besseren Lösungen mehr enthalten können. Ein vergleichbares Archiv wird auch von Šramko [18] für drei Probleme verwendet (MAX-SAT, Royal Road function, NK landscapes).

In allen Fällen zeigt sich, dass mit einem solchen Archiv mehr unterschiedliche Lösungen in der gleichen Zeit untersucht werden können, was in vielen Fällen zu besseren Endresultaten führt.

Die Probleme, die bislang mit dem Trie-basierten Archiv untersucht wurden, haben alle eins gemeinsam: Ihre Lösungen können einfach als Bitstrings dargestellt werden. Das führt dazu, dass als Struktur ein binärer Trie verwendet werden kann. Im Fall des GMSTP ist das allerdings nicht möglich, was zu einem anderen Aussehen des Trie und möglicherweise zu neuen Problemen führt.

Geeignete Datenstrukturen

Um eine geeignete Struktur für ein Lösungsarchiv zu finden, muss man zunächst überlegen, welche Anforderungen daran gestellt werden:

- Schnelles Einfügen von Lösungen
- Schnelles Überprüfen, ob eine Lösung enthalten ist
- Schnelles/Einfaches Umwandeln von Duplikaten in neue Lösungen
- Geringer Speicherplatzbedarf

Eine Untersuchung mehrerer in Frage kommender Strukturen (insbesondere Hashtabelle, Binärer Suchbaum und Trie) wurde von Zaubzer ([14]) und Šramko ([18]) bereits mit dem gleichen Ergebnis durchgeführt und soll daher hier nicht ausführlich stattfinden.

structure	memory	insertion	check	suggestion
hash table	$O(l.n)$	$O(l)$	$O(l)$	$O(2^l)$
binary tree	$O(l.n)$	$O(l.\log_2(n))$	$O(l.\log_2(n))$	$O(l.\log_2(n))$
trie	$O(l.n)$	$O(l)$	$O(l)$	$O(l)$

Abb. 5: Vergleich dreier Datenstrukturen hinsichtlich der Verwendung als Lösungsarchiv [18]

Man sieht, dass der Trie hinsichtlich der Geschwindigkeit sehr gute Eigenschaften bietet und vom Speicherplatz her zumindest nicht deutlich schlechter ist als die anderen Varianten.
Da aber wie schon erwähnt das GMSTP eine etwas andere Trie-Struktur erfordert (der binäre Trie, wie er für diese Tabelle angenommen wurde, ist nicht geeignet) kann der Speicherbedarf doch ein größeres Problem werden, als die Tabelle vermuten lassen würde. Dieser Aspekt wird in 3.3, „Speicherplatz“ genauer behandelt.

2.3 Tries

Allgemeine Tries

Ein Trie („Information Re**trie**val“) ist ein Baum, der als Datenstruktur zur Speicherung von Strings dient. Er wird auch als Präfixbaum bezeichnet, da alle Kinder eines Knoten in einem Trie den gleichen Präfix haben. Im Gegensatz zu einem binären Suchbaum enthalten die Knoten selbst keine Information über den String, dieser entsteht anhand des Wegs von der Wurzel bis zu diesem Knoten. Tries ermöglichen sehr effizientes Einfügen und Suchen von Werten und eignen sich besonders für die Implementierung von Wörterbüchern.

Eine Form des Trie, die für diese Arbeit relevant ist, ist der Indexed Trie. Jeder Knoten in einem Indexed Trie entspricht einer Position des Strings und enthält einen *next*-Pointer für jedes mögliche Zeichen an dieser Position. Zudem enthält er für jeden Pointer ein *end*-Flag, das angibt, ob an dieser Stelle ein String endet. Das ist nötig, um Strings zu speichern, die Präfixe eines anderen gespeicherten Strings sind.

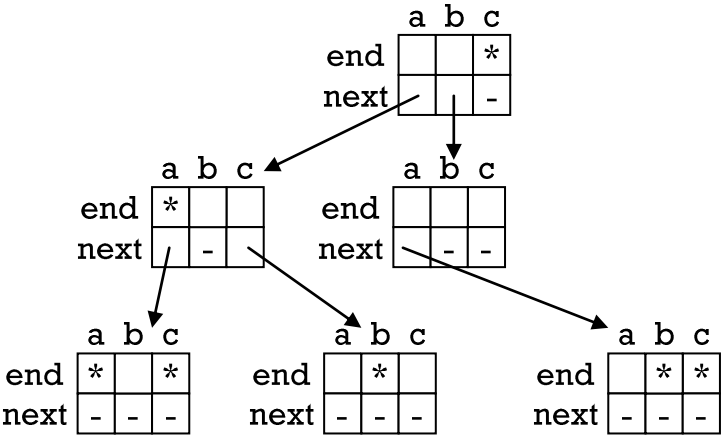

Abb. 6: Indexed Trie mit den Wörtern aa, aaa, aac, acb, bab, bac, c. (* = Flag auf true gesetzt, - = Null)

Tries als Lösungsarchive

Für die Lösungsarchive in bisherigen Arbeiten konnte man die folgenden Eigenschaften ausnutzen, um die Triestruktur zu verbessern:

- Die Lösungen können als Binärstrings dargestellt werden. Jeder Knoten benötigt also nur zwei Pointer für 0 und 1.

- Alle zu speichernden Werte haben die gleiche Länge. Ein Wert kann also kein Präfix eines anderen sein. Das *end*-Flag ist also nicht notwendig.

Es ist trotzdem nötig, das Ende einer Lösung irgendwie zu markieren, allerdings kann man anstelle des *end*-Flags direkt den *next*-Pointer verwenden, der ja an dieser Stelle nicht benötigt wird, und Speicherplatz sparen. Der Pointer wird üblicherweise als „*complete*“ markiert, was entweder das Ende einer Lösung oder einen vollständig untersuchten Subtrie repräsentieren kann.

Es ergibt sich dadurch eine sehr effiziente Struktur zur Speicherung binärer Lösungen, besonders wenn man bedenkt, dass vollständig untersuchte Teile des Trie gelöscht werden können.

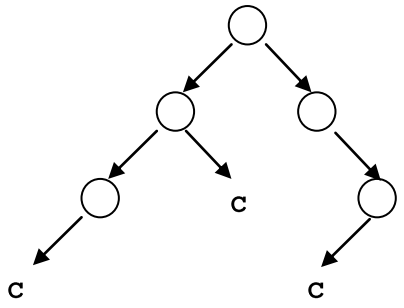

Abb. 7: Binärer Trie mit den Einträgen 000, 010, 011, 110 (links = 0, rechts = 1, c = *complete*)

Lösungen des GMST-Problems lassen sich nicht auf sinnvolle Weise als Binärstring darstellen. Ein binärer Trie dieser Art kann also nicht verwendet werden. Was allerdings erhalten bleibt, ist die gleiche Länge aller Lösungen und die Möglichkeit, vollständig untersuchte Subtries zu löschen.

3 Implementierung

Zur Lösung des GMST-Problems wurde in C++ ein EA auf Basis der EALIB-Bibliothek implementiert, welches um ein Trie-basiertes Lösungsarchiv erweitert wurde. Zur lokalen Verbesserung von Lösungen wird ein lokales Suchverfahren eingesetzt, das mit Nachbarschaften angelehnt an die Ansätze von Ghosh [8] und Pop [4] arbeitet.

Hu, Leitner und Raidl verwendeten in ihrem VNS-Ansatz [12] ebenfalls die EALIB-Bibliothek. Codeteile zur Evaluierung von GMST-Lösungen und lokale Verbesserung wurden aus dieser Arbeit übernommen.

3.1 Ansätze und Nachbarschaften

Man kann das GMST-Problem in zwei Teilprobleme aufteilen:

- Auswählen der Knoten aus jedem Cluster
- Verbinden der Cluster (bzw. der gewählten Knoten aus den Clustern)

Je nachdem, in welcher Reihenfolge diese Probleme behandelt werden, kann man zwischen zwei grundsätzlichen Ansätzen unterscheiden, die die Grundlage für die Repräsentation der Lösungen und die Nachbarschaften der lokalen Suche bilden. Diese Ansätze werden hier als Ghosh- bzw. Pop-Ansatz bezeichnet.

Der Ghosh-Ansatz

Eine Variante ist es, sich zunächst nur auf die Auswahl der Knoten zu beschränken und diese zu optimieren. Sind die Knoten festgelegt, wird das GMSTP zum klassischen MSTP, das einfach mit dem Algorithmus von Kruskal zu lösen ist.

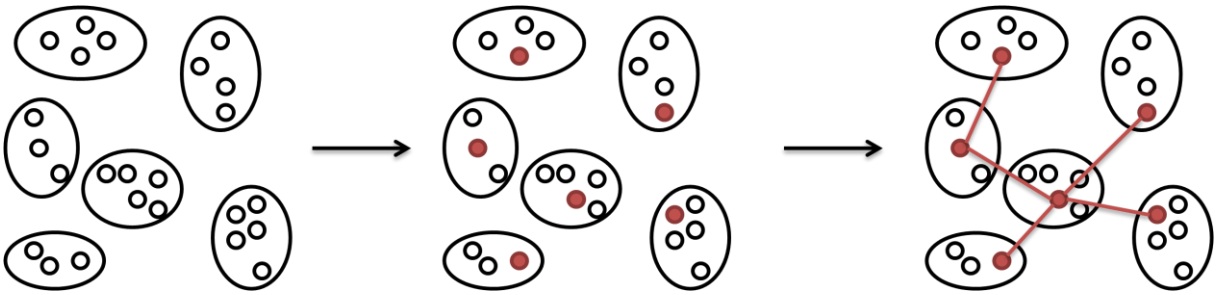

Abb. 8: Lösungsansatz basierend auf Ghosh: Zunächst Auswahl der Knoten, danach Lösung mittels Kruskals MST-Algorithmus decodieren

Da der dabei gefundene MST zu gegebenen Knoten eindeutig ist, reicht es zur Repräsentation der Lösung, die gewählten Knoten abzuspeichern. Eine Lösung ist also in diesem Fall ein Vektor $p = (p_1, p_2, \ldots p_r)$, wobei r die Anzahl der Cluster ist und p_i der in Cluster i gewählte Knoten. Diese Codierung eignet sich gut zur Anwendung in einem EA und wird deshalb auch primär verwendet.

Auf Basis dieser Repräsentation ergibt sich eine Nachbarschaft für die lokale Suche (als Ghosh-Nachbarschaft bezeichnet). Die Nachbarschaft besteht aus allen Lösungen, bei denen ein Eintrag des Vektors p verändert wurde, d.h. bei denen in einem Cluster ein neuer Knoten gewählt wurde. Die Größe dieser Nachbarschaft ist offensichtlich $|V|$, und mit Hilfe inkrementeller Evaluierung kann sie effizient durchsucht werden [12]. Dort wird ebenfalls eine Erweiterung dieser Nachbarschaft beschrieben, bei der mehrere Knoten auf einmal ausgetauscht werden. Die Größe der Nachbarschaft (und damit der Rechenaufwand) erhöht sich dadurch allerdings erheblich. Mit einer Anzahl von zwei Knoten („Ghosh2“) wird diese Erweiterung auch hier verwendet, allerdings nur in einer sehr begrenzten Anzahl von Fällen (siehe Abschnitt 3.2, „Lokale Verbesserung“).

Der Pop-Ansatz

Bei dieser Variante wird im Prinzip umgekehrt vorgegangen: Zunächst werden die Verbindungen festgelegt, dann der günstigste Knoten aus jedem Cluster ausgewählt. Dazu geht man von einem „globalen Graphen“ aus, der für jeden Cluster einen Knoten enthält (und die Knoten innerhalb der Cluster ignoriert). Auf diesem Graphen wird nun ein Spannbaum gesucht. Für diesen „globalen Spannbaum“ wird nun die günstigste Auswahl von Knoten aus jedem Cluster berechnet. Die Anzahl der Möglichkeiten ist hierbei sehr groß, kann allerdings mit einem Ansatz der dynamischen Programmierung effizient durchsucht werden. Für Details siehe [12].

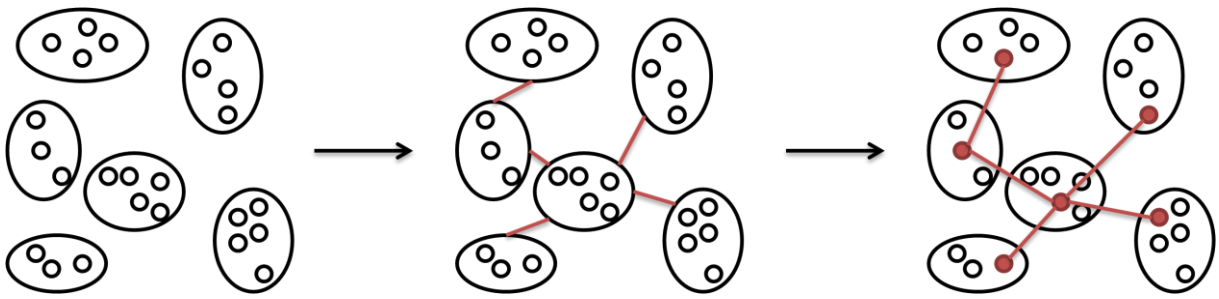

Abb. 9: Lösungsansatz basierend auf Pop: Zunächst Verbindung der Cluster, dann Wahl der Knoten

Eine Lösung wird durch die Menge der Kanten im globalen Graphen repräsentiert. Zur Evaluierung wird von dieser Codierung aus mittels des oben

genannten dynamischen Algorithmus eine GMST-Lösung erzeugt. Diese Repräsentation wird in dieser Arbeit ergänzend zum Ghosh-Ansatz ebenso verwendet, allerdings nur zum Zweck der lokalen Verbesserung von Lösungen, nicht für die Anwendung der EA-Operatoren.

Die Pop-Nachbarschaft sieht ausgehend von dieser Darstellung so aus, dass im globalen Graphen eine Kante entfernt und eine weitere Kante neu eingefügt wird, sodass der Graph ein Spannbaum bleibt. Die Größe der Nachbarschaft hängt von der Anzahl der Cluster r ab und beträgt r^2. Ob sie aufwändiger zu durchsuchen ist als die Ghosh-Nachbarschaft, hängt von der Relation der Clusteranzahl zur Knotenanzahl ab (in den meisten Fällen ist sie es).

Kombination beider Darstellungen

Die beiden genannten Ansätze sind in gewisser Weise komplementär. Der Ghosh-Ansatz löst die Verbindung der Knoten exakt und versucht die Wahl der Knoten zu optimieren. Der Pop-Ansatz löst die Wahl der Knoten exakt und versucht die Verbindung der Knoten zu optimieren. Es scheint daher günstig, beide Varianten zu kombinieren.

Abgesehen von der Verwendung der Ghosh-Nachbarschaft bei der lokalen Suche (die kein bedeutender Faktor in diesem EA ist) kommen beide Darstellungen bei jeder neu generierten Lösung zum Einsatz. Nur die Ghosh-Darstellung ist dabei für die EA-Operatoren wichtig, aber die Pop-Darstellung optimiert die gefundene Lösung. Es hat sich in Tests gezeigt, dass diese Kombination die Qualität der Lösungen erheblich verbessert.

3.2 Der Evolutionäre Algorithmus

Die Basis dieses Projekts ist ein steady-state-EA basierend auf EALIB.
EALIB ist eine C++-Bibliothek, die zum Entwickeln evolutionärer Algorithmen und anderer Optimierungsverfahren entworfen wurde. Die grundsätzlichen Abläufe des EA sind alle fertig implementiert, die Arbeit beim Einsatz dieser Bibliothek besteht hauptsächlich darin, für das Problem eine entsprechende Chromosomenklasse zu schreiben, die sich um die Repräsentation und Evaluierung von Lösungen und die genaue Funktion der EA-Operatoren für das konkrete Problem kümmert. Zusätzlich musste natürlich in diesem Fall das zu verwendende Lösungsarchiv implementiert werden.

Grundsätzlicher Ablauf des EA

Wie ein steady-state-EA grundsätzlich abläuft, wurde in Algorithmus 1 beschrieben. Ergänzt um die Einbindung des Lösungsarchivs sieht der Ablauf nun so aus:

Algorithmus 2: steady-state EA mit Archiv

Generiere zufällige Population *pop* **und füge sie in *archiv* ein**
Solange Abbruchkriterium nicht erfüllt
{ *elternteil1* ← selektion(*pop*)
elternteil2 ← selektion(*pop*)
neu ← crossover(*elternteil1*, *elternteil2*)
mutation(*neu*)
lokaleVerbesserung(*neu*)
Wenn *neu* in *archiv* enthalten (Duplikat)
Wandle *neu* zu unbesuchter Lösung um
Füge *neu* in *archiv* ein
Ersetze eine Lösung in *pop* durch *neu*
}

Die Ergänzungen im Vergleich zum Basis-EA (Alg. 1) sind im Alg. 2 fett markiert. Jede neue Lösung, die im Lauf des Algorithmus generiert wird (auch die aus der Startpopulation) wird im Lösungsarchiv gesucht. Ist sie dort schon enthalten, handelt es sich um ein Duplikat. Dieses Duplikat wird dann anhand des Archivs in eine neue Lösung umgewandelt, die garantiert noch nicht untersucht wurde.

Da es sich bei dem in dieser Arbeit verwendeten Lösungsarchiv nicht notwendigerweise um ein „vollständiges“ Archiv handelt, unterscheidet die konkrete Implementation des EA sich in Details von diesem Algorithmus. Diese werden in 3.3, „Aufteilung in mehrere Tries“ genauer beschrieben.

Repräsentation der Lösung im Chromosom

Grundsätzlich wird für Codierung einer Lösung der Ghosh-Ansatz verwendet. Das heißt, das Chromosom enthält einen Vektor mit den jeweils gewählten Knoten aus jedem Cluster. Zur Decodierung wird aus diesen Knoten ein Graph erstellt und auf diesen ein MST-Algorithmus angewendet. Übernimmt man die dabei entstandenen Kanten in den ursprünglichen Graphen, ergibt sich dadurch eine Lösung des GMST-Problems, die dann (durch Addieren der Kantenkosten) evaluiert werden kann.

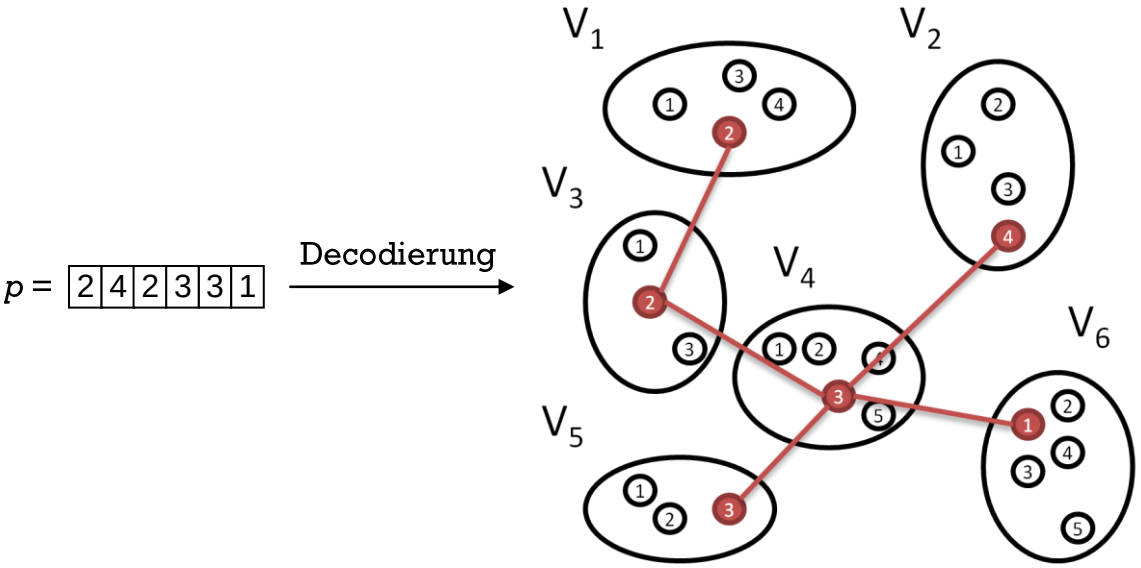

Abb. 10: Lösung codiert als Chromosom (Genotyp) und decodiert als GMST-Lösung (Phenotyp)

Die Pop-Optimierung

Eine Optimierung besteht darin, nach der Durchführung dieser auf dem Ghosh-Ansatz basierenden Decodierung, die zu den gegebenen Knoten die optimalen Kanten findet, noch den im Abschnitt „Pop-Ansatz“ erwähnten dynamischen Programmier-Algorithmus durchführt, der zu den so gefundenen Kanten die optimalen Knoten findet. Dieser Schritt erscheint vielleicht redundant, da die Optimierung der Knoten ja an sich vom EA übernommen wird, hat sich aber in der Praxis als sehr günstig erwiesen. Der zusätzliche Rechenaufwand fällt praktisch nicht ins Gewicht.

Die Anwendung dieses zusätzlichen Schritts wird im Folgenden als „Pop-Optimierung“ bezeichnet.

Wie die Testresultate zeigen, liefert diese Optimierung sehr gute Ergebnisse. Es ist daher wünschenswert, sie möglichst auf jede neue Lösung anzuwenden.

Konkret wird die Pop-Optimierung ausgeführt, nachdem alle EA-Operatoren durchgeführt wurden, d.h. nachdem Crossover und Mutation erledigt sind und ggf. eine lokale Suche durchgeführt wurde, aber vor dem Einfügen in das Lösungsarchiv.

Sollte das Lösungsarchiv die Lösung verändern, so erhält man wieder eine neue Lösung und kann nun überlegen, ob es günstig ist, erneut die Pop-Optimierung anzuwenden. Für Genaueres dazu siehe Abschnitt 4.2, „Resultate“.

Startpopulation

Für die Initialisierung der Population zu Beginn werden reine Zufallslösungen verwendet, um möglichst viel Diversität zu erzeugen. Jede Lösung enthält also aus jedem Cluster einen zufälligen Knoten.

Selektion

Die Auswahl der Lösungen für die Rekombination erfolgt mittels des „Tournament Selection"-Verfahrens. Hierbei wird eine gewisse Anzahl von Chromosomen aus der Population zufällig ausgewählt. Das beste unter diesen Chromosomen „gewinnt das Turnier" und wird einer der Elternteile. Der andere wird auf die gleiche Art bestimmt.

Durch die Wahl der Teilnehmeranzahl für eine Runde lässt sich die Geschwindigkeit der Konvergenz verändern. Bei einer hohen Anzahl haben durchschnittliche und schlechte Lösungen wegen der großen Konkurrenz nur eine minimale Chance gewählt zu werden. Das EA konvergiert dann schneller zu einem lokalen Optimum, allerdings steigt das Risiko, das globale Optimum nicht zu finden.
Aus diesem Grund wurde hier eine sehr kleine Größe verwendet, nämlich 2 (die Standardeinstellung von EALIB).

Crossover

Für die Rekombination wurde ein simples Uniform Crossover implementiert. Für jedes Gen wird individuell bestimmt, von welcher Elternlösung es übernommen wird (mit jeweils gleicher Chance).
Die Wahrscheinlichkeit, in einer Generation ein Crossover durchzuführen, beträgt 1 (eine andere Einstellung ist bei einem steady-state-EA nicht sinnvoll).

Mutation

Eine Mutation besteht darin, ein oder mehrere Gene zufällig zu verändern, d.h. in einem oder mehreren Clustern einen anderen Knoten auszuwählen. Die Anzahl der zu verändernden Gene wird jedesmal zufällig bestimmt.
Da eine Hauptfunktion der Mutation die Erhaltung der Diversität in der Population ist und diese Aufgabe hier vom Lösungsarchiv übernommen werden kann, ist die Mutation für dieses EA nicht unbedingt notwendig. Sie kann allerdings nach wie vor helfen, lokalen Optima zu entkommen.

Für die Mutationsrate wurde der EALIB-Parameter pmut auf -1 gestellt. Das bedeutet, dass bei einer neuen Lösung im Durchschnitt ein Gen verändert werden soll, oder genauer: Die Anzahl der zu mutierenden Gene wird mittels einer Poisson-verteilten Zufallszahl mit Durchschnitt 1 bestimmt. Versuche mit anderen Werten zeigten keine merklichen Unterschiede in den Resultaten.

Lokale Verbesserung

Zum lokalen Verbessern von Lösungen wird eine lokale Suche unter Verwendung der Ghosh-Nachbarschaft verwendet. Diese Suche ist allerdings trotz nur einer Nachbarschaft sehr zeitaufwändig und wird deshalb eher selten durchgeführt. Bei ohnehin guten Lösungen lässt sich normalerweise kaum noch eine Verbesserung erzielen, dazu bräuchte man eine mächtigere (und damit langsamere) lokale Suche mit mehr/größeren Nachbarschaften. Wie Tests gezeigt haben, ist das im Vergleich zu dem Fortschritt, den das EA in der gleichen Zeit erzielen kann, den Aufwand nicht wert. Bei schlechteren Lösungen ist die Chance besser, mit der lokalen Suche eine Verbesserung zu erzielen, allerdings wird die so entstandene Lösung oft immer noch relativ schlecht sein und ohnehin bald aus der Population fallen. Die Zeit für die Suche ist dann verschwendet.
In zwei Situationen kann die lokale Suche zum Einsatz kommen:

- Eine neue global beste Lösung wurde gefunden. Speziell in frühen Generationen, wo sich die beste Lösung häufig ändert, führt das zu guten Ergebnissen. Später bewirkt sie kaum noch etwas, kostet allerdings auch keine Zeit – insofern kann es durchaus sinnvoll sein, diese Optimierung durchzuführen. Für diesen Fall wird zusätzlich die „Ghosh2"-Nachbarschaft eingesetzt, bei der zwei Knoten gleichzeitig ausgewechselt werden.
- Ausführen bei jeder Lösung mit einer gewissen Wahrscheinlichkeit. Diese Wahrscheinlichkeit sollte sehr gering gewählt werden, da der Aufwand für die lokale Suche sonst schnell den Aufwand für das EA dominiert. Die Wahrscheinlichkeit, damit eine neue global beste Lösung zu finden, ist sehr gering. Allerdings ist es möglich, dass eine so gefundene Lösung zumindest teilweise gute Gene enthält, die in der Population noch nicht vorhanden oder schon verloren gegangen sind. Man kann diesen Fall also als eine Art zusätzliche Mutation sehen.
  In Tests wurden allerdings keine merklichen Verbesserungen festgestellt, weder mit hohen Werten (0,1 oder 1) noch mit niedrigen (0,005-0,01).

Zu beachten ist, dass der EALIB-Parameter *plocim* für die „Wahrscheinlichkeit einer lokalen Suche" immer auf 1 zu stellen ist, da in der aufgerufenen Funktion locallyImprove auch das Einfügen in den Trie behandelt wird. Für die

Konfiguration der eigentlichen lokalen Suche innerhalb dieser Funktion gibt es eigene Parameter (siehe Abschnitt 4.3).

3.3 Der Trie

Die hier verwendete Triestruktur basiert auf dem Indexed Trie (siehe Abschnitt 2.3, „Allgemeine Tries“), allerdings ohne *end*-Flag, und nicht jeder Knoten hat die gleiche Anzahl von Pointern. Jeder Knoten des Trie repräsentiert einen Cluster des zu untersuchenden Graphen und enthält einen Pointer für jeden Knoten in diesem Cluster.

Man kann nun überlegen, welchem Trieknoten man welchen Cluster zuordnet. Die offensichtliche Variante besteht darin, der *i*-ten Ebene des Graphen jeweils Cluster V_i zuzuordnen. Für die weiteren Erklärungen in diesem Abschnitt wird vorerst diese Variante angenommen.

Es spricht allerdings einiges dafür, die Zuordnung zufällig zu machen. Dazu später mehr.

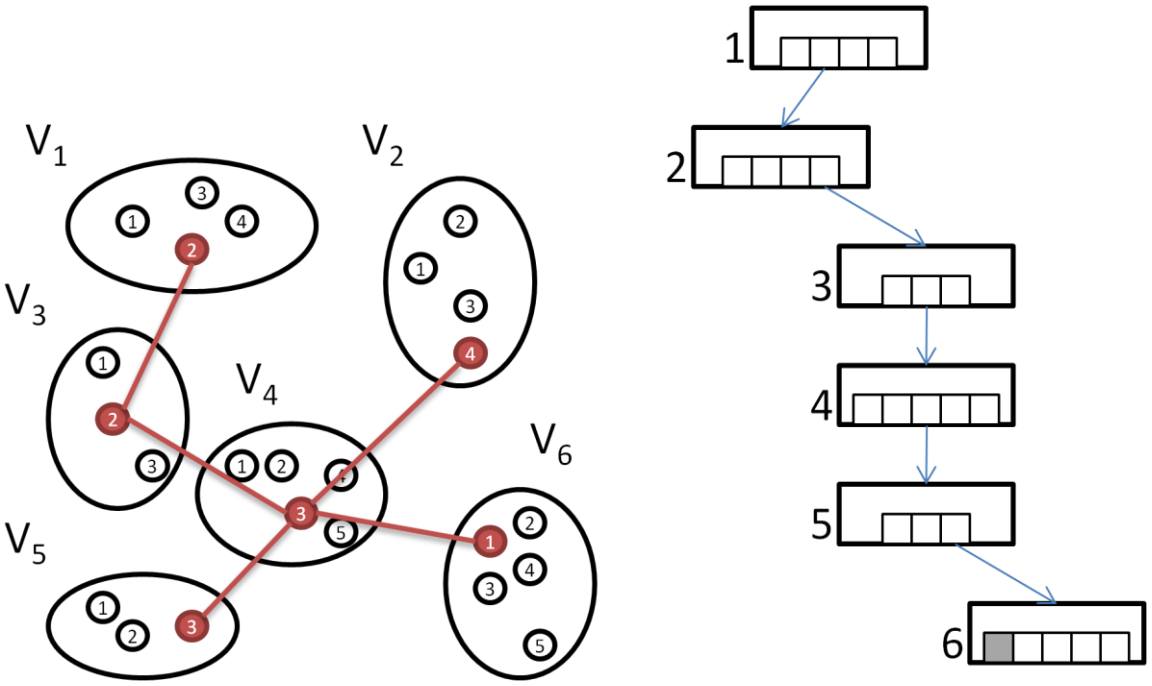

Abb. 11: Lösung [242331] in den Trie eingefügt (leere Pointer = *null*, grau markiert = *complete*)

Für jeden Pointer gibt es 3 mögliche Zustände:

- *null*: Dieser Wert gibt an, dass es noch keine Lösung gibt, die im entsprechenden Cluster diesen Knoten enthält (und den Rest der Lösung

von der Wurzel bis zu diesem Pointer). Solange noch keine Lösung eingefügt wurde, ist die Wurzel *null*.
- *complete*: Dieser Wert gibt an, dass diese Richtung nicht weiter verfolgt werden muss. Entweder, weil es sich hierbei um das Ende der Lösung handelt (wie in Abb. 11) oder weil bereits sämtliche Lösungen, die im folgenden Subtrie enthalten sind, untersucht wurden. Sollte im Verlauf des Algorithmus die Wurzel *complete* sein, wurde der gesamte Lösungsraum untersucht.
- Normaler Pointer zum nächsten Trie-Knoten. Das bedeutet, dieser Knoten wurde bereits in einer Lösung gewählt, es gibt aber im Folgenden Verzweigungen, die noch nicht untersucht wurden.

Speicherplatz

Der genaue Speicherplatzbedarf des Trie hängt stark von der Anzahl und der Struktur der Lösungen ab. Haben viele Lösungen einen gemeinsamen Präfix, wird weniger Platz verbraucht als bei stark unterschiedlichen Lösungen. Durch das Abtrennen vollständig untersuchter Subtries können im späteren Verlauf neue Lösungen sogar Speicher einsparen, anstatt zusätzlichen zu verbrauchen.

Die Höhe des nicht-leeren Trie entspricht stets der Anzahl der Cluster r, es sei denn, jede einzelne Option auf der tiefsten Ebene wurde bereits untersucht (kommt nur bei sehr kleinen Problemen oder sehr langen Laufzeiten vor). Sei die maximale Anzahl der Knoten in einem Cluster m.

Die erste Ebene des Trie enthält nur die Wurzel. Jede weitere enthält allerdings im Worst Case bis zu m mal so viele wie die vorige (jeder Knoten auf Ebene i verlinkt auf bis zu m Knoten der Ebene $i+1$). Die Breite auf Ebene i beträgt also bis zu m^{i-1}, und auf der tiefsten Ebene r dementsprechend m^{r-1}.

Man sieht leicht, dass hier auch bei relativ kleinen Instanzen astronomisch hohe Werte zustande kommen. Gehen wir davon aus, dass jeder Knoten m 32-bit-Pointer und sonst nichts enthält, erhalten wir für die Größe allein der tiefsten Ebene bei einer Instanz mit 100 Knoten und 20 Clustern ($m = 5$)

$$5^{19} * 4 * 5 = 381.469.726.562.500\ B \approx 347\ TB$$

Ein offensichtlich nicht vertretbarer Aufwand. Allerdings spielen diese theoretischen Werte in der Praxis keine Rolle, schließlich werden ja nur Lösungen eingefügt, die durch das EA zunächst erzeugt wurden, und die Anzahl dieser Lösungen ist durch die Laufzeit begrenzt. Die dabei auftretenden Werte für den Speicherbedarf erreichen nicht annähernd diese Größenordnung. Selbst bei

größeren Instanzen mit über 400 Knoten und 80 Clustern bewegte sich der Speicherbedarf bei 10 Minuten Laufzeit eher im Bereich 20-30 MB, was kein Problem darstellt.

Durch die exponentielle Skalierung mit der Anzahl der Cluster kann aber bei noch größeren Instanzen und längeren Laufzeiten sehr wohl ein Speicherproblem auftreten. Um auch für diese Zwecke einsetzbar zu sein, gibt es die Idee, nicht den gesamten Graphen in einem Trie darzustellen, sondern nur einen Teil. Mehrere Tries ergeben dann die Gesamtlösung.

Aufteilung in mehrere Tries

Wenn als Archiv mehrere Tries verwendet werden sollen, repräsentiert jeder einzelne Trie einen Teil des ursprünglichen Graphen. Wir haben also mehrere Tries T_1, T_2, … T_k. Jedem Trie ist eine Teilmenge der Cluster zugeordnet, wobei die Teilmengen paarweise disjunkt sind.

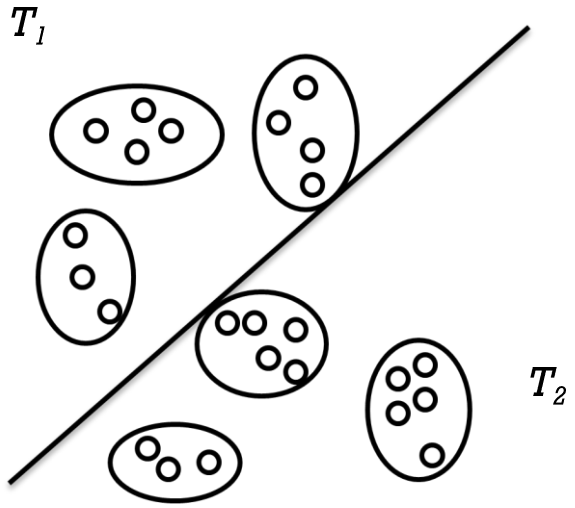

Abb. 12: Aufteilung des Graphen bei Verwendung von 2 Tries

Damit lässt sich eine erhebliche Speicherplatzersparnis erzielen. Bei 2 Tries halbiert sich die Anzahl der Cluster für jeden Trie und damit auch der Exponent beim Worst Case-Speicherplatz. Anstelle der Größenordnung m^r haben wir nun 2 Tries mit jeweils $m^{r/2}$, also insgesamt $2m^{r/2}$. Bei vergleichbarem Worst Case-Speicherbedarf lassen sich also Instanzen mit ca. doppelt so vielen Clustern behandeln, und bei gleich großen Instanzen verringert sich der Worst Case-Speicherbedarf größenordnungsmäßig auf die Wurzel des Werts mit einem Trie.

Allgemein haben wir analog zum oben genannten Worst Case-Fall bei k Tries $km^{r/2-1}$ Knoten auf der untersten Ebene des so entstehenden „Walds“. Betrachten

wir das oben genannte Beispiel mit 100 Knoten und 20 Clustern (das vorhin 347 TB benötigte), aber diesmal mit zwei Tries.

$2 * 5^9 * 4 * 5 = 78.125.000 \approx 75\,MB$

Diese Ersparnis hat allerdings ihren Preis, denn eine wichtige Eigenschaft des Lösungsarchivs geht dabei verloren: Es ist nicht mehr eindeutig feststellbar, ob eine Lösung schon enthalten ist.

Betrachten wir als Beispiel den Graphen aus Abb. 11 und die Lösung [242331]. Trie T_1 enthält die Teillösung für die Cluster V_1-V_3 (also [242]), T_2 die für V_4-V_6 (also [331]).

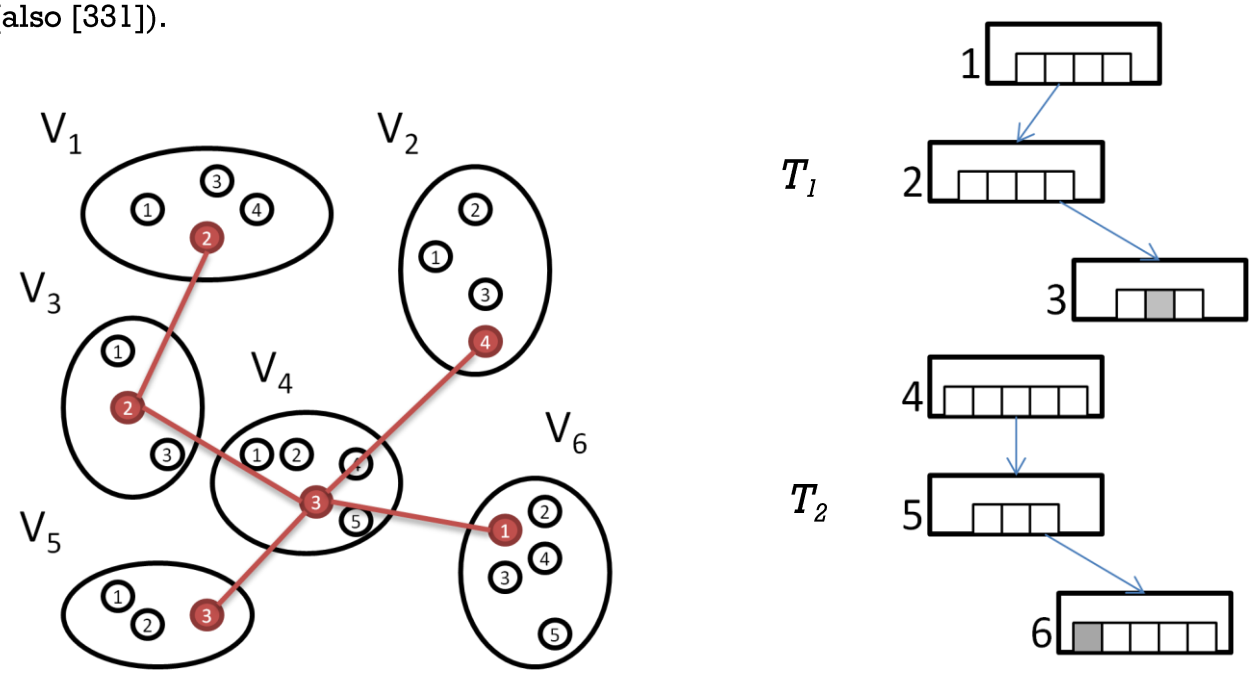

Abb. 13: Lösung [242331] in 2 Tries gespeichert

T_1 enthält [242] und T_2 enthält [331]. Noch ist kein Problem zu sehen. Will man überprüfen, ob die Lösung [242331] enthalten ist, sucht man in T_1 nach [242] und in T_2 nach [331], wird fündig und erkennt die Lösung korrekt als schon vorhanden. Überprüft man eine beliebige andere Lösung, findet man sie zumindest in einem der beiden Tries nicht, erkennt sie also korrekt als noch nicht vorhanden.

Das Problem zeigt sich, sobald mehr als eine Lösung gespeichert ist. Fügen wir nun Lösung [211531] ein.

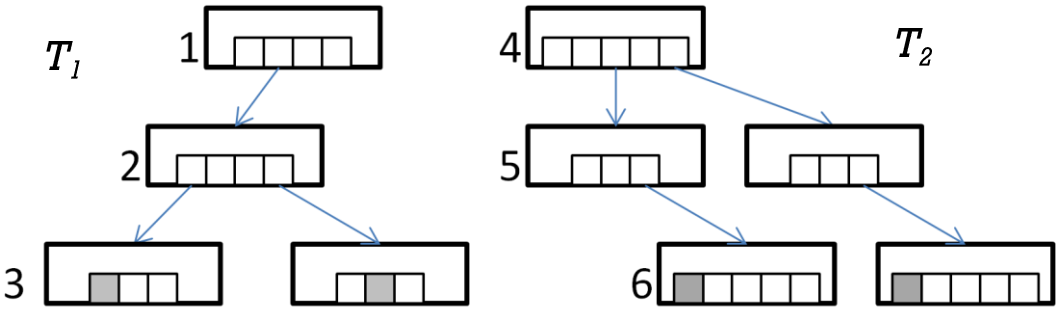

Abb. 14: Lösungen [242331] und [211531] in 2 Tries gespeichert

T_1 enthält nun [242] und [211], T_2 enthält [331] und [531]. Nach wie vor werden beide gespeicherten Lösungen korrekt erkannt. Ebenso werden allerdings die Lösungen [242531] und [211331] erkannt, die nicht explizit abgespeichert wurden.

Jeder Trie enthält Teillösungen. Es sind aber keine Informationen darüber bekannt, in welcher Kombination diese Teillösungen ursprünglich vorhanden waren. Deshalb werden nach dem Einfügen von n Lösungen in das Archiv im Worst Case (d.h. wenn die n Lösungen keine gemeinsamen „Hälften“ haben) n^2 Lösungen als Duplikate erkannt.
Nun ist es aber so, dass die einzufügenden Lösungen von einem EA erzeugt werden. Durch die Eigenschaften eines EA ähneln sich die dabei auftretenden Lösungen, und gemeinsame Teillösungen sind sehr häufig. Insofern hat man es in der Praxis nicht wirklich mit der vollen quadratischen Anzahl „falscher“ Duplikate zu tun, aber die Anzahl ist trotzdem nicht zu vernachlässigen.

Da man sich bei Verwendung mehrerer Tries nicht mehr sicher sein kann, dass eine als Duplikat erkannte Lösung wirklich doppelt ist, werden in diesem Fall zwei Ergänzungen zum Ablauf vorgenommen:

Algorithmus 3: steady-state EA mit Archiv (für mehrere Tries)

```
Generiere zufällige Population p und füge sie in archiv ein
Solange Abbruchkriterium nicht erfüllt
{       elternteil1 ← selektion(p)
        elternteil2 ← selektion(p)
        neu ← crossover(elternteil1, elternteil2)
        mutation(neu)
        lokaleVerbesserung(neu)
        Wenn neu in archiv enthalten (Duplikat) und neu kein neues Optimum
        {       Lasse neu mit gewisser (kleiner) Wahrscheinlichkeit
                unverändert, sonst wandle neu zu unbesuchter Lösung um
        }
        Füge neu in archiv ein
        Ersetze eine Lösung in p durch neu
}
```

Erstens wurde ein „Aspirationskriterium“, wie es aus der Tabusuche bekannt ist, hinzugefügt. Wenn die neue Lösung besser als die bislang beste gefundene Lösung ist, wird sie auf jeden Fall akzeptiert (in diesem Fall ist ja offensichtlich, dass es sich nicht um ein Duplikat handeln kann). Zweitens besteht eine gewisse Wahrscheinlichkeit, dass ein Duplikat unabhängig von dieser Bedingung akzeptiert wird. Dadurch ist garantiert, dass jede noch nicht untersuchte Lösung die Chance hat, akzeptiert zu werden. Dass dadurch auch einige echte Duplikate in die Population aufgenommen werden, muss dabei in Kauf genommen werden. Natürlich darf die Wahrscheinlichkeit zum Akzeptieren eines Duplikats nicht sehr hoch sein, da sonst der gesamte Effekt des Archivs verloren geht. In Tests wurden Werte von 0,05-0,1 verwendet.

Werden noch mehr als 2 Tries verwendet, nehmen auch die ungünstigen Effekte entsprechend zu. Bei k Tries werden durch das Einfügen von n Lösungen bis zu n^k Lösungen als Duplikate erkannt. Riesige Bereiche des Lösungsraums werden dadurch fälschlicherweise ausgeschlossen. Es ist daher empfehlenswert, möglichst wenige Tries einzusetzen. In den meisten praktischen Szenarien wird einer völlig ausreichen. Bei sehr großen Instanzen und/oder langen Laufzeiten sind möglicherweise zwei nötig. Mehr als zwei sollten nur in absoluten Ausnahmefällen verwendet werden.

Zuteilung von Clustern zu Tries

Um mehrere Tries einzusetzen, müssen die Cluster gruppiert werden, sodass jedem Trie eine dieser Gruppen zugeordnet werden kann.

Ein Ziel für diese Gruppierung ist, sicherzustellen, dass die Tries jeweils möglichst gleich viele Cluster enthalten. Enthält ein Trie nämlich zuviele Cluster, ähnelt die Variante zu sehr der mit nur einem Trie. Der Speicherplatzvorteil ist dann nicht mehr gegeben. Enthält ein Trie dagegen zuwenige, ist er zu schnell „voll“ und erkennt dann jede Teillösung als Duplikat, womit unnötigerweise große Teile des Lösungsraums ausgeschlossen werden.

Die einfachste Variante ist, die Cluster der Nummerierung nach zuzuteilen. Bei 100 Clustern und 2 Tries würde man also V_1-V_{50} dem ersten Trie und V_{51}-V_{100} dem zweiten Trie zuordnen. Eine andere Variante wäre eine zufällige Aufteilung.
Man könnte auch versuchen, die Cluster „intelligenter“ zuzuordnen, indem man geographisch nahe beisammen liegende Cluster dem gleichen Trie zuordnet. Dieser Ansatz wird in dieser Arbeit verwendet, und zwar mit einem sehr simplen Greedy-Algorithmus: Es wird für jeden Trie jeweils ein beliebiger noch nicht zugeordneter Cluster gewählt und anschließend die Cluster, die diesem am nächsten sind dem Trie zugeordnet, bis r/k Cluster gewählt sind (r = Anzahl der Cluster, k = Anzahl der Tries).

Das ist ein Punkt, den man vermutlich noch optimieren könnte. Allerdings ist unklar, ob eine andere Gruppierung der Cluster überhaupt zu Verbesserungen führt, solange das oben erwähnte Ziel erreicht wird. Die Unterschiede in den Resultaten sind jedenfalls minimal und daher schwierig zu testen.

Klassenstruktur

Das Lösungsarchiv wurde in Form von 3 Klassen implementiert:

- *solutionArchive*: Diese Klasse repräsentiert das gesamte Lösungsarchiv. Sie enthält einen Vektor von *gmstTrie*s und die Funktionen *insert* und *convert* zum Einfügen von Lösungen und Umwandeln von Lösungen zu neuen. Bei der Initialisierung führt die Klasse auch die Zuteilung von Clustern zu Tries durch.
- *gmstTrie*: Ein einzelner Trie des Lösungsarchivs zur Speicherung von Teillösungen (oder ganzen Lösungen, sofern nur ein Trie verwendet werden soll). Ein *gmstTrie* enthält einen Pointer zur Wurzel des Trie, sowie einen Vektor, der die Nummern der diesem Trie zugeordneten Cluster enthält. Funktionen sind *insert* und *convert*, analog zum *solutionArchive* (von wo aus sie auch aufgerufen werden).

- *trieNode*: Ein Knoten eines Trie. Er enthält *next*-Pointer zu den Kinderknoten (da die Anzahl dieser nicht fix ist, wird dafür ein Vektor verwendet) und die Nummer des ihm zugeordneten Clusters. Neben kleinen Hilfsfunktionen bietet *trieNode* vor allem die Funktion *checkCompleteness*, die vollständig untersuchte Teilbäume als *complete* markiert und den entsprechenden Speicherplatz wieder freigibt.

Initialisierung

Zu Beginn des Algorithmus wird ein *solutionArchive* erzeugt, das dann wiederum die gewünschte Anzahl *gmstTries* erstellt und ihnen Cluster zuweist. Die Tries selbst sind zu diesem Zeitpunkt noch leer (die Wurzeln zeigen jeweils auf *null*). Trieknoten werden nur nach Bedarf erzeugt, der Wurzelknoten entsteht dabei, sobald die erste Lösung eingefügt wird.

Bei der Initialisierung eines Knotens muss zunächst bestimmt werden, welcher Cluster diesem Knoten zugeordnet wird (dieser kann der Tiefe des Knotens entsprechen oder auch zufällig gewählt werden, siehe dazu Abschnitt 3.3, „Zuordnung von Clustern zu Trieknoten“). Anschließend wird der *next*-Vektor initialisiert (enthält einen Pointer pro Knoten in diesem Cluster).

Einfügen und Überprüfen von Lösungen

Das Einfügen einer Lösung in den Trie ist relativ simpel – beginnend von der Wurzel wird jeweils dem Pointer gefolgt, der dem gewählten Knoten aus dem jeweiligen Cluster entspricht. Ist dieser Pointer noch *null*, wird ein neuer Knoten eingefügt. Ist er *complete*, ist die Lösung schon vorhanden und die Funktion bricht ab. Sobald eine Lösung erfolgreich eingefügt wurde, wird überprüft, ob durch das Einfügen neue Knoten als *complete* markiert werden können und damit Speicherplatz freigegeben werden kann.
Die folgenden Pseudocodes sind etwas detaillierter und teilweise an C++-Syntax angelehnt.

Algorithmus 4: gmstTrie::insert

Input: Einzufügende Lösung *sol*
Output: Erfolgreich eingefügt (*true/false*)

```
Wenn root = null
{       Bestimme Cluster c für root
        root ← neuer Knoten mit Cluster c
}
Wenn root = complete                          // Baum vollständig untersucht
        return false
curr ← root                                   // Aktueller Knoten
for(i = 0; i < Anzahl der Cluster im Trie; i++)
{       pos ← sol[Cluster von curr]
        Wenn curr→next[pos] = complete        // Lösung schon enthalten
                return false
        Wenn curr→next[pos] = null
        {       Bestimme Cluster c für neuen Knoten
                curr→next[pos] ← neuer Knoten mit Cluster c
        }
        curr ← curr→next[pos]
}
Entlang von sol auf vollständig untersuchte Teilbäume prüfen
return true                                   // Erfolgreich eingefügt
```

Eine eigene Funktion zum Suchen von Lösungen im Trie ist nicht nötig. Wenn im Programm eine Lösung im Trie gesucht und nicht gefunden wird, soll sie immer gleich eingefügt werden, also kann *insert* diese Aufgabe übernehmen.

Was hier nicht genauer erläutert wurde, ist, wie genau ein Cluster für einen Knoten bestimmt wird. Siehe dazu Abschnitt 3.3, „Zuordnung von Clustern zu Trieknoten“.

Der obige Pseudocode beschreibt die insert-Funktion eines einzelnen Tries. Die insert-Funktion des eigentlichen Archivs ruft für jeden einzelnen Trie diese Funktion auf. Das Einfügen gilt als erfolgreich, wenn es zumindest für einen Trie erfolgreich ist.
Liefern alle einzelnen *insert*-Funktionen *false*, so handelt es sich bei der Lösung um ein Duplikat (bei mehr als einem Trie allerdings ein möglicherweise falsches Duplikat).

Algorithmus 5: solutionArchive::insert

Input: Einzufügende Lösung *sol*
Output: Erfolgreich eingefügt (*true/false*)

```
success ← false
Für jeden Trie t des Archivs
{       t→insert(sol)
        Wenn insert erfolgreich
                success ← true
}
return success
```

Vollständig untersuchte Teilbäume

Sind alle von einem Knoten ausgehenden Wege vollständig untersucht worden (d.h. alle Pointer des Knoten sind *complete*), so kann der Knoten selbst freigegeben und der auf ihn zeigende Pointer auf *complete* gesetzt werden. Auf diese Art wird Speicher gespart, und die Laufzeit verbessert sich, da der Versuch, eine Lösung einzufügen frühzeitig abgebrochen werden kann.

Die Durchführung dieses Vorgangs ist die Aufgabe der Funktion *checkCompleteness*. Sie wird immer dann aufgerufen, wenn eine neue Lösung in den Trie eingefügt wurde. Dabei muss nicht der gesamte Trie untersucht werden. Knoten, die zuvor nicht *complete* waren und die sich beim Einfügen nicht verändert haben, sind sicher weiterhin nicht *complete*. Es genügt also, die zu untersuchen, die sich verändern konnten – also die in der gerade eingefügten Lösung enthaltenen Knoten.

Der Aufruf der Funktion erfolgt immer von der Wurzel aus, also *root→checkCompleteness*(*sol*). Beginnend von der Wurzel wird dann im Trie entlang der übergebenen Lösung rekursiv nach unten gewandert. Es wird jeweils untersucht, ob der aktuelle Knoten als *complete* markiert werden kann und das Resultat zurückgegeben. Eine Ebene in der Rekursion höher wird der Knoten dann tatsächlich freigegeben und als *complete* markiert, wenn möglich. Ob eine Markierung möglich ist, hängt davon ab, ob alle next-Pointer *complete* sind. Sind sie das nicht, muss aber immer noch weiter entlang der Lösung nach unten gewandert werden. Die Ergebnisse der unteren Ebenen könnten ja einen weiteren *next*-Pointer als *complete* markieren, womit auch der aktuelle Knoten möglicherweise markiert werden kann.

Der Rückgabewert des ursprünglichen Aufrufs gibt an, ob die Wurzel als *complete* markiert ist, also der gesamte Baum vollständig untersucht wurde.

Algorithmus 6: trieNode::checkCompleteness

Input: Lösung *sol*, entlang der der Trie untersucht werden soll
Output: Knoten ist complete (*true/false*)

```
Wenn alle next-Pointer complete sind
        return true
cluster ← Nummer des Clusters, den der aktuelle Knoten repräsentiert
Wenn next[sol[cluster]] = null oder next[sol[cluster]] = complete
        return false
Wenn next[sol[cluster]]→checkCompleteness(sol) = true
{       next[sol[cluster]] löschen und Speicher freigeben
        next[sol[cluster]] ← complete
        Wenn alle next-Pointer complete sind
                return true
        Sonst
                return false
}
Sonst
        return false
```

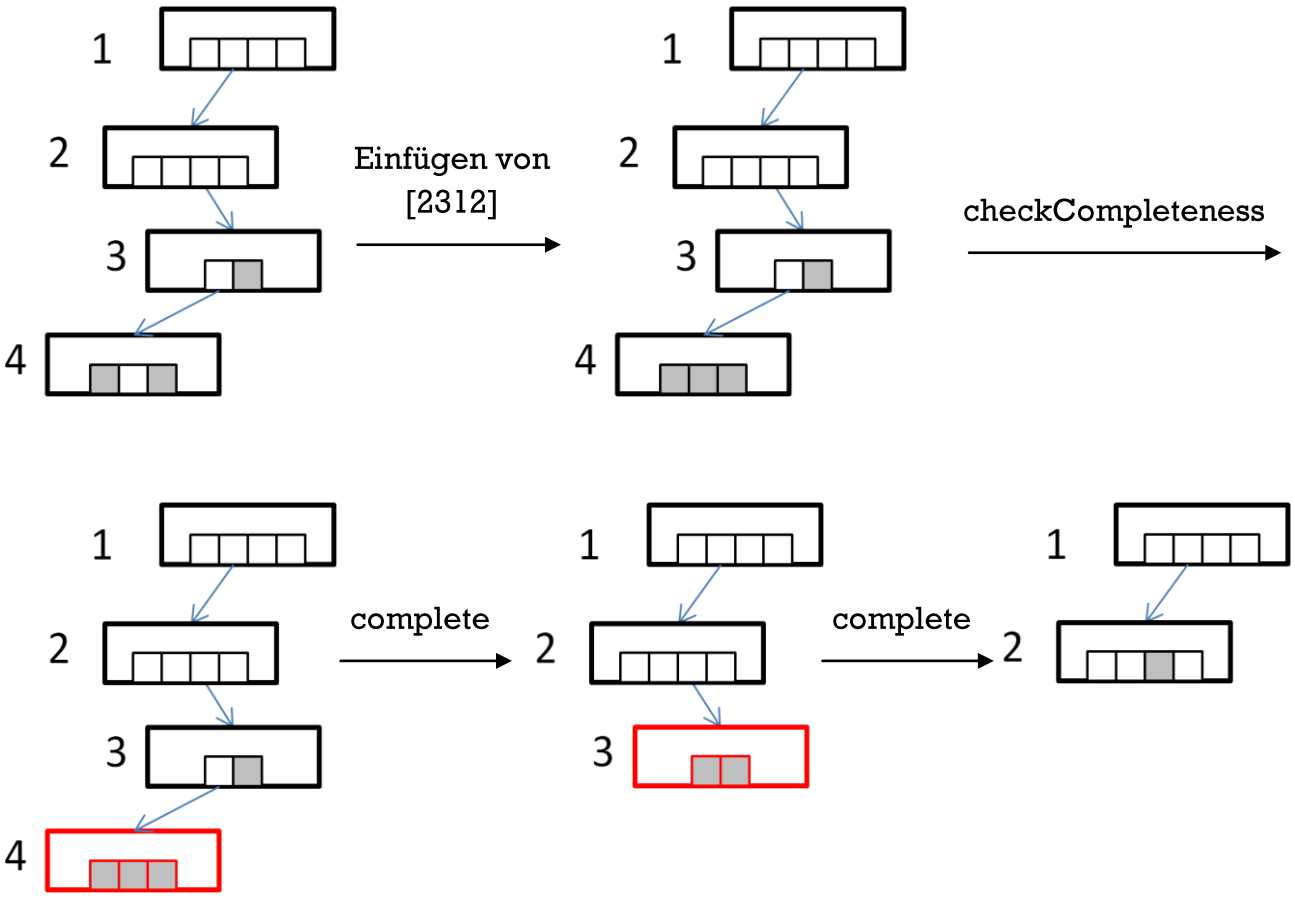

Abb. 15: Einfügen von Lösung [2312] und darauf folgendes Löschen von Teilbäumen (grau = *complete*)

Abb. 15 illustriert diesen Vorgang anhand eines Beispiels. Der Trie enthält zu Beginn die Lösungen [2311] und [2313], sowie alle Lösungen mit dem Präfix [232] (siehe *complete*-Markierung auf der dritten Ebene). In diesen Trie wird nun die Lösung [2312] eingefügt und anschließend *checkCompleteness* aufgerufen. Durch die eingefügte Lösung ergibt sich ein Knoten, in dem alle Pointer auf *complete* gesetzt sind. Dieser kann also entfernt werden und der auf ihn zeigende Pointer wird wiederum *complete* gesetzt. Dadurch ergibt sich ein weiterer Knoten, der entfernt werden kann.
Im Endeffekt enthält der verbleibende Trie nur noch eine *complete*-Markierung. Diese besagt, dass der Trie alle Lösungen mit dem Präfix [23] enthält. Die einzelnen Lösungen zu speichern ist nicht mehr nötig.
Man sieht an diesem Beispiel, dass das Einfügen einer Lösung den Trie auch kleiner machen kann.

Finden unbesuchter Lösungen

Der größte Zeitvorteil bei der Verwendung des Archivs besteht darin, dass doppelte Lösungen nicht einfach verworfen, sondern auf effiziente Weise zu einer neuen Lösung umgewandelt werden, die garantiert kein Duplikat ist. Das wird von der Funktion *convert* übernommen.

Zur besseren Verständlichkeit wird hier nicht sofort die endgültige Version dieser Funktion angegeben, sondern sie wird schrittweise erweitert. Beginnen wir mit der Basisversion:

Soll eine als Duplikat erkannte Lösung umgewandelt werden, beginne bei der Wurzel und suche nach noch nicht benutzten Verzweigungen (*null*-Pointer). Gibt es so eine, ändere die Lösung entsprechend ab und lasse sie ansonsten unverändert (Abbruch). Gibt es keine, folge der alten Lösung und wiederhole den Vorgang im nächsten Knoten. Es ist aber möglich, dass der zur alten Lösung führende Pointer bereits auf *complete* gesetzt ist. In diesem Fall muss ein anderer Weg gewählt werden, der noch nicht *complete* ist.
Anstatt bei der Wurzel zu beginnen, könnte man vom unteren Ende der Lösung nach oben vorgehen. Das grundsätzliche Verhalten des Algorithmus ändert sich dabei nicht, allerdings besteht eine höhere Chance, dass sich im unteren Teil des Trie vollständig untersuchte Subtries bilden, die man dann löschen kann. Diese Variante hat allerdings das gleiche Grundproblem wie die zuvor beschriebene (siehe später).

Als Rückgabewert gibt die Funktion an, ob eine neue Lösung gefunden wurde (sollte nur dann *false* sein, wenn der gesamte Baum untersucht wurde, also die Wurzel *complete* ist).

Algorithmus 7: gmstTrie::convert (Version 1)

Input: Umzuwandelnde Lösung *sol*
Output: Erfolgreich umgewandelt (*true*/*false*)

```
curr ← root                                          // Aktueller Knoten
Solange curr != null und curr != complete
{       Wenn curr mindestens einen null-Pointer enthält
        {       a ← Nummer eines zufälligen null-Pointers in curr
                sol[curr→cluster] ← a                // Lösung erfolgreich
                return true                          // umgewandelt
        }
        Wenn curr→next[sol[curr→cluster]] != complete
                curr ← curr→next[sol[curr→cluster]]  // Folge der alten Lösung
        Sonst
        {       a ← Nummer eines zufälligen nicht-complete-Pointers in curr
                sol[curr→cluster] ← a
                curr ← curr→next[sol[curr→cluster]]
        }
}
return false
```

Hier ein Beispiel. Wir haben einen Trie, der unter anderem die Lösung [24231] enthält. Angenommen, diese Lösung taucht im späteren Verlauf wieder auf: Sie wird als Duplikat erkannt und soll nun mittels des obigen Algorithmus zu einer neuen Lösung umgewandelt werden.

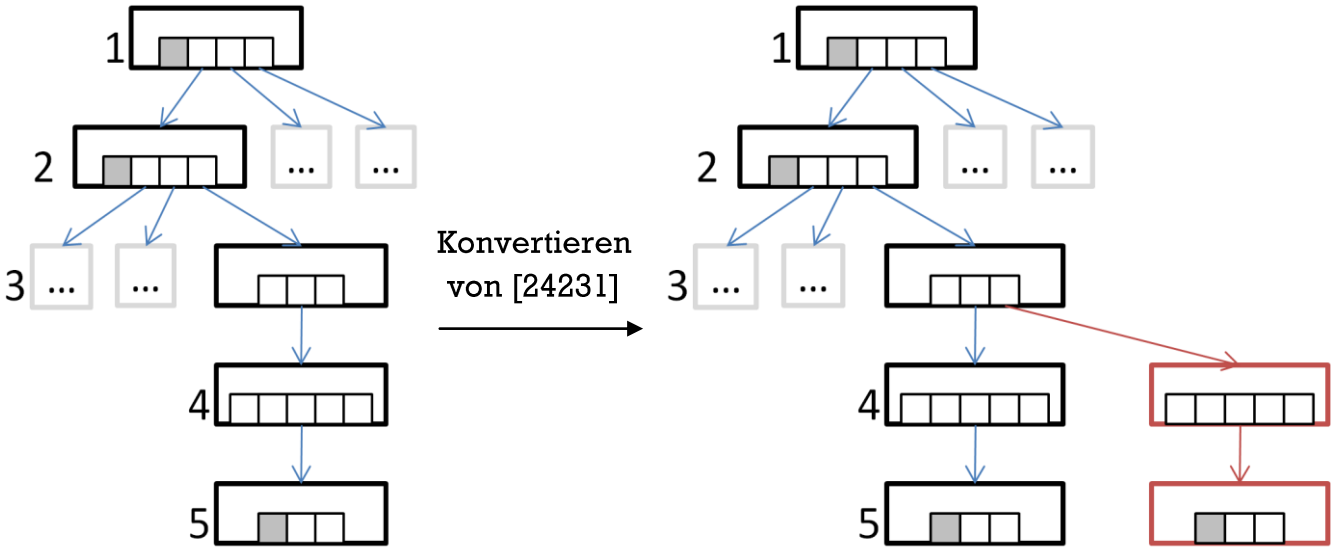

Abb. 16: Umwandeln der doppelten Lösung [24231] zur neuen Lösung [24331] mittels convert, Version 1

Bei der Wurzel gibt es keinen *null*-Pointer. Also wird weiter der alten Lösung gefolgt. Das gleiche passiert im nächsten Knoten. Im darauf folgenden Knoten

gibt es zwei *null*-Pointer. Einer davon wird zufällig ausgewählt. An dieser Stelle kann der Vorgang abgebrochen werden, der Rest der Lösung bleibt unverändert. Sobald diese neue Lösung [24331] in den Trie eingefügt wird, ergibt sich das obige Bild.

Dieser Algorithmus verändert offensichtlich die Lösung relativ wenig – wann immer möglich, wird der alten Lösung gefolgt, und beim ersten *null* wird sofort abgebrochen – und ist effizient: Im Worst Case muss der Trie einmal von oben bis unten durchgewandert werden, also Laufzeit k (bei k Clustern).

Es gibt aber ein Problem: Da die Suche jedes Mal bei der Wurzel begonnen wird und schrittweise die Ebenen nach unten abgearbeitet werden, besteht für die Knoten weiter oben eine höhere Chance, verändert zu werden, also für die unteren. Ein Knoten auf der untersten Ebene kann ja nur verändert werden, wenn es in keinem einzigen Knoten darüber einen *null*-Pointer gab. Entsprechend werden die ersten Gene der Lösung wesentlich öfter verändert als die letzten.

Ein einfacher Ansatz für dieses Problem besteht darin, nicht grundsätzlich bei der Wurzel zu beginnen, sondern an einer zufälligen Stelle der Lösung, wobei darauf geachtet werden muss, dass dieser Teil nicht innerhalb eines als *complete* markierten Bereichs des Trie liegt. Auf diese Art hat jeder Knoten die gleiche Chance, als erstes gewählt zu werden.
Die Situation verbessert sich dadurch, allerdings dreht sich das Problem in gewisser Weise um. Wird im ersten Knoten kein *null*-Pointer gefunden, so wird weiter nach unten gesucht. Das heißt, es besteht immer die Chance, dass sich in einem der unteren Knoten etwas ändert, egal wo begonnen wurde. Aber es kann nie nach oben gewandert werden, d.h. die oberen Knoten können sich nur verändern, wenn entsprechend weit oben begonnen wurde. Das Problem eines „gebiasten" Tries wird also nicht gelöst, sondern verschoben (allerdings wird der Effekt abgeschwächt). Für einen besseren Lösungsansatz für dieses Problem siehe Abschnitt 3.3, „Zuordnung von Clustern zu Trieknoten".

In jedem Fall verschlechtert die Zufallsauswahl die Situation nicht, und den unteren Knoten eine höhere „Chance" auf Änderungen zu geben, hat auch einen Vorteil: Es ergeben sich dadurch schneller vollständig untersuchte Knoten, die dann freigegeben werden können.

Algorithmus 8 enthält diese Erweiterung. Die Ergänzungen im Vergleich zu Alg. 7 sind fett markiert.

Algorithmus 8: gmstTrie::convert (Version 2)

Input: Umzuwandelnde Lösung *sol*
Output: Erfolgreich umgewandelt (*true*/*false*)

```
startpoints ← leeres Array
curr ← root
Solange curr != complete                    // Mögliche Startpunkte in
{       curr in startpoints einfügen        // Lösung bestimmen
        curr ← curr→next[sol[curr→cluster]]
}
curr ← zufälliger Eintrag aus startpoints   // Aktueller Knoten
Solange curr != null und curr != complete
{       Wenn curr mindestens einen null-Pointer enthält
        {       a ← Nummer eines zufälligen null-Pointers in curr
                sol[curr→cluster] ← a       // Lösung erfolgreich
                return true                 // umgewandelt
        }
        Wenn curr→next[sol[curr→cluster]] != complete
                curr ← curr→next[sol[curr→cluster]]   // Folge der alten Lösung
        Sonst
        {       a ← Nummer eines zufälligen nicht-complete-Pointers in curr
                sol[curr→cluster] ← a
                curr ← curr→next[sol[curr→cluster]]
        }
}
return false
```

Hier erneut das obige Beispiel, diesmal wurde bei einem zufälligen Startpunkt begonnen (in diesem Fall auf Ebene 4). Es ergibt sich eine andere Lösung.

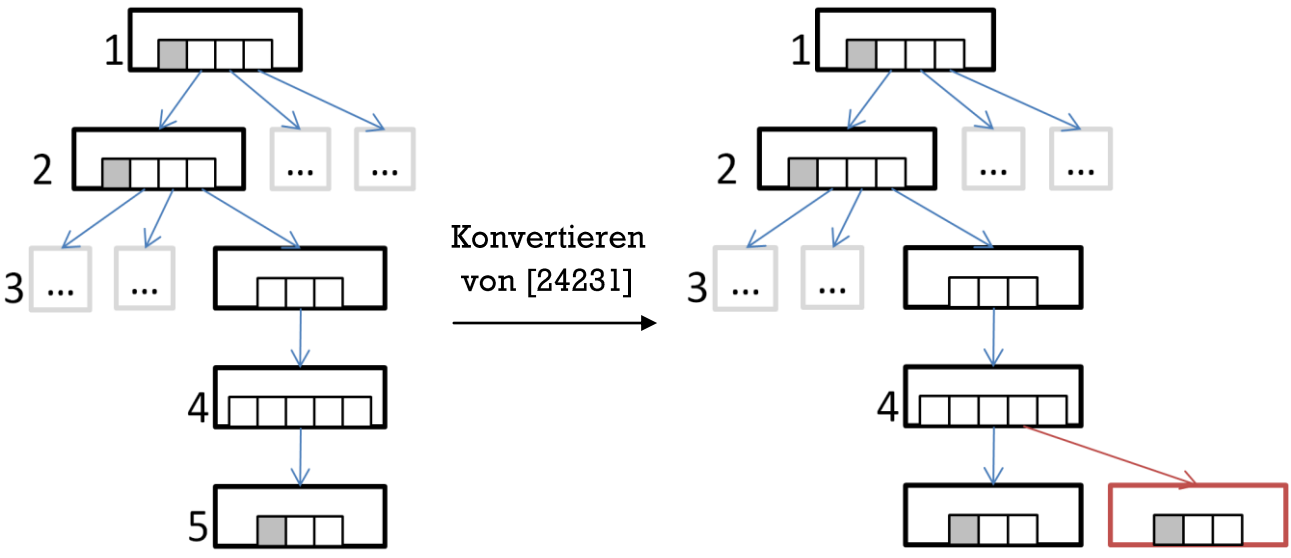

Abb. 17: Konvertieren der doppelten Lösung [24231] zur neuen Lösung [24241] mittels convert, Version 2

Eine weitere Überlegung besteht darin, im Fall einer Auswahl zwischen mehreren Pointern eine möglichst gute Wahl zu treffen. Dieser Fall tritt dann auf, wenn ein Knoten mehrere *null*-Pointer enthält, oder wenn ein Knoten keine *null*-Pointer enthält und der Weg entlang der alten Lösung bereits *complete* ist. In beiden Fällen wurde die Auswahl bislang zufällig getroffen. Es könnte aber sinnvoll sein, an dieser Stelle zusätzliche Informationen einfließen zu lassen, um die „richtige" Änderung vorzunehmen. Eine Information, die sich dafür anbietet, ist die beste bislang gefundene Lösung.
So ergibt sich die endgültige Version der *convert*-Funktion. Änderungen im Vergleich zu Alg. 8 sind fett markiert.

Algorithmus 9: gmstTrie::convert (Endgültige Version)

Input: Umzuwandelnde Lösung *sol*, Basislösung *base*
Output: Erfolgreich umgewandelt (*true/false*)

```
startpoints ← leeres Array
curr ← root
Solange curr != complete                          // Mögliche Startpunkte in
{       curr in startpoints einfügen              // Lösung bestimmen
        curr ← curr→next[sol[curr→cluster]]
}
curr ← zufälliger Eintrag aus startpoints         // Aktueller Knoten
Solange curr != null und curr != complete
{       Wenn curr mindestens einen null-Pointer enthält
        {       Wenn curr→next[base[curr→cluster]] = null
                        a ← base[curr→cluster]
                Sonst
                        a ← Nummer eines zufälligen null-Pointers in curr
                sol[curr→cluster] ← a             // Lösung erfolgreich
                return true                       // umgewandelt
        }
        Wenn curr→next[sol[curr→cluster]] != complete
                curr ← curr→next[sol[curr→cluster]]   // Folge der alten Lösung
        Sonst
        {       Wenn curr→next[base[curr→cluster]] != complete
                        a ← base[curr→cluster]
                Sonst
                        a ← Nummer eines zufälligen nicht-complete-Pointers in curr
                sol[curr→cluster] ← a
                curr ← curr→next[sol[curr→cluster]]
        }
}
return false
```

Der Parameter *base* (Basislösung) ist üblicherweise die beste bislang gefundene Lösung. Übergibt man stattdessen die alte Lösung (d.h. *base* = *sol*), entspricht der Ablauf Version 2 des Algorithmus (Alg. 8).

Wann immer eine Auswahl zwischen mehreren Optionen getroffen werden muss, wird zunächst überprüft, ob eine dieser Optionen dem entsprechenden Teil der Basislösung entspricht. Wenn ja, wird diese Option gewählt. Wenn nein, bleibt es wie bisher bei der Zufallsentscheidung.

convert gibt es, so wie *insert*, in den Klassen *solutionArchive* und *gmstTrie*. Die Funktion des Archivs ruft dabei die Funktion der einzelnen Tries in zufälliger Reihenfolge auf. Sobald ein Trie eine neue Lösung liefert, kann abgebrochen werden (die so entstandene Lösung kann kein Duplikat mehr sein).

Algorithmus 10: solutionArchive::convert

Input: Umzuwandelnde Lösung *sol*, Basislösung *base*
Output: Erfolgreich umgewandelt (*true*/*false*)

```
tries ← Liste aller Tries des Archivs
Mische tries zufällig durch
Für jeden Trie t aus tries
{       t→convert(sol, base)
        Wenn convert erfolgreich
                return true
}
return false
```

Zuordnung von Clustern zu Trieknoten

Der Trie ist so aufgebaut, dass jeder Knoten einem Cluster des Originalgraphen entspricht. Es stellt sich die Frage, welcher Cluster welchem Knoten zugeordnet werden soll. Bislang wurde angenommen, dass die simpelste Variante verwendet wird, nämlich Knoten auf der *i*-ten Ebene jeweils Cluster V_i zuzuordnen.

Der Vorteil dieser Variante ist ihre Einfachheit und Effizienz: Die aktuelle Ebene kann bei den jeweiligen Algorithmen einfach mitgezählt werden. Es sind also keine weiteren Berechnungen nötig und es wird kein zusätzlicher Speicherplatz verbraucht, um die Zuordnung abzuspeichern.

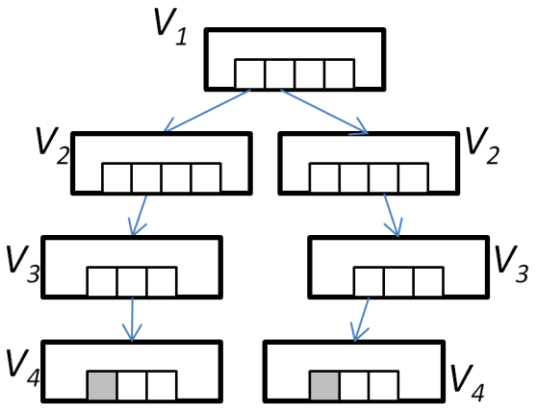

Abb. 18: Zuordnung von Trie-Ebene *i* zu Cluster V_i (Trie enthält [1221] und [2311])

Aber die Variante hat auch einen erheblichen Nachteil. Wie in vorigen Abschnitt beschrieben wurde, gibt es das Problem des Bias im Trie. Je nach Variante ist es wahrscheinlicher, dass die oberen Ebenen des Trie verändert werden, oder die unteren – in jedem Fall aber ist die Wahrscheinlichkeit nicht für alle Ebenen gleich. Ordnet man nun jede Ebene fix einem Cluster zu, so ergibt sich daraus automatisch, dass auch die Wahrscheinlichkeiten für Änderungen in jedem Cluster unterschiedlich sind.
Im allgemeinen Fall kann man aber nicht wissen, in welchem Cluster die Änderungen „wichtiger" sind, d.h. in welchen Clustern eine Änderung zu besseren Ergebnissen führen würde. Es ist also wünschenswert, dass alle Cluster möglichst gleich oft für eine Änderung gewählt werden.

Wie aber zuvor festgestellt wurde, gibt es keine Möglichkeit (oder genauer: es wurde keine gefunden), sicherzustellen, dass Änderungen in allen Ebenen des Trie gleich wahrscheinlich sind. Daraus folgt, dass eine fixe Zuweisung eines Clusters zu einer Ebene nicht den gewünschten Effekt haben kann.

Gesucht ist also eine Zuordnung, die nicht von der Ebene des Trie abhängt. Gleichzeitig muss aber die Zuordnung für jeden Knoten eindeutig sein, sie muss also von anderen bekannten Informationen abhängen – und es darf nicht passieren, dass mehrere Knoten innerhalb einer Lösung dem gleichen Cluster zugeordnet sind. Zudem soll diese Zuordnung nicht bei jedem Aufruf des Algorithmus identisch sein, also ein Zufallselement enthalten („random ordering", siehe z.B. [14], [18]) Überlegt man nun, welche Informationen verwendet werden könnten, bietet sich der Präfix an.

Wie im Abschnitt 3.1, „Allgemeine Tries" beschrieben wurde, ist ein Trie ein „Präfixbaum", d.h. ein Knoten steht für Lösungen mit einem bestimmten Präfix. Basierend auf diesem kann man nun dem jeweiligen Knoten einen Cluster zuordnen.

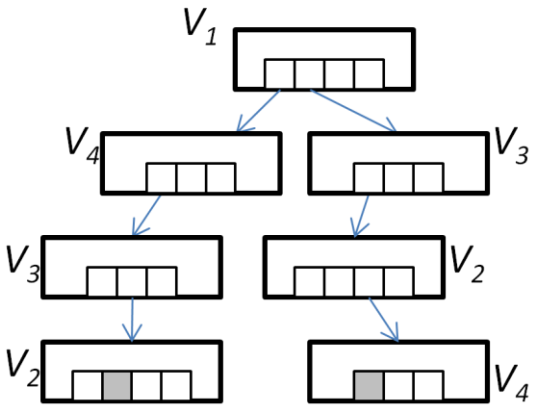

Abb. 19: Zuordnung von Clustern zu Trieknoten unabhängig von der Ebene (Trie enthält [1221] und [2311])

Der in Abb. 19 gezeigte Trie enthält die gleichen Lösungen wie der in Abb. 18, aber die Struktur ist offensichtlich anders. Knoten auf der gleichen Ebene sind unterschiedlichen Clustern zugeordnet, abhängig davon, welche Verzweigungen auf dem Weg von der Wurzel gewählt wurden.

Knoten auf Ebene 2 sind also von der jeweiligen Wahl in Ebene 1 abhängig: Beginnt eine Lösung mit 1, entspricht der darauffolgende Knoten V_4. Beginnt sie stattdessen mit 2, entspricht er V_3.
Knoten auf tieferen Ebenen sind von der Wahl in **jeder** darüberliegenden Ebene abhängig. Ein Knoten auf Ebene 3 entspricht in diesem Beispiel V_3, wenn auf der ersten Ebene 1 und auf der zweiten Ebene ebenfalls 1 gewählt wurde. Wurde auf Ebene 1 dagegen 2 gewählt, entspricht er V_2.
Die Wurzel hat keinen Präfix. Ihr Cluster muss also fix zugeordnet sein. In diesem Fall entspricht sie V_1, eine sinnvolle Variante ist eine zufällige Wahl.

Eine derartige Variante kann das Bias-Problem sehr stark verbessern. Nach wie vor kann nicht garantiert werden, dass Knoten auf jeder Ebene gleich oft gewählt werden. Sehr wohl allerdings ist garantiert, dass verschiedene Cluster ungefähr gleich oft gewählt werden (die Cluster haben ja mit der Ebene nichts mehr zu tun).

Nun stellt sich die Frage, wie so eine Struktur implementiert werden soll, insbesondere, wie man aus dem Präfix die Nummer eines zulässigen (d.h. noch nicht benutzten) Clusters erhält. Zaubzer [14] und Šramko [18] haben eine derartige Struktur verwendet und benutzen dabei eine Pseudozufallsfunktion, die aus den Teillösungen und einem seed die Clusternummer generiert.
Der Vorteil bei diesem Ansatz liegt darin, dass innerhalb der Knoten kein zusätzlicher Speicher benötigt wird, um die Clusternummer abzuspeichern, da sie bei jeder Abfrage neu berechnet wird. Das spielt bei den in [14] und [18] verwendeten Tries insbesondere eine größere Rolle, weil es sich um binäre Tries

handelt. Da auf diese Art nur zwei Pointer (=8 Byte) pro Knoten benötigt werden und die Clusternummer einen weiteren Integer (=4 Byte) hinzufügen würde, stiege der Speicherplatzbedarf sofort um 50% an. Die Variante mit der ständigen Neuberechnung spart also viel Speicher, und da es sich um binäre Lösungen handelt, ist sie auch nicht sehr aufwändig.

Eine derartige Variante wäre für das GMST-Problem auch denkbar. Gewählt wurde allerdings eine etwas andere: Der Cluster, der einem Knoten zugeordnet ist, wird nur ein einziges Mal berechnet und dann im Knoten abgespeichert. Der Nachteil dieser Variante bleibt erhalten – ein zusätzlicher Integer pro Knoten. Allerdings fallen die dazu notwendigen 4 Byte in diesem Fall relativ gesehen weniger ins Gewicht, da einzelne Knoten ohnehin schon deutlich größer sind (anstelle von fixen 8 Byte sind es 4 Byte pro Pointer, plus Overhead für den Vektor).
Worin besteht der Vorteil? In erster Linie in der Einfachheit, sowohl was das Verständnis des Codes betrifft, als auch die Berechnung des jeweiligen Clusters (man hat es hier nicht mit binären Lösungen zu tun, es ist also nicht so einfach, die Teillösung direkt als Argument für eine Pseudozufallsfunktion zu verwenden).

Implementiert wurde dieses „random ordering“ wie folgt:
Das Erstellen neuer Knoten geschieht immer im Rahmen von *gmstTrie*::*insert* (Alg. 4). Jeweils beim Erstellen eines Knoten wird ihm ein Cluster zufällig zugewiesen. Welche Cluster noch verfügbar sind, kann ermittelt werden, indem zu Beginn der Funktion eine Liste aller Cluster erstellt wird und beim Durchwandern des Trie jeweils die schon „vergebenen“ Cluster aus der Liste entfernt werden.
Muss die Wurzel neu erstellt werden (im ersten Aufruf von *insert*), so bekommt sie einen zufällig ausgewählten Cluster, wobei alle Cluster noch zur Verfügung stehen.

Hier noch einmal der Pseudocode für die *insert*-Funktion, wobei die für das random ordering relevanten Teile detaillierter ausgeführt sind (man beachte, dass intern die Cluster von 0 bis k-1 nummeriert sind). Die Unterschiede zu Alg. 4 sind fett markiert.

Algorithmus 11: gmstTrie::insert (mit random ordering)

Input: Einzufügende Lösung *sol*
Output: Erfolgreich eingefügt (*true/false*)

```
Wenn root = null
{       k ← Anzahl der Cluster des Graphen
        c ← Zufallszahl < k
        root ← neuer Knoten mit Cluster c
}
Wenn root = complete                          // Baum vollständig untersucht
        return false
clusters ← Liste mit allen Zahlen von 0 bis k-1
curr ← root                                   // Aktueller Knoten
for(i = 0; i < Anzahl der Cluster im Trie; i++)
{       pos ← sol[Cluster von curr]
        Entferne den Cluster von curr aus clusters
        Wenn curr→next[pos] = complete        // Lösung schon enthalten
                return false
        Wenn curr→next[pos] = null
        {       c ← Zufälliger Eintrag aus clusters
                Entferne c aus clusters
                curr→next[pos] ← neuer Knoten mit Cluster c
        }
        curr ← curr→next[pos]
}
Entlang von sol auf vollständig untersuchte Teilbäume prüfen
return true                                   // Erfolgreich eingefügt
```

Da alle für diese Variante notwendigen Schritte sehr einfach sind, ist der zusätzliche Laufzeitaufwand im Vergleich zur trivialen Zuordnung minimal.

4 Tests und Resultate

4.1 Testinstanzen

Zum Testen wurden grundsätzlich 3 Arten von Instanzen verwendet.

- TSPlib-Instanzen:
  Die größeren Instanzen aus TSPlib mit geographischem Clustering, so wie es in [4] verwendet wurde. Diese Instanzen haben zwischen 150 und 442 Knoten und unterschiedliche Knotenanzahlen in jedem Cluster (im Durchschnitt 5).

- „Grouped Euclidean"-Instanzen:
  Hierbei werden Instanzen zufällig generiert, indem für jeden Cluster ein Quadrat der Seitenlänge *span* erzeugt wird, die als (*row* x *col*)-Raster im Abstand *sep* angeordnet werden. Innerhalb jedes Quadrats werden zufällig Knoten generiert, die dann diesem Cluster zugeordnet werden. Jeder Cluster hat gleich viele Knoten. Diese Instanzen basieren auf der Arbeit von Ghosh [8] und wurden ebenfalls in [4] verwendet.

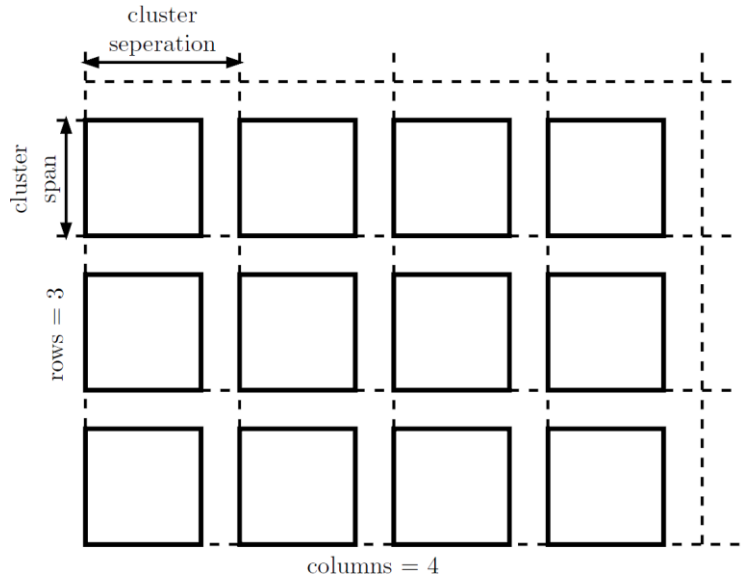

Abb. 20: Grouped Euclidean-Instanzen [4].

- „Random Euclidean"-Instanzen:
  Ebenfalls in [4] verwendet, hierbei werden Instanzen vollständig zufällig generiert, indem Knoten zufällig innerhalb eines Bereichs verteilt und zufällig Clustern zugeordnet werden. Im konkreten Fall wurden diese Instanzen auf die gleiche Art wie die „Grouped Euclidean"-Instanzen erzeugt, indem *sep* auf 0 gesetzt wurde.

Alle Instanzen sind vollständige Graphen.

Es wurden nicht bei jedem Test alle Instanzen verwendet, besonders in der früheren Phase. Die ersten Tests verwenden hauptsächlich die TSPlib-Instanzen.

Die Tests wurden auf einem Intel Core 2-Prozessor mit 2,83 GHz und 4 GB RAM durchgeführt. Es wurden jeweils 10 Durchläufe pro Instanz und Test gemacht und die Durchschnittswerte berechnet.

4.2 Ergebnisse

Als erster Schritt wurde das EA ohne Archiv implementiert, und mit den in [4] verwendeten TSPlib-Instanzen getestet, um die Ergebnisse vergleichen zu können und eine Idee zu haben, wie effektiv der Algorithmus grundsätzlich arbeitet.
Die Spalte „Vergleichswert“ bezieht sich hierbei auf die beste in [4] angegebene Lösung und dient zum Vergleich mit bisherigen Ansätzen.

				EA ohne Archiv		Vergleichswert
	$\|V\|$	k	Zeit (s)	$C(T)$	St.abw.	$C(T)$
kroa150	150	30	150	**9815**	0	**9815**
krob200	200	40	300	**11244**	0	**11244**
ts225	225	45	300	**62268,3**	0,5	62268,5
gil262	262	53	300	**942**	0	**942**
pr264	264	54	300	**21886**	0	**21886**
pr299	299	60	450	20343	24,3	**20332,6**
lin318	318	64	450	18519,7	8,8	**18506,8**
rd400	400	80	600	**5942,1**	6,3	5943
fl417	417	84	600	**7982**	0	**7982**
pr439	439	88	600	**51796,3**	12,4	51847,9
pcb442	442	89	600	19656,2	38,8	**19621**

Abb. 21: Erste Resultate EA ohne Archiv

Bei dieser Variante wird unter anderem auch eine lokale Suche mittels ghosh-Nachbarschaften mit einer Wahrscheinlichkeit von 0,5% durchgeführt, die allerdings minimale Auswirkungen auf die Lösungen hatte (sofern die Suche nicht zufällig innerhalb einer der ersten Generationen aufgerufen wurde, konnte sie im allgemeinen keinen einzigen Verbesserungsschritt finden) und deshalb im Folgenden zumeist nicht verwendet wurde.
Man sieht, dass die gefundenen Resultate bereits sehr gut sind, in einigen Fällen besser als die besten bislang bekannten. Dafür ist, wie sich später herausstellte, vor allem die Pop-Optimierung verantwortlich, die in diesem Test bereits verwendet wurde.

Als nächstes ist natürlich interessant, was das Archiv bewirkt. Hier sind die Ergebnisse mit der ersten Version des Archivs. Zu beachten ist, dass hierbei noch stark mit der Anzahl der Tries experimentiert wurde – in diesem Fall wurde als Einstellung dafür die dritte Wurzel aus der Anzahl der Cluster verwendet (das sind bei diesen Größen 3-4 Tries). Die lokale Suche blieb im Vergleich zu vorhin unverändert, in dieser Variante war zudem noch kein Aspirationskriterium und kein „random ordering" implementiert.

				EA ohne Archiv		EA mit Archiv	
	$\|V\|$	k	Zeit (s)	$C(T)$	St.abw.	$C(T)$	St.abw.
kroa150	150	30	150	**9815**	0	**9815**	0
krob200	200	40	300	**11244**	0	11245,1	3,5
ts225	225	45	300	**62268,3**	0,5	62268,8	0,4
gil262	262	53	300	**942**	0	**942**	0
pr264	264	54	300	**21886**	0	21886,7	2,2
pr299	299	60	450	20343	24,3	**20318,5**	4,6
lin318	318	64	450	18519,7	8,8	**18509,2**	9,9
rd400	400	80	600	**5942,1**	6,3	5945	8,3
fl417	417	84	600	**7982**	0	**7982**	0
pr439	439	88	600	**51796,3**	12,4	51801,3	14,9
pcb442	442	89	600	19656,2	38,8	**19646,6**	35,3

Abb. 22: Erste Resultate mit Archiv

Die Ergebnisse lassen noch keine klaren Schlüsse zu. In ein paar Fällen scheint die Version mit Archiv bessere Ergebnisse zu liefern, aber insgesamt sind die Resultate zu ähnlich.

Die Resultate dieses Tests zeigten aber etwas, das später noch Probleme machen sollte, nämlich die Effekte der Pop-Repräsentation. Bis jetzt war vorgesehen, dass die Optimierung durch diese Repräsentation nach jeder Änderung der Lösung durchgeführt wird. Im Fall der Verwendung des Archivs bedeutet das, dass zunächst jede Lösung nach der Anwendung der EA-Operatoren einmal optimiert wird, anschließend anhand des Archivs überprüft, und im Fall einer Umwandlung ein weiteres Mal optimiert wird. Dabei ergibt sich das Problem, dass diese zweite Optimierung das Lösungsarchiv „unterlaufen" kann, da die umgewandelten Lösungen erneut verändert werden und dann wieder Duplikate sein könnten. Das ist in diesem Fall daran zu sehen, dass das Testprotokoll eine Menge „duplicate eliminations" angibt, obwohl die entsprechende Wahrscheinlichkeit für diesen Test auf 0 gesetzt wurde und somit ein richtig funktionierendes Lösungsarchiv keine doppelten Lösungen akzeptieren dürfte.

Dieses Problem wurde vorerst gelöst, indem die Pop-Optimierung nach der Umwandlung nicht mehr verwendet wurde.

Als nächstes ist interessant, zu sehen, wie groß das Problem der doppelten Lösungen überhaupt ist, d.h. wieviele Duplikate in einem Durchlauf vorkommen. Betrachten wir als erstes die Anzahl der Generationen, die in der gegenenen Zeit bearbeitet werden, und die Anzahl der duplicate eliminations, die EALIB standardmäßig durchführt. Zu beachten ist hierbei, dass EALIB nur Duplikate innerhalb der aktuellen Population erkennen kann. Die tatsächliche Anzahl von Duplikaten (unter Berücksichtigung aller bislang erzeugten Lösungen) wird also deutlich höher sein. Das Archiv wurde hierbei nicht verwendet.

				EA ohne Archiv	
	$\|V\|$	k	Zeit (s)	Generationen	Dup.elims.
kroa150	150	30	150	88081	68694
krob200	200	40	300	95083	54690
ts225	225	45	300	73282	23991
gil262	262	53	300	57985	37936
pr264	264	54	300	59486	38662
pr299	299	60	450	65468	23400
lin318	318	64	450	59704	29668
rd400	400	80	600	48951	28194
fl417	417	84	600	42498	39180
pr439	439	88	600	39749	24107
pcb442	442	89	600	35716	17456

Abb. 23: Anzahl der Generationen und duplicate eliminations

Trotz der Einschränkung dieser duplicate elimination-Methode ist schon zu sehen, dass sehr viele Lösungen unnötig mehrmals untersucht werden. Für genauere Werte, die Duplikate während der gesamten Laufzeit beinhalten, betrachten wir nun die Resultate mit Archiv.

				EA mit Archiv	
	$\|V\|$	k	Zeit (s)	Generationen	converts
kroa150	150	30	150	59540	57284
krob200	200	40	300	69836	66074
ts225	225	45	300	59000	53306
gil262	262	53	300	42149	39728
pr264	264	54	300	41330	38962
pr299	299	60	450	51341	46042
lin318	318	64	450	44785	41323
rd400	400	80	600	35868	30664
fl417	417	84	600	35475	35294
pr439	439	88	600	29225	26665
pcb442	442	89	600	28860	23108

Abb. 24: Anzahl der Generationen und erfolgreich umgewandelter Lösungen durch das Archiv

Für diesen Test wurde nur ein Trie verwendet. Es ist also garantiert, dass alle als Duplikat erkannten Lösungen tatsächlich Duplikate sind. Zu erkennen ist die extrem hohe Anzahl von mehrfach vorkommenden Lösungen, teilweise sind über 99% aller Lösungen Duplikate! Ebenso sieht man, dass die Anwendung des Trie Zeit braucht, da die Anzahl der insgesamt untersuchten Generationen im Vergleich zum vorigen Test gesunken ist. Dafür ist aber garantiert, dass in jeder Generation eine wirklich neue Lösung untersucht wird.

Was zählt, sind die „effektiven" Generationen: Also die, in denen eine neue Lösung untersucht wird. Stellen wir dazu die Ergebnisse der beiden vorigen Tests gegenüber.

	EA ohne Archiv			EA mit Archiv
	Generationen	Dup.elims.	Effektiv	Effektiv
kroa150	88081	68694	19387	**59540**
krob200	95083	54690	40393	**69836**
ts225	73282	23991	49291	**59000**
gil262	57985	37936	20049	**42149**
pr264	59486	38662	20824	**41330**
pr299	65468	23400	42068	**51341**
lin318	59704	29668	30036	**44785**
rd400	48951	28194	20757	**35868**
fl417	42498	39180	3318	**35475**
pr439	39749	24107	15642	**29225**
pcb442	35716	17456	18260	**28860**

Abb. 25: Vergleich der effektiven Generationen

Bei der Variante mit Archiv ist jede Generation effektiv. Für die Berechnung der effektiven Generationen des EA ohne Archiv wird dabei einfach (*Generationen - duplicate eliminations*) verwendet. Diese Berechnung überschätzt die Anzahl der effektiven Generationen, und dennoch ist das Resultat bei allen Instanzen niedriger als bei der Variante mit Archiv.

Es ist also offensichtlich, dass ein EA mit Unterstützung des Archivs mehr Lösungen in der gleichen Zeit untersuchen kann. Das sollte sich eigentlich in den Resultaten zeigen.

Die weiteren Tests wurden mit einer mehr oder weniger finalen Version des Algorithmus gemacht. Das random ordering ist implementiert, ebenso die lokale Suche mittels ghosh1 und ghosh2 nach einem neuen globalen Optimum. Die Pop-Optimierung wird verwendet, allerdings nicht auf umgewandelte Duplikate.

				EA ohne Archiv		EA mit Archiv	
	$\|V\|$	k	Zeit (s)	$C(T)$	St.abw.	$C(T)$	St.abw.
pr299	299	60	450	**20318,1**	6,6	20336,8	23,9
lin318	318	64	450	**18515,8**	11,7	18532,3	9,2
rd400	400	80	600	5943,1	15,1	**5940,5**	9,3
fl417	417	84	600	**7982**	0	**7982**	0
pr439	439	88	600	**51791**	0	51803,3	30,2
pcb442	442	89	600	19640	34,2	**19629,2**	22
random400	400	80	600	**314,6**	14	315	12,2
random600	600	60	600	186,3	7,4	**180,2**	6,9
grouped100x5	500	100	600	**5872,2**	20,7	5892,5	29,5
grouped60x10	600	60	600	3118,6	6,3	**3118,3**	5,8
grouped80x10	800	80	600	**3187,4**	19,3	3193,9	32,1
grouped80x15	1200	80	600	**3374,2**	11,6	3425,2	47,6
grouped60x20	1200	60	600	**3285,1**	24,3	3286,7	19,5

Abb. 26: Vergleich ohne/mit Archiv, finale Version

Ein unerwartetes Resultat. In den meisten Fällen ist die Version ohne Archiv deutlich besser, ansonsten annähernd gleich gut. Dafür scheint es zunächst keine Erklärung zu geben – es wurde ja gerade gezeigt, dass die Version mit Archiv deutlich mehr Lösungen in der gleichen Zeit untersuchen kann, also warum sollen die Ergebnisse schlechter sein?
Ein weiterer Test liefert eine Erklärung: Hier die Resultate, wenn man die Pop-Repräsentation nicht verwendet.

				EA ohne Archiv		EA mit Archiv	
	$\|V\|$	k	Zeit (s)	$C(T)$	St.abw.	$C(T)$	St.abw.
kroa150	150	30	150	9872,9	65,6	**9830,3**	32,4
krob200	200	40	300	11286,4	71,4	**11282,1**	59,6
ts225	225	45	300	**62580,3**	62,2	62648,1	182,7
gil262	262	53	300	959,6	8,3	**957**	9,4
pr264	264	54	300	21936,3	45,1	**21920,6**	36
pr299	299	60	450	20435,6	61,1	**20430,3**	91,7
lin318	318	64	450	**18557,1**	37,2	18562,2	22,8
rd400	400	80	600	6001,6	31,2	**5967,3**	29,7
fl417	417	84	600	**7983,7**	1,4	7983,8	2,1
pr439	439	88	600	51997,7	113,6	**51953,9**	86,4
pcb442	442	89	600	19899,8	151,3	**19805,9**	67,6
random400	400	80	600	314	16,2	**311,2**	13,9
random600	600	60	600	196,5	14	**193,7**	14,1
grouped100x5	500	100	600	5948,7	45,3	**5945,9**	52,6
grouped60x10	600	60	600	3177,9	35,3	**3162,8**	34,9
grouped80x10	800	80	600	3319,6	47,3	**3302,5**	55,4
grouped100x10	1000	100	600	**5480,5**	77,2	5513,5	47
grouped60x20	1200	60	600	3378	26,6	**3365,7**	57,6
grouped80x15	1200	80	600	**3541,1**	76	3570,1	65,6

Abb. 27: Vergleich ohne/mit Archiv, ohne Verwendung der Pop-Repräsentation

Die Version mit Archiv liefert nun für den Großteil der Instanzen bessere Resultate. Bei den allergrößten Instanzen scheint der Effekt nachzulassen, was auch leicht zu erklären ist:

- Je mehr verschiedene Lösungen möglich sind, desto geringer die Chance, auf Duplikate zu stoßen.
- Bei großen Instanzen benötigt die Evaluierung einer Lösung mehr Zeit, was zu weniger Generationen führt, und frühere Generationen haben ja eine geringere Wahrscheinlichkeit, Duplikate zu erzeugen.

				EA mit Archiv	
	$\|V\|$	k	Zeit (s)	Generationen	converts
grouped80x10	800	80	600	24050	6558
grouped100x10	1000	100	600	17174	4055
grouped60x20	1200	60	600	68102	28713
grouped80x15	1200	80	600	24050	6558

Abb. 28: Anzahl umgewandelter Lösungen bei den größten Instanzen

Im Verhältnis zur insgesamten Anzahl an Generationen wurden hier relativ wenige Duplikate gefunden.
Wenn man es tatsächlich mit solchen Problemgrößen zu tun hat, ist es ohnehin empfehlenswert, längere Laufzeiten zu verwenden. Diese würden dann auch die Anzahl der Duplikate und damit den Effekt des Archivs erhöhen.

Der Algorithmus mit Archiv liefert also bei Verwendung der Pop-Optimierung schlechtere Lösungen als der ohne Archiv, aber ohne die Optimierung dreht sich die Situation um. Das bedeutet, die Version ohne Archiv profitiert mehr von dieser Art der Repräsentation.

Der Grund dafür ist einfach: Wird das Archiv nicht verwendet, so kann die Pop-Optimierung auf jede neu generierte Lösung angewendet werden. Auf Lösungen, die durch das Archiv umgewandelt wurden, trifft das aber nicht zu: Diese Lösungen wurden zwar ursprünglich im Rahmen der Decodierung sehr wohl auch mittels Pop optimiert. Aber während der Umwandlung wird die Lösung verändert. Wenn man die Ergebnisse betrachtet, scheint die durch die Optimierung gewonnene Verbesserung wieder verloren zu gehen. Und auf das Resultat der Umwandlung ein weiteres Mal die Pop-Optimierung anzuwenden, würde bedeuten, dass die Lösung möglicherweise wieder ein Duplikat ist.

Man könnte sich nun überlegen, ob es sinnvoll wäre, dieses Problem zu ignorieren und dennoch auf jede umgewandelte Lösung die Pop-Optimierung ein weiteres Mal anzuwenden (Die „*pop_after_convert*“-Option).

				EA ohne Archiv		EA mit Archiv		Archiv + Pop	
	$\|V\|$	k	Zeit (s)	$C(T)$	St.abw.	$C(T)$	St.abw.	$C(T)$	St.abw.
pr299	299	60	450	**20318,1**	6,6	20336,8	23,9	**20318,1**	6,6
lin318	318	64	450	**18515,8**	11,7	18532,3	9,2	18525,4	12,4
rd400	400	80	600	5943,1	15,1	5940,5	9,3	**5939,8**	6,2
fl417	417	84	600	**7982**	0	**7982**	0	**7982**	0
pr439	439	88	600	**51791**	0	51803,3	30,2	**51791**	0
pcb442	442	89	600	19640	34,2	**19629,2**	22	19638,3	30,5
random400	400	80	600	314,6	14	315	12,2	**308,4**	16,2
random600	600	60	600	186,3	7,4	**180,2**	6,9	183,3	9,8
grouped100x5	500	100	600	**5872,2**	20,7	5892,5	29,5	5879,4	22,4
grouped60x10	600	60	600	3118,6	6,3	3118,3	5,8	**3117,1**	4,7
grouped80x10	800	80	600	**3187,4**	19,3	3193,9	32,1	3196,8	26,1
grouped80x15	1200	80	600	**3374,2**	11,6	3425,2	47,6	3382,5	23,6
grouped60x20	1200	60	600	3285,1	24,3	3286,7	19,5	**3277**	15,9

Abb. 29: Ergebnisse bei Algorithmus mit Pop-Optimierung, Vergleich zwischen: Kein Archiv, Archiv ohne Pop bei umgewandelten Lösungen, Archiv mit Pop bei umgewandelten Lösungen

Die Ergebnisse sind durchgehend besser als bei der Variante mit Archiv und ohne Pop bei umgewandelten Lösungen. Insgesamt sind die Lösungen dann vergleichbar mit der Variante ohne Archiv – teilweise leicht darüber, teilweise leicht darunter.

Das ist auch leicht zu erklären, wenn man die Anzahl der Lösungen betrachtet:

	EA	EA mit Archiv		
	Generationen	Generationen	converts	Duplikate
pr299	54184	37768	4377	27302
lin318	50778	33975	2866	28190
rd400	44562	29959	4530	20395
fl417	42444	28016	219	27505
pr439	31969	21276	2394	16117
pcb442	19640	23866	3791	15112
random400	35106	27165	6837	5235
random600	44543	35464	9732	7244
grouped100x5	24179	17931	3121	10360
grouped60x10	56001	38621	5512	29230
grouped80x10	24303	18792	5034	5023
grouped80x15	17765	14938	4159	3878
grouped60x20	31729	23153	3543	15617

Abb. 30: Anzahl untersuchter und umgewandelter Lösungen bei Verwendung von „*pop_after_convert*“

Nur ein kleiner Teil der Duplikate wird „erfolgreich“ umgewandelt (ist also nach der Pop-Optimierung noch immer neu). Die restlichen Duplikate werden in neue Lösungen umgewandelt, dann durch die Pop-Optimierung wieder in Duplikate zurück konvertiert. Der Effekt des Archivs wird also stark reduziert, während die Kosten (bezüglich Laufzeit) unverändert bleiben. Die erfolgreich umgewandelten Lösungen gleichen anscheinend ungefähr die zusätzlichen Generationen der Version ohne Archiv aus – beide Varianten sind praktisch äquivalent. Der Sinn des Archivs in diesem Fall ist daher fraglich.

Es sieht also so aus, als würden sich die Optimierung mittels Pop und das Lösungsarchiv miteinander nicht vertragen. Für welche Variante soll man sich also entscheiden?

				EA+Pop, ohne Archiv		EA mit Archiv, ohne Pop	
	$\|V\|$	k	Zeit (s)	$C(T)$	St.abw.	$C(T)$	St.abw.
pr299	299	60	450	**20318,1**	6,6	20430,3	91,7
lin318	318	64	450	**18515,8**	11,7	18562,2	22,8
rd400	400	80	600	**5943,1**	15,1	5967,3	29,7
fl417	417	84	600	**7982**	0	7983,8	2,1
pr439	439	88	600	**51791**	0	51953,9	86,4
pcb442	442	89	600	**19640**	34,2	19805,9	67,6
random400	400	80	600	314,6	14	**311,2**	13,9
random600	600	60	600	**186,3**	7,4	193,7	14,1
grouped100x5	500	100	600	**5872,2**	20,7	5945,9	52,6
grouped60x10	600	60	600	**3118,6**	6,3	3162,8	34,9
grouped80x10	800	80	600	**3187,4**	19,3	3302,5	55,4
grouped80x15	1200	80	600	**3374,2**	11,6	3570,1	65,6
grouped60x20	1200	60	600	**3285,1**	24,3	3365,7	57,6

Abb. 31: Vergleich der Pop-Optimierung mit dem Archiv

Die Lösungen sind bei der Variante mit der Pop-Optimierung fast durchgehend besser, auch die Streuung ist deutlich geringer. Es scheint also im Zweifelsfall günstiger sein, diese Variante zu wählen.

Im Folgenden soll jedoch der Effekt des Archivs noch weiter untersucht werden, es wird daher die Variante mit Archiv und ohne Pop verwendet.

Nun soll untersucht werden, wie sich die Verwendung von mehreren Tries auswirkt.

				EA ohne Archiv		EA mit Archiv, 1 Trie		EA mit Archiv, 2 Tries	
	$\|V\|$	k	Zeit (s)	$C(T)$	St.abw.	$C(T)$	St.abw.	$C(T)$	St.abw.
kroa150	150	30	150	9872,9	65,6	9830,3	32,4	**9815**	0
krob200	200	40	300	11286,4	71,4	11282,1	59,6	**11275,6**	69,5
ts225	225	45	300	62580,3	62,2	62648,1	182,7	**62553,8**	114,6
gil262	262	53	300	959,6	8,3	957	9,4	**953,2**	5,3
pr264	264	54	300	21936,3	45,1	21920,6	36	**21916,1**	31,9
pr299	299	60	450	20435,6	61,1	20430,3	91,7	**20398**	65,9
lin318	318	64	450	18557,1	37,2	18562,2	22,8	**18550,1**	16,7
rd400	400	80	600	6001,6	31,2	**5967,3**	29,7	5990,2	35,6
fl417	417	84	600	7983,7	1,4	7983,8	2,1	**7983**	0,9
pr439	439	88	600	51997,7	113,6	**51953,9**	86,4	51970,8	104,7
pcb442	442	89	600	19899,8	151,3	**19805,9**	67,6	19828,5	138,7
random400	400	80	600	314	16,2	**311,2**	13,9	318,3	15,1
random600	600	60	600	196,5	14	**193,7**	14,1	200,1	15
grouped100x5	500	100	600	5948,7	45,3	**5945,9**	52,6	5957,2	45,3
grouped60x10	600	60	600	3177,9	35,3	3162,8	34,9	**3161,4**	29,2
grouped80x10	800	80	600	3319,6	47,3	**3302,5**	55,4	3344,2	39
grouped100x10	1000	100	600	**5480,5**	77,2	5513,5	47	5488,3	51,3
grouped60x20	1200	60	600	3378	26,6	**3365,7**	57,6	3381,7	26,4
grouped80x15	1200	80	600	**3541,1**	76	3570,1	65,6	3552	65,7

Abb. 32: Vergleich der Ergebnisse ohne Archiv / mit einem Trie / mit zwei Tries

Abb. 32 vergleicht die Ergebnisse ohne Archiv, mit einem Trie und mit zwei Tries. Es zeigt sich, dass die Ergebnisse bei Verwendung mehrerer Tries (was ja zu fälschlicherweise ausgeschlossenen Lösungen führen kann) nicht automatisch schlechter sein müssen. Im Gegenteil, bei den kleineren Instanzen liefert diese Variante sogar durchgehend bessere Resultate als die mit einem Trie (allerdings immer noch durchgehend schlechter als die Pop-Optimierung). Möglicherweise führt die häufigere Ausführung von convert als eine Art zusätzliche Mutation dazu, dass größere Teile des Lösungsraums abgesucht und damit lokale Minima vermieden werden.

Die Lösungsqualität leidet also nicht deutlich unter der Verwendung mehrerer Tries. Es bleibt die Frage, wie es mit dem Speicherbedarf aussieht. Interessant ist, ob der Bedarf bei Verwendung eines einzelnen Trie vertretbar bleibt und wie stark sich die Situation bei zwei Tries verändert.

				Archiv mit 1 Trie		Archiv mit 2 Tries	
	$\|V\|$	k	Zeit (s)	avg (MB)	max (MB)	avg (MB)	max (MB)
kroa150	150	30	150	14,7	16,9	3,1	3,8
krob200	200	40	300	22,7	25,9	5,7	6,4
ts225	225	45	300	32	36,2	9,6	11,5
gil262	262	53	300	21,1	23,9	7,4	8,3
pr264	264	54	300	20,4	25,2	6,6	7,2
pr299	299	60	450	29,6	32,7	9,5	11
lin318	318	64	450	29,4	32,4	9	10,3
rd400	400	80	600	38	49	13,9	15,5
fl417	417	84	600	35,3	41,5	13,6	15,3
pr439	439	88	600	29,5	33,5	11,6	13,5
pcb442	442	89	600	38,1	43,2	14,1	16,7
random400	400	80	600	41,1	46,4	17,3	19,6
random600	600	60	600	87,4	100,8	30,7	33,2
grouped100x5	500	100	600	39,5	42,6	18,2	20
grouped60x10	600	60	600	89,8	106,3	32,7	36,2
grouped80x10	800	80	600	67	71,9	33,9	36,7
grouped100x10	1000	100	600	65,5	73,2	35,9	37,8
grouped60x20	1200	60	600	180,6	199,4	70,2	81,1
grouped80x15	1200	80	600	58,5	72,9	36,2	39,7

Abb. 33: Maximale vom Trie erreichte Größe während des Laufs

Der Speicherplatzbedarf bleibt bei allen getesteten Instanzen im Rahmen. Man sieht allerdings auch, dass die Größe sehr stark von der Größe der Instanz abhängt, insbesondere von der Anzahl der Knoten pro Cluster. Die Verwendung mehrerer Tries bringt wie erwartet eine erhebliche Ersparnis, bei weiterhin guten Ergebnissen.

5 Zusammenfassung

In dieser Arbeit wurde ein evolutionärer Algorithmus zur Lösung des Generalized Minimum Spanning Tree-Problems vorgestellt. Dieser Algorithmus wurde durch zwei verschiedene Optimierungen erweitert.

Einerseits wurde ein Lösungsarchiv basierend auf einer Triestruktur verwendet, um Lösungen abzuspeichern, auf diese Art mehrfach auftretende Lösungen zu erkennen und Zeit bei der Evaluierung zu sparen. Dieses Archiv bietet zudem die Möglichkeit, erkannte Duplikate effizient in neue Lösungen umzuwandeln.

Es wurde auch untersucht, ob es möglich ist, durch das Aufspalten des Archivs in mehrere Tries Speicherplatz einzusparen, ohne dabei die Qualität der Lösungen zu stark zu beeinträchtigen. Die Ergebnisse bestätigen, dass das durchaus der Fall ist. In manchen Fällen sind die so gefundenen Lösungen sogar besser als bei Verwendung eines einzelnen Tries.

Andererseits wurden neu generierte Lösungen mit Hilfe der „Pop-Optimierung“ verbessert, die auf einer alternativen Darstellung von Lösungen basiert und aufgrund der beim Decodieren entstandenen Kanten neue Knoten generiert.

Es zeigte sich, dass beide Ansätze die Ergebnisse bei vielen Probleminstanzen verbessern. Allerdings zeigte sich ebenso, dass sich die beiden Optimierungen schlecht vertragen. Bei Anwendung der Pop-Optimierung führt die zusätzliche Verwendung des Lösungsarchivs zu schlechteren anstatt besseren Resultaten.

Insgesamt lieferte der evolutionäre Algorithmus mit der Pop-Optimierung die besten Resultate. Die dabei gefundenen Lösungswerte können mit den Resultaten aller bisherigen Ansätze für das GMST-Problem mithalten und sie teilweise übertreffen.

5.1 Mögliche zukünftige Arbeiten

Die “Unverträglichkeit” von Lösungsarchiv und Pop-Optimierung dürfte darauf zurückzuführen sein, dass das Archiv auf einer komplett anderen Lösungsrepräsentation basiert. Wenn man garantieren könnte, dass durch das Archiv umgewandelte Lösungen „Pop-optimal“ sind, wäre es vermutlich möglich, die beiden Ansätze zu kombinieren und so vielleicht noch bessere Ergebnisse zu erzielen. Es wäre deshalb interessant, eine andere Struktur für das Archiv zu suchen, basierend auf der Pop-Darstellung.

Anhang: Hinweise zur Benutzung

In diesem Abschnitt sollen die Kommandozeilenparameter erklärt werden, mit denen sich das System konfigurieren lässt. Zusätzlich zu den hier aufgeführten problemspezifischen Parametern können auch alle Standardparameter von EALIB konfiguriert werden (siehe EALIB-Dokumentation), sofern sie nicht unter „Fixe Parameter“ aufgeführt sind.

Eingabedateien

Die zu verwendenden Instanzdaten werden durch die Kommandozeilenparameter

edgefile Dateiname
clusterfile Dateiname
coordfile Dateiname

angegeben, wobei die TSPlib-Dateiformate für die Eingabedaten zu verwenden sind. Zu beachten ist, dass das System von einem vollständigen Graphen ausgeht.

Fixe Parameter

Diese Parameter müssen immer mit den angegebenen Werten verwendet werden.

eamod 0

eamod bestimmt die Art des zu verwendenden Grundalgorithmus. 0 steht für ein steady-state-EA. Da alle problemspezifischen Klassen und das Archiv unter der Annahme geschrieben wurden, dass ein steady-state-EA verwendet wird, muss dieser Parameter auf 0 gesetzt werden.

plocim 1

plocim gibt die Wahrscheinlichkeit an, dass die Funktion *locallyImprove* für ein neu erzeugtes Chromosom aufgerufen wird. Normalerweise werden hier niedrigere Werte als 1 verwendet, allerdings findet innerhalb dieser Funktion in dieser Implementation die Einbindung des Lösungsarchivs statt. Niedrigere Werte als 1 für *plocim* würden dazu führen, dass das Archiv für manche Lösungen übergangen wird (d.h. sie würden weder auf Duplikate überprüft noch in das Archiv eingefügt). Um die Wahrscheinlichkeit auf eine Verbesserung mittels

lokaler Suche zu konfigurieren, gibt es einen neuen Parameter (siehe Abschnitt 4.2, *ls_prob*).

maxi 0

maxi gibt an, ob es sich um ein Maximierungsproblem handelt (1) oder um ein Minimierungsproblem (0). Aspekte wie das Aspirationskriterium oder die Speicherung der bislang besten Lösung wurden unter der Annahme implementiert, dass es sich um ein Minimierungsproblem handelt (wie es beim GMSTP der Fall ist) und würden bei einem Maximierungsproblem daher nicht richtig arbeiten. Allerdings wäre es kein besonders großer Aufwand, diese Codeteile für einem Maximierungsproblem entsprechend abzuändern.

Konfigurierbare Parameter

Diese Parameter können frei gewählt werden, um das Verhalten des Systems zu verändern.

archive
Zulässige Werte: 0, 1 oder 2
Standard: 1
Legt fest, ob das Trie-basierte Lösungsarchiv verwendet werden soll. 1 = ja, 0 = nein. Für Testzwecke kann auch 2 verwendet werden, in diesem Fall wird das Archiv nur benutzt, um Duplikate zu zählen, sie aber nicht umzuwandeln.

numtries
Zulässige Werte: Ganze Zahlen >= 0
Standard: 0
Gibt die Anzahl der Tries an, die vom Archiv verwendet werden sollen. 0 steht für „automatisch“, wobei für Instanzen mit 100 Clustern oder weniger 1 Trie verwendet wird, bei größeren 2.

accept_duplicate
Zulässige Werte: 0-1
Standard: 0
Gibt die Wahrscheinlichkeit an, mit der eine als Duplikat erkannte Lösung trotzdem in die Population aufgenommen werden soll. Nur sinnvoll, wenn *numtries* > 1. Empfohlene Werte für diesen Fall sind 0,05-0,1.

ls_prob
Zulässige Werte: 0-1
Standard: 0
Gibt die Wahrscheinlichkeit an, mit der eine neu erzeugte Lösung mittels lokaler Suche (mit Ghosh) verbessert werden soll. Hohe Werte verlangsamen den Algorithmus erheblich.

locim_best
Zulässige Werte: 0 oder 1
Standard: 1
Gibt an, ob eine Lösung, die ein neues globales Optimum darstellt, jeweils durch lokale Suche (mit Ghosh und Ghosh2) verbessert werden soll.

ls_maxtime
Zulässige Werte: Zahlen >= 0
Standard: 20
Gibt an, wie lange eine lokale Suche maximal dauern soll (in Sekunden). Gilt für jede lokale Suche, egal ob sie durch *ls_prob* oder durch *locim_best* gestartet wurde. Bei 0 wird die Zeit nicht beschränkt.

ghosh2_maxtime
Zulässige Werte: Zahlen >= 0
Standard: 10
Gibt an, wie lange ein einzelner Durchlauf der ghosh2-Nachbarschaft maximal dauern soll (in Sekunden). Diese Nachbarschaft ist sehr umfangreich und hat deswegen einen eigenen Parameter zur Beschränkung der Zeit. Bei 0 wird die Nachbarschaft gar nicht verwendet.

locim_startgen
Zulässige Werte: Zahlen >= 0
Standard: 500
Gibt an, ab welcher Generation lokale Suche verwendet werden soll. In den ersten paar Generationen liefert die lokale Suche zwar üblicherweise deutliche Verbesserungen, braucht dazu aber auch sehr lang. In der gleichen Zeit kann das EA alleine oft bessere Ergebnisse erzielen. Deshalb gibt es dieses Limit, um lokale Suchen erst ab einem gewissen Zeitpunkt zuzulassen (wenn das EA nicht mehr so schnelle Verbesserungen liefert). Gilt für jede lokale Suche, egal ob sie durch *ls_prob* oder durch *locim_best* gestartet wurde.

use_pop
Zulässige Werte: 0 oder 1
Standard: 1
Gibt an, ob die zweite Repräsentation der Lösung basierend auf dem Pop-Ansatz (siehe Abschnitt 3.1, „Der Pop-Ansatz“) verwendet werden soll. Führt üblicherweise zu deutlich besseren Lösungen.

pop_after_convert
Zulässige Werte: 0 oder 1
Standard: 0
Gibt an, ob Lösungen, die durch *convert* generiert wurden, mit Hilfe der Pop-Darstellung optimiert werden sollen (*use_pop* muss dafür ebenfalls auf 1 gesetzt sein). Kann die Lösungen verbessern, führt aber auch dazu, dass der Effekt des Lösungsarchivs stark reduziert wird, da dadurch eine große Anzahl Duplikate entstehen kann, die dann in die Population aufgenommen werden.

conv_type
Zulässige Werte: 0 oder 1
Standard: 1
Gibt an, ob als Basislösung (siehe Abschnitt 3.3, „Finden unbesuchter Lösungen“) die beste bislang gefundene Lösung (1) oder die aktuelle Lösung selbst (0) verwendet werden soll.

ordering
Zulässige Werte: 0 oder 1
Standard: 1
Gibt an, ob die Zuordnung von Clustern zu Trieknoten auf zufällige Weise (1) oder anhand der Ebene des Knoten (0) passieren soll.

Bibliographie

[1] B. Hu, M. Leitner, G. R. Raidl (2008): "*Combining Variable Neighborhood Search with Integer Linear Programming for the Generalized Minimum Spanning Tree Problem.*" Journal of Heuristics 14 (5), 473-499.

[2] P. C. Pop (2005): *"On some polynomial solvable cases of the Generalized Minimum Spanning Tree Problem."* Proceedings of the International Conference on Theory and Application of Mathematics and Informatics ICTAMI 2005 - Alba Iulia, Romania

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

[4] P. C. Pop (2002): *"The Generalized Minimum Spanning Tree Problem."* Ph.D. thesis, University of Twente, The Netherlands.

[5] P. C. Pop, G. Still, W. Kern (2005): *"An Approximation Algorithm for the Generalized Minimum Spanning Tree Problem with Bounded Cluster Size."* Algorithms and Complexity in Durham 2005, Proceedings of the first ACiD Workshop, H. Broersma, M. Johnson, S. Szeider, eds. King's College Publications, Texts in Algorithmics (4), 115–121

[6] C. Feremans, M. Labbe, G. Laporte (2002): *"A Comparative Analysis of Several Formulations for the Generalized Minimum Spanning Tree Problem."* Networks 39 (1), 29–34

[7] C. Feremans, M. Labbe, and G. Laporte (2004): *"The Generalized Minimum Spanning Tree Problem: Polyhedral Analysis and Branch-and-Cut Algorithm."* Networks 43 (2), 71–86.

[8] D. Ghosh (2003): *"Solving Medium to Large Sized Euclidean Generalized Minimum Spanning Tree Problems."* Technical Report NEP-CMP-2003-09-28, Indian Institute of Management, Research and Publication Department, Ahmedabad, India.

[9] M. Dror, M. Haouari, J. S. Chaouachi (2000): *"Generalized spanning trees."* European Journal of Operational Research 120, 583–592.

[10] E. Ihler, G. Reich, P. Widmayer (1999): *"Class Steiner Trees and VLSI-design."* Discrete Applied Mathematics 90, 173–194.

[11] B. Golden, S. Raghavan, D. Stanojevic (2005): *"Heuristic Search for the Generalized Minimum Spanning Tree Problem."* INFORMS Journal on Computing 17 (3), 290–304.

[12] B. Hu, M. Leitner, G. R. Raidl (2005): *"Computing Generalized Minimum Spanning Trees with Variable Neighborhood Search."* Proceedings of the 18th Mini Euro Conference on Variable - 22 - Neighborhood Search, P. Hansen, N. Mladenovi´c, J. A. M. P´erez, B. M. Batista, J. M. Moreno- Vega, eds. Tenerife, Spain.

[13] G. R. Raidl, J. Gottlieb (1999): *"On the importance of phenotypic duplicate elimination in decoder-based evolutionary algorithms."* Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference, 204-211.

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

[15] S. Ronald (1994): *"Preventing diversity loss in a routing genetic algorithm with hash tagging."* R. Stonier X. H. Yu, editors, Complex Systems: Mechanism of Adaption, 133–140, Amsterdam

[16] S. Ronald (1998): *"Duplicate genotypes in a genetic algorithm."* D. B. Fogel, H.-P. Schwefel, T. Bäck, X. Yao, editors, *IEEEWorld Congress on Computational Intelligence (WCCI'98)*, 793–798, Piscataway NJ

[17] R. J. Povinelli, X. Feng (1999): *"Improving genetic algorithms performance by hashing fitness values."* Artificial Neural Networks in Engineering, 399-404, 1999.

[18] A. Šramko (2009): *„Enhancing a Genetic Algorithm by a Complete Solution Archive Based on a Trie Data Structure."* Diplomarbeit, Technische Universität Wien

[19] J. Kruskal (1956): "*On the shortest spanning subtree and the traveling salesman problem."* Proceedings of the American Mathematical Society. 7, 48–50