TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

D I P L O M A R B E I T

Ein Genetischer Algorithmus für das Optimum Communication Spanning Tree Problem

ausgeführt am Institut für
Computergraphik und Algorithmen (E186)
der
Technischen Universität Wien

unter Anleitung von
ao.Univ.Prof. Dr. Günther Raidl

durch
Günther Gruber
9425434
Leopold Figl-Gasse 504/1/7
A-3571 Gars am Kamp

Datum

Unterschrift

Zusammenfassung

Genetische Algorithmen sind Optimierungsalgorithmen, die sich für schwierige Problemstellungen mit extrem großen und komplexem Suchraum eignen. Einzelne Lösungen werden miteinander rekombiniert, um neue Lösungen zu erzeugen. Gute Eigenschaften der Elternteile werden an die Nachkommen weitergegeben, um eine optimale bzw. annähernd optimale Lösung für die Aufgabenstellung zu finden.

Das Optimum Communication Spanning Tree Problem (OCSTP) ist ein graphentheoretisches Problem. Ausgehend von einem kompletten Ausgangsgraphen mit gegebenen Distanzen und Kommunikationsbedürfnissen zwischen allen Knotenpaaren soll ein Spannbaum gefunden werden, der die Kommunikationskosten der Knoten minimiert. Die Kommunikationskosten einer Kante sind definiert als das Produkt der Summe aller Daten, welche die beiden Teilbäume miteinander austauschen, die durch diese Kante verbunden sind und der Summe der Distanzen für jeden Kommunikationspfad. Diese Problemstellung findet vor allem in Bereichen der Telekommunikation Anwendung.

Die vorliegende Arbeit befaßt sich mit Problemlösungsstrategien für das OCSTP. Es wird ein neuer Genetischer Algorithmus vorgestellt, welcher auf Edge-Sets beruht. Die Kodierung eines Spannbaums mit Hilfe von Edge-Sets – also auf der Menge der Kanten selbst – ermöglicht eine direkte Manipulation des Baumes mit Hilfe von Mengenoperatoren. Dadurch ist das Arbeiten mit dem Spannbaum einfach und erlaubt eine direkte Anwendung mathematischer Algorithmen. Es werden verschiedene lokale Heuristiken bei der Konstruktion der Spannbäume verwendet, um eine Leistungssteigerung zu erzielen. Die gefundenen Ergebnisse werden miteinander verglichen und diskutiert.

Der vorgestellt Algorithmus arbeitet zuverläßig und liefert sehr gute Ergebnisse für die verwendeten Testinstanzen. In zwei Fällen werden bessere Ergebnisse als die in der Literatur gefundenen Angaben erzielt.

Abstract

Genetic Algorithms are algorithms for optimization, especially for difficult problems with a huge and complex search space. Solutions are recombined to create new ones. Good properties of parent solutions will be inherited to their children in order to achieve optimal or near-optimal results for problem instances.

The Optimum Communication Spanning Tree Problem (OCSTP) is a graph theoretical problem. Given a complete input graph with distances and communication requirements between all pairs of nodes, a spanning tree minimizing the communication costs shall be found. The communication cost of an edge is the product of the sum of all data exchanged between the subtrees connected by this edge and the sum of distances for all communication paths. This problem appears applied especially in telecommunication applications.

This thesis deals with problem solution strategies for the OCSTP. A new Genetic Algorithm is presented, which is based on Edge-Sets. Coding a spanning tree using Edge-sets – the set of its edges – allows direct manipulation of the tree using set operations. Working with the tree is simple and a direct application of mathematical algorithmes is straight-forward. Different types of local heuristics are used to improve the performance. The found results are compared and discussed.

The presented algorithm works reliable and delivers good results for the used test instances. In two cases better results than those found in literature are achieved.

Inhaltsverzeichnis

1 Einführung

Das Design von Netzwerken ist eine grundlegende und häufig auftretende Problemstellung. Ziel ist es, ein Netzwerk aus einer Menge möglicher Verbindungen zwischen gegebenen Knoten zu finden, welches verschiedensten Beschränkungen unterworfen sein kann und wobei die Transportkosten über dieses Netzwerk minimal sind. Praktische Anwendungsgebiete sind Planung von Transporten, Kommunikationssystemen, VLSI, Wasser-/Gasversorgung und Logistik.

Das Design optimaler Kommunikationsnetzwerke war bereits Ziel intensiver Untersuchungen und Studien. Es existieren viele Varianten des Problems für unterschiedliche Einsatzgebiete mit oder ohne verschiedenen Auflagen und Einschränkungen an den Algorithmus. Es wurden bereits einige exakte und heuristische Lösungsmethoden gefunden und implementiert.

Da sich durch unterschiedliche Auflagen eines konkreten Problems immer spezielle Anforderungen an einen Lösungsalgorithmus ergeben, ist es schwierig, einen generellen Lösungsweg für alle Netzwerk-Design-Probleme zu finden. Oft erschwert eine spezielle Anforderung die Verwendung eines allgemeinen Algorithmus, da gefundene Lösungen für dieses konkrete Problem nicht verwendbar sind. Es ergibt sich somit eine schwierige Aufgabenstellung für das Design eines solchen Algorithmus. Der Algorithmus soll die jeweiligen Auflagen (constraints) beherrschen aber auch flexibel auf Änderungen der Vorgaben reagieren können, z.B. Änderung von Tarifen, Maut, Materialkosten usw.

1.1 Anwendungen und verwandte Bereiche

Das Finden eines Minimalen Spannbaum (*Minimum Spanning Tree*, MST) ist ein einfaches Netzwerk-Design-Problem. Es handelt sich hierbei um einen Baum (als Teil eines Ausgangsgraphen), welcher in bezug auf seine Wegstrecke minimal ist. Andere Einschränkungen als die Zyklenfreiheit und der Zusammenhang des Baumes gibt es in diesem Fall nicht. Hierfür existieren effiziente Algorithmen, welche eine Lösung des Problems präsentieren, zum Beispiel die Algorithmen von Kruskal [8] und Prim [9].

In der Praxis sind jedoch meist kompliziertere Aufgabenstellungen gegeben. Oft sind Einschränkungen in der Infrastruktur und andere Auflagen maßgeblich. Zum Beispiel darf ein Knoten keine beliebige Anzahl an ausgehenden oder eingehenden Kanten besitzen. Das entsprechende Problem wird als *Degree-Constraint Minimum Spanning Tree* (dc-MST) bezeichnet. Es ist dies eine Erweiterung des einfachen MST-Problems. Hier ist die maximale Anzahl an Kanten pro Knoten – der Grad des Knoten – beschränkt. Ein praktisches Beispiel für diese Problemstellung ist das Design eines Straßennetzes: Eine Kreuzung mit mehr als vier Straßen ist sehr unübersichtlich und sollte vermieden werden (von einem Kreisverkehr als Alternative natürlich abgesehen). Auch bei Computernetzen können bzw. sollen oftmals nicht mehr als eine bestimmte vorgegebene Anzahl an Datenleitungen zusammentreffen. Aufgrund der zunehmenden Komplexität der Aufgabenstellungen können Lösungen nicht mehr mit polynominellem Aufwand berechnet werden; das dc-MST-Problem ist NP-schwer. Einige heuristische Lösungsansätze zu diesem Thema werden in [10] und [11] beschrieben. Genetische Algorithmen für die Berechnung eines dc-MST werden in [12] und [13] vorgestellt.

Beim *Steiner Tree Problem* (StTP) werden die Knoten des Ausgangsgraphen in zwei Mengen unterteilt: Eine Menge mit obligatorischen Knoten, welche beim Lösungsgraphen unbe-

dingt vorhanden sein müssen, und eine Menge mit optionalen Knoten, welche nach Ermessen des Algorithmus in der Lösung verwendet werden können. Diese optionalen Knoten werden auch Steinerknoten genannt. Diese Aufgabenstellung findet im Bereich des VLSI-Designs und beim Design von Computernetzen und Telekommunikationsnetzwerken Verwendung.

Einer der bekanntesten heuristischen Lösungsansätze für das Steiner-Problem ist die Minimum Path Heuristic (MPH) von Takahashi und Matsuyama [14], welche relativ gute Lösungen liefert und schnell ist. Als weitere klassische Heuristik sei noch die Average Distance Heuristic (ADH) von Rayward-Smith [15] genannt, welche bessere Ergebnisse als die MPH liefert, aber auch mehr Zeit zum Finden dieses Ergebnisses benötigt. Esbensen [16] bietet einen Genetischen Algorithmus an, welcher viele andere Heuristiken aussticht.

Beim *Minimum Diameter Spanning Tree* (MDST) Problem ist ein Baum gesucht, bei welchem die größte Entfernung zweier Knoten minimal ist. Der Eingabegraph ist ein euklidischer Graph. Der Durchmesser eines Baum kann definiert werden als a) der längste Pfad oder auch als b) der Pfad mit den meisten Kanten. Diese Aufgabenstellung findet beim Design von Computernetzwerken Anwendung.

Ein weiteres klassisches Optimierungsproblem ist das *Travelling Salesman Problem* (TSP) [8]. Hier soll eine Reiseroute errechnet werden, mit welcher alle Knoten eines Netzwerkes genau einmal besucht werden und die Reisekosten dabei minimal sind. Als Reisekosten wird im Normalfall die zurückgelegte Wegstrecke verwendet. Eine Lösung des TSP stellt sich somit als Permutation über alle N Knoten des Eingabegraphen dar.

Das Finden eines Spannbaumes mit minimalen Kommunikationskosten für einen Graphen bei gegebenen Kommunikationsbedürfnissen zwischen allen Paaren von Knoten wurde von Hu (1974) als *Optimum Communication Spanning Tree Problem* (OCSTP) definiert (oft auch als *Minimum Communication Cost Spanning Tree Problem* bezeichnet). Hier soll nun ein Spannbaum gefunden werden, dessen Knoten mit minimalem Aufwand kommunizieren können. Das OCSTP wurde von Johnson (1978) als NP-schwierig bewiesen.

1.2 Problemdefinition

Das Optimum Communication Spanning Tree Problem ist ein Subproblem der generell als Network Design Problem beschriebenen Gruppe [1]. Von diesem generellen Problem sind alle spezielleren Aufgabenstellungen abgeleitet. Für das OCSTP wird ein Ergebnisgraph gesucht, der einen Spannbaum darstellt.

> *Optimum Communication Spanning Tree Problem* (OCSTP): Gegeben sei eine Menge von N Knoten $(n_0, n_1, ..., n_N)$ und die Distanzen (distances) $d_{u,v}$ für eine Verbindung zwischen jeweils zwei Knoten sowie der Kommunikationsbedarf (requirement) $r_{u,v}$ zwischen allen Paaren. Gesucht ist ein Spannbaum, bei welchem die Kommunikationskosten minimal sind.

Formal wird das OCST-Problem wie folgt definiert: Sei $G = (V, E)$ ein ungerichteter Graph mit der Anzahl von Knoten $n = |V|$ und der Anzahl von Kanten $m = |E|$. Der Kommunikationsbedarf der einzelnen Knoten sei mit einer $n \times n$ Bedarfsmatrix (requirement matrix)

$R = (r_{u,v})$ gegeben, wobei $r_{u,v}$ den Bedarf zweier Knoten u und v definiert. Eine $n \times n$ Entfernungsmatrix (distance matrix) $D = (d_{u,v})$ bestimmt die Entfernungen zweier Knoten u und v bzw. die Länge der Kante (u, v).

Ein Baum $T = (V, E')$, wobei $E' \subseteq E$ und $|E'| = |V| - 1$ gilt, ist ein Spannbaum von G sofern er alle Knoten des Ausgangsgraphen verbindet und frei von Zyklen ist.

Kosten der Kommunikation $C_{u,v}$ zweier Knoten u und v sind durch die Funktion

$$C_{u,v} = f(D, R)$$

definiert. Ziel ist es nun, einen Baum $T = (V, E')$ mit $E' \subseteq E$ zu finden, wobei die Gesamtsumme der Kosten minimal ist:

$$\min_{G'} \sum_{u,v \in V} C_{u,v}.$$

In der Praxis ist die Kostenfunktion $C_{u,v}$ nicht immer eine lineare Funktion, da viele Kosten von der Auswahl der Kanten abhängen. Entscheidet man sich für den Aufbau einer Verbindung, gibt es viele Optionen: Soll eine schnellere Verbindung mit mehr Kapazität verwendet werden, welche dann wiederum etwas teurer ist? Ab welcher Entfernung ist eine Leitung zu teuer oder gar technisch unbrauchbar? Weitere Aspekte sind Leitungsgebühren, Tarife, Mengenrabatte, etc. Hier wird jedoch ein vereinfachtes Kostenmodell betrachtet, welches in dieser Arbeit verwendet wird: Die Kosten einer Verbindung zweier Knoten u und v ergeben sich schlicht aus dem Produkt der Entfernung und dem erforderlichen Kommunikationsbedarf. Formal:

$$\text{cost}(T) = \sum_{u,v \in V} \left(r_{u,v} \sum_{(i,j) \in P_{u,v}(T)} d_{i,j} \right)$$

wobei $P_{u,v}(T)$ der (einzige und somit auch der kürzeste) Pfad zwischen den Knoten u und v im Baum T sei.

Ziel ist es nun, einen Spannbaum T zu finden, dessen Kosten $\text{cost}(T)$ minimal sind.

Als Beispiel sei hier die Berechnung der Kosten eines Problems mit 6 Knoten von Berry angeführt:

$$\text{Distances } D = \begin{pmatrix} 0 & 3 & 6 & 5 & 9 & 7 \\ 0 & 0 & 3 & 2 & 4 & 8 \\ 0 & 0 & 0 & 3 & 7 & 2 \\ 0 & 0 & 0 & 0 & 9 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \text{ Requirements } R = \begin{pmatrix} 0 & 5 & 13 & 12 & 8 & 9 \\ 0 & 0 & 7 & 4 & 2 & 6 \\ 0 & 0 & 0 & 3 & 10 & 15 \\ 0 & 0 & 0 & 0 & 11 & 7 \\ 0 & 0 & 0 & 0 & 0 & 12 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Der Spannbaum:

Die Kosten des Spannbaumes in Abbildung 1 betragen:

$$\begin{aligned} \text{cost}(T) &= r_{12}(d_{12}) + r_{13}(d_{12} + d_{24} + d_{46} + d_{63}) + r_{14}(d_{12} + d_{24}) + ... + r_{46}(d_{46}) + r_{56}(d_{56}) \\ &= 5(3) + 13(3 + 2 + 2 + 2) + 12(3 + 2) + ... + 7(2) + 12(1) \\ &= 534 \end{aligned}$$

Die Kosten ergeben sich somit aus der Summe der Verbindungen aller Knoten miteinander über den jeweiligen Pfad im Spannbaum (Summe der Distanzen der einzelnen Kanten).

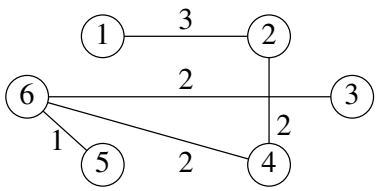


Abbildung 1: Spannbaum mit 6 Knoten

1.3 Übersicht über die weiteren Kapitel

Kapitel 2 erläutert die grundsätzliche Arbeitsweise eines Genetischen Algorithmus. In Kapitel 3 wird auf bisherige Arbeiten auf diesem Gebiet eingegangen. Im Kapitel 4 erfolgt eine Gegenüberstellung verschiedener Ansätze zur Darstellung von Bäumen. Kapitel 5 stellt einen neuen Genetischen Algorithmus für das OCSTP vor. In den weiteren Kapiteln folgt die Beschreibung der Implementierung (Kapitel 6) und die experimentellen Ergebnisse (Kapitel 7).

2 Genetische Algorithmen

Dieses Kapitel beschreibt die grundlegende Arbeitsweise eines Genetischen Algorithmus und erläutert die in diesem Zusammenhang verwendeten Begriffe und Methoden.

Genetische Algorithmen wurden von J. H. Holland [26] in den 70er Jahren entwickelt und wurden seitdem weiterentwickelt und für spezielle Probleme implementiert. Diese Art der Problemlösung basiert auf der natürlichen Evolution. Dabei wurden auch die Bezeichnungen und Begriffe der Genetik übernommen und auf den Algorithmus übertragen.

- Genetische Algorithmen sind vor allem für Probleme mit einem sehr großen, komplexen Suchraum geeignet, wodurch das Aufzählen aller Lösungen zur Optimumsuche nicht mehr bzw. nur mit einem enormen Zeitaufwand möglich ist.
- Es wird grundsätzlich das globale Maximum gesucht und nicht nur das nächstliegende lokale Maximum (globale Methode).
- Es gibt keine grundsätzlichen Einschränkungen bezüglich der zu optimierenden Funktion (z.B. Stetigkeit, Ableitbarkeit, ...)
- Mit Hilfe von Genetischen Algorithmen können nicht nur numerische Probleme gelöst werden.
- Es werden nicht unbedingt spezielle Kenntnisse und Informationen über den Suchraum benötigt.
- Genetische Algorithmen können effizient auf Parallelrechnerstrukturen implementiert werden.

Es muß aber bedacht werden, daß das Auffinden der optimalen Lösung (bzw. einer annähernd gleichwertigen) nicht garantiert werden kann.

Weiterführende Literatur zu diesem Thema ist in [27, 28, 29, 30, 31] zu finden.

Algorithm 1 Genetischer Algorithmus

```
procedure GA
begin
  t := 0;
  initialize(P(t));
  evaluate(P(t));
  while not terminate do
  begin
    t := t + 1;
    Q_s(t) := select(P(t − 1));
    Q_r(t) := recombine(Q_s(t));
    P(t) := mutate(Q_r(t));
    evaluate(P(t));
  end;
end.
```

2.1 Das Chromosom

Der erste und wichtigste Schritt beim Design eines Genetischen Algorithmus ist die Kodierung einer Lösung. Lösungen müssen durch eine Datenstruktur repräsentiert werden. Ein *Chromosom* (Individuum) bildet die Grundlage des Genetischen Algorithmus. Es repräsentiert einen Punkt im Suchraum und enthält alle relevanten Informationen zur Darstellung einer Lösung.

Ein Chromosom besteht aus *Genen*, welche die Elemente einer Lösung darstellen. Besteht das Chromosom beispielsweise aus einem Bitvektor, so stellen die einzelnen Bits die Gene des Chromosoms dar. Wird ein Spannbaum durch seine Kanten dargestellt, so entsprechen die einzelnen Kanten den Genen. Die Anzahl der Gene stellt somit ein Maß für die Größe einer Lösung dar, wonach strategische Parameter wie die Anzahl der Mutationen pro Individuum bestimmt werden können.

Die Kodierung muß so gewählt werden, daß alle möglichen Lösungen dargestellt werden können. Weiters können Hilfsdatenstrukturen aus Performancegründen verwendet werden. Parameter, mit denen das Verhalten des Algorithmus angepaßt wird und die Einfluß auf Rekombination und Mutation haben, sollten nicht im Chromosom enthalten sein. Diese sollten global gehalten werden. Da eine ganze Population von Individuuen erzeugt wird, sollte ein Individuum nicht allzu viel Speicherplatz belegen. Eine genaue Auflistung der Anforderungen an die Kodierung wird im Kapitel 4 vorgestellt und analysiert.

2.2 Der Algorithmus

Algorithmus 1 stellt den Ablauf eines Genetischen Algorithmus dar. Zu Beginn wird eine *Anfangspopulation* P für die *Generation* $t = 0$ erstellt. Diese Population wird nun bewertet. Solange die Abbruchbedingung nicht erfüllt ist, wird eine neue Generation erzeugt. Zunächst werden die bewerteten Individuen *selektiert*. Danach erfolgt Erzeugung der Nachkommen (durch *Rekombination* der selektierten Individuen) und die *Mutation*. Diese neu entstandende Population wird nun wieder bewertet bevor die Abbruchbedingung erneut überprüft wird.

Wird eine neue Population erzeugt, werden im Normalfall alle Individuen in dieser neuen

Population durch Rekombination neu erzeugt. Es besteht aber auch die Möglichkeit, nur einen bestimmten Anteil neu zu erzeugen (overlapped population). Im Extremfall ist es auch möglich, nur ein neues Individuum pro Generation neu zu erzeugen (steady-state algorithm).

Eine weitere Option ist die Verwendung von Elitismus: Dabei wird immer das beste Individuum unverändert in die neue Population übernommen. Damit ist sichergestellt, das eine einmal gefundene gute Lösung durch Rekombination oder Mutation nicht wieder verloren gehen kann.

Manchmal kann es auch nützlich sein, doppelte Lösungen zu eliminieren. Wird also durch Rekombination oder Mutation eine bereits existierende Lösung generiert, wird sie wieder verworfen. Dadurch wird ein Mindestmaß an Vielfältigkeit in der Population gewahrt und verhindert, daß sich das beste Individuum zu stark vervielfacht und der Algorithmus in weiterer Folge zu schnell terminiert.

2.2.1 Initialisierung

In der Regel wird die Anfangspopulation mit zufällig generierten Individuen erzeugt. Je nach konkreter Aufgabenstellung können jedoch bereits in diesem Stadium Heuristiken angewendet werden, um vielversprechende Lösungen zu erstellen und dem Algorithmus somit eine bessere Ausgangsposition zu verschaffen. Allerdings ist es wichtig, eine ausreichende Vielfalt in der Ausgangspopulation zu garantieren, um die vorzeitige Konvergenz zu vermeiden.

Im speziellen Fall des OCSTP empfielt es sich, bereits bei der Initialisierung möglichst minimale Spannbäume zu erzeugen, um den Algorithmus voranzutreiben.

2.2.2 Evaluierung

Jedes Chromosom muß mit Hilfe einer problemabhängigen Fitneßfunktion $f(individuum)$ bewertet werden können. Diese Funktion liefert eine Aussage über die Brauchbarkeit der Lösung. Der Genetische Algorithmus versucht, Individuen mit guter Fitneß bei der Selektion zu bevorzugen und die Population auf diese Weise zu verbessern.

2.2.3 Selektion

Die Selektion ist jener Schritt, bei dem die Elternteile für die Individuen der nächsten Generation ausgesucht werden. Dies geschieht meist per Zufallsauswahl der Individuen aus der aktuellen Population. Um aber dem Prinzip der natürlichen Auslese zu entsprechen, müssen Individuen mit besserer Fitneß eher bzw. häufiger als Eltern herangezogen werden als Individuen mit schlechter Fitneß. Die Selektion treibt somit die Individuen in Richtung besserer Lösungen.

Es existieren verschiedene Ansätze für die Selektion, von denen eine kleine Auswahl nachfolgend vorgestellt wird:

- Fitness-Proportional-Selection: Jedes Individuum erhält eine Wahrscheinlichkeit proportional zu seiner Fitneß, um für die Rekombination ausgewählt zu werden. Selektionswahrscheinlichkeit p_s für Individuum i bei Populationsgröße n:

$$p_s(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$$

Deterministische Fitneß-Proportional-Selection zur Reduktion des stochastischen Fehlers: Jedes Individuum wird entsprechend seiner Fitneß vervielfältigt. Die zu erwartende Anzahl an Nachkommen des Individuums i wird wie folgt berechnet: $e(i) = p_s(i) \cdot n = \frac{f(i)}{\sum_{j=1}^{n} f(j)} \cdot n$. Nun wird jedes Individuum $\lfloor e(i) \rfloor$ mal vervielfältigt. Um die Population zu vervollständigen werden die Individuen mit den größten Werten $e(i) - \lfloor e(i) \rfloor$ jeweils einmal selektiert (deterministic sampling). Aus diesem Pool werden dann die Elternteile für die Rekombination zufällig ausgewählt.

- Rank Selection: Die Individuen werden entsprechend ihrer Fitneß sortiert und für die Selektion wird nur die Position innerhalb dieser Reihenfolge herangezogen. Diese Variante eignet sich für Populationen, bei welcher die Verteilung der Fitneßwerte für die Fitneß-Proportional-Selection problematisch ist.

- Tournament Selection: Man erhält die Elternteile dadurch, daß man k Individuen zufällig auswählt und davon jenes mit der besten Fitneß nimmt. Diese Variante kann sehr effizient implementiert werden und eignet sich für große Populationen.

Ein wichtiger Aspekt bei der Implementierung der Selektion ist die Steuerung des Selektionsdruckes (Bevorzugung guter Individuen). Ist der Selektionsdruck zu hoch, werden gute Individuen zu stark bevorzugt. Dies führt zu einer raschen Vermehrung dieser Superindividuen und der Algorithmus konvergiert vorzeitig gegen ein lokales Optimum. Ist der Selektionsdruck wiederum zu niedrig, vermehren sich gute Individuen nur sehr langsam und schlechte Lösungen bleiben in der Population erhalten. In diesem Fall konvergiert der Algorithmus nicht oder nur sehr langsam.

In diesem Zusammenhang sei das *Scaling* erwähnt – ein Verfahren zum Anpassen der Werte der problemspezifischen Bewertungsfunktion an die Bedürfnisse der Selektion. Die Bewertungsfunktion $g(i)$ liefert für ein Individuum i Werte in einem Bereich, der an das Problem angepaßt ist, welche jedoch nicht unmittelbar für die Selektion verwendet werden können. Beispielsweise könnten negative Ergebnisse gültige oder gar bessere Lösungen für ein bestimmtes Problem darstellen. In diesem Fall wird die Fitneßfunktion $f(i) = scale(g(i))$ verwendet. $f(i)$ liefert nun Werte, die für die Selektion geeignet sind: Individuum i ist besser als Individuum j, wenn $f(i) > f(j)$ gilt.

Mit Hilfe des Scaling kann somit – beispielsweise durch eine schwächere (oder stärkere) Gewichtung höherer Fitneßwerte – Einfluß auf das Konvergenzverhalten des Algorithmus genommen werden.

2.2.4 Rekombination

Mittels Rekombination werden nun neue Individuen aus den selektierten Eltern erzeugt. Hierbei soll möglichst viel Erbmaterial der Elternteile an die Nachkommen übergeben werden. Die Rekombination wird bei GAs meist als primärer Operator gesehen, um neue Lösungen zu erzeugen. Die Rekombinationswahrscheinlichkeit p_c legt fest, mit welcher Wahrscheinlichkeit eine Rekombination durchgeführt oder das selektierte Individuum unverändert in die neue Population übernommen wird.

Abbildung 2 zeigt eine Rekombination am einfachen Beispiel eines Binär-String-Chromosoms. Dabei wird ein beliebiger Crossover-Punkt gewählt und die entstandenen Teile der beiden Elternteile (oben) neu zusammengesetzt (unten).

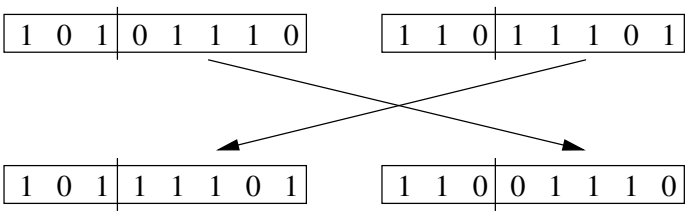


Abbildung 2: 1-point crossover

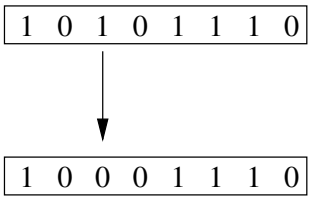


Abbildung 3: Mutation

Der Rekombinationsoperator muß für jede konkrete Problemstellung implementiert werden. Dabei ist speziell darauf zu achten, daß ein Nachkomme das Erbmaterial seiner Elternteile übernimmt und neu kombiniert. Neu hinzugekommendes Genmaterial (Eigenschaften), das nicht von den Eltern ererbt wird, sollte möglichst gering gehalten werden. Dieses Prinzip wird Lokalität genannt und ist bei den Spannbaum-Repräsentationen im Kapitel 4 detailiert beschrieben.

2.2.5 Mutation

Die Mutation wird bei GAs meist als sekundärer Operator angesehen und dient dazu, neues oder verlorengegangenes Erbmaterial wieder in die Population einzubringen. Dabei wird ein zufällig ausgewähltes Gen verändert. Die Mutation wird im allgemeinen selten angewendet. Häufig wird die Mutationswahrscheinlichkeit p_m so gewählt, daß im Erwartungsfall eine Mutation pro Chromosom durchgeführt wird: $p_m = \frac{1}{n}$ bei einer Chromosomlänge (Anzahl der Gene) von n.

Abbildung 3 zeigt eine Mutation anhand eines Binär-String-Chromosoms: Ein zufällig ausgewähltes Gen (=Bit) wird verändert.

2.2.6 Termination

Das Abbruchkriterium bestimmt den Zeitpunkt, wann der Algorithmus terminieren soll. Auch hier gibt es mehrere Möglichkeiten, von denen hier nur einige genannt werden sollen. Die Termination des Algorithmus kann erfolgen

- nach Ablauf einer bestimmten Anzahl an Generationen.
- sobald sich das beste Individuum nach einer bestimmten Anzahl an Generationen nicht mehr weiter verbessert hat.
- sobald die Population oder ein bestimmter Anteil nur noch aus identischen Individuen besteht.

- nach Ablauf einer bestimmten Zeitspanne.

Die Wahl des Terminationskriteriums ist abhängig von der gegebenen Problemstellung und sollte mit Bedacht gewählt bzw. implementiert werden.

3 Bisherige Arbeiten

Wie eingangs erwähnt wurden bereits etliche Arbeiten zu diesem umfangreichen Thema publiziert. Das Hauptproblem stellt der exponentiell große Suchraum solcher komplexer Aufgabenstellungen dar. Generell bieten sich zwei Zielfindungswege an: Die Suche des Optimums durch aufzählende Verfahren wie zum Beispiel Branch-and-Bound oder Dynamic Programming und heuristische Suchalgorithmen.

Erstere versuchen den gesamten Lösungsraum zu durchforsten und garantieren dadurch, daß die beste Lösung auch wirklich gefunden wird. Mit Hilfe der Berechnung von unteren und oberen Schranken versucht man dabei, möglichst große Teile des Suchraumes so früh wie möglich auszuschließen, damit der Suchaufwand dadurch minimiert wird. Dies ist wichtig, da für größere Problemstellungen der Suchraum exponentiell größer wird und der benötigte Rechenaufwand zum Auswerten aller möglichen Lösungen dramatisch ansteigt. Wie bei allen NP-schwierigen Problemen ist dies ein wichtiger Punkt, denn bei großen Instanzen kann es sonst sehr lange dauern, bis die beste Lösung gefunden wird.

Heuristiken hingegen versuchen, innerhalb des Suchraumes zielgerichtet auf die beste oder annähernd beste Lösung zu steuern. Man versucht, sich auf jenen Teil des Suchraums zu beschränken, in welchem die beste Lösung zu finden ist. Es ist allerdings nicht gewährleistet, daß die beste Lösung auch wirklich gefunden wird. Der Algorithmus sollte jedoch zumindest eine Lösung präsentieren, die annähernd der besten Lösung entspricht. Ein weiterer zu beachtender Aspekt ist die Verwendung einer geeigneten Heuristik. Gute Heuristiken können schneller zu einer Lösung führen. Ändert sich jedoch die Zielsetzung, so muß die Heuristik angepaßt werden, damit der Algorithmus weiterhin stabil bleibt. Wird ein Genetischer Algorithmus verwendet, muß man jedoch darauf achten, daß eine zu starke Heuristik den Algorithmus nicht zu schnell in ein lokales Optimum drängt und er dann nicht mehr in der Lage ist, die beste Lösung – also das globale Optimum – zu finden.

3.1 Einfache Heuristiken

Einfache Heuristiken können zum Finden einer guten Ausgangsbasis für weitere Optimierungen verwendet werden. Sie verfolgen einen überschaubaren Ansatz zur Lösungsfindung, sind also als Ergänzungen bzw. Startpunkt einer komplexeren Suche geeignet. Einige Vertreter wurden von Palmer [1] beschrieben und sind nachfolgend dargestellt:

Random Search: Es werden zufällige Spannbäume erzeugt und der beste Spannbaum wird verwendet. Bei 10.000 Versuchen [1] fand dieses Verfahren jedesmal das gleiche beste Ergebnis, welche das Maximum von einer Million Zufallsbäume entspricht. Palmer verwendete diese Verfahren zur Berechnung optimaler Spannbäume, um Vergleichsergebnisse für weitere Forschungen zu erhalten.

Star-Search Heuristic: Diese Heuristik erzeugt Sterne und Variationen davon. Ein Stern ist ein Spannbaum, bei dem ein Knoten das Zentrum bildet und mit dem alle anderen Knoten

durch eine Kante verbunden sind. Bei der Untersuchung von einigen realen Problemstellungen für Telefonnetzwerken und deren Lösungen [1] hat sich gezeigt, daß sternförmige Graphen eine gute Ausgangsbasis zur Lösungsfindung darstellen. Abkömmlinge bzw. Variationen von Sternen sind Bäume, welche aus mehreren Sternen bestehen - also mehrere Mittelknoten besitzen:

1. Erzeuge alle n Sterne eines Graphen und speichere das beste Ergebnis.

2. Im nächsten Schritt werden alle n^2 Bäume berechnet, die aus zwei Sternen bestehen, bei denen die Wurzelknoten über eine Kante verbunden sind. Auch hier wird der beste Baum gespeichert.

3. Wähle ein Zentrum M und erzeuge einen minimalen Spannbaum mit M als Startknoten. Nun wird eine Verbesserung dieses Baumes durchgeführt: Für alle Knoten, deren Predecessor nicht der Knoten M ist, nehme man dessen Predecessor und versuche, ihn zum Predecessor des ursprünglichen Knotens zu machen. Formell: $Pred(i) = Pred(Pred(i))$. Berechne die Fitneß und speichere das Ergebnis, falls es besser ist als der Ausgangsbaum.

4. Wähle das beste Ergebnis aus 1), 2) und 3)

Local Exchange Heuristic: Dieses Verfahren versucht, einen Baum durch das Austauschen von Kanten zu optimieren und entspricht im Wesentlichen dem unter Kapitel 3.3 vorgestellten *Tree-Improvement*.

1. Erzeuge einen Zufallsbaum

2. Für jede Kante: Entferne diese Kante aus dem Baum und bestimme die Menge der Kanten welche diese Kante ersetzen könnte, um die entstandenen Teilbäume wieder zu verbinden.

3. Setze jede Kante dieser Menge in den Baum ein und berechne das neue Ergebnis. Falls der entstandene Baum besser als alle bisher gefundenen Lösungen ist, speichere diese Lösung. Fahre bei Schritt 2) fort.

4. Vergleiche das Ergebnis mit dem bisher besten gefundenen Baum und beginne wieder bei 1). Terminiere den Algorithmus, falls keine Verbesserung mehr auftritt.

Der Aufwand eines Durchlaufes beträgt $O(n^5)$ und errechnet sich aus dem Aufwand für die Kostenberechnung des Baumes $O(n^3)$, ausgeführt für jede Kante, die ersetzt wird $O(n)$, dies wird wiederum ausgeführt für jede entfernte Kante (nocheinmal $O(n)$). Das Ganze wird nun solange wiederholt, bis keine Verbesserung des besten gefundenen Ergebnisses auftritt oder, falls technisch nicht anders möglich, eine vorher definierte Anzahl an Durchläufen erreicht ist.

Die Berechnung für ein Problem mit 98 Knoten dauerte bis zu vier Tage und länger (IBM RiscSystem/6000 model 550 and 560 workstations mit Performance von 40 und 50 MIPS).

3.2 Exakte Algorithmen

Exakte Algorithmen betrachten den gesamten Lösungsraum und versuchen das Optimum zu finden. Wie schon eingangs erwähnt, stellt der enorme Suchraum die Algorithmen vor eine große Herausforderung, da die Rechenzeit mit exponentieller Ordnung zur Problemgröße steigt. Das Auswerten aller Möglichkeiten übersteigt bei größeren Aufgabenstellungen die Grenze des Machbaren.

Beim Branch-and-Bound Algorithmus wird versucht, durch Berechnung einer lokalen Schranke die beste Lösung, welche in einem Bereich des Suchraumes gefunden werden kann, abzuschätzen. Bei Maximierungsproblemen spricht man von einer lokalen Oberschranke und bei Minimierungsproblemen von einer lokalen Unterschranke. Die globale Schranke stellt die beste zu einem Zeitpunkt gefundene Lösung selbst dar. Sollte die beste in einem Bereich auffindbare Lösung nicht besser sein als das bis zu diesem Zeitpunkt gefundene Exemplar, so kann die Suche hier abgebrochen und im nächsten Teilraum weitergeführt werden.

Die Vorgehensweise ist nun folgende: Man steht vor der Entscheidung, welcher Weg eingeschlagen werden soll. Man betrachtet nun die Alternativen und berechnet für jede davon eine Schranke, um die beste Lösung abzuschätzen. Die Alternativen werden nach der Schranke sortiert und im Speicher plaziert. Nun entnimmt man die Alternative mit der besten Schranke aus dem Speicher und ersetzt sie durch ihre Nachfolger indem man sie als Grundlage für die nächste Entscheidung herannimmt und auch für die hier erhaltenen Alternativen die Schranken berechnet. Die so bewerteten Alternativen werden wieder im Speicher abgelegt. Auf diese Weise wird das Problem Schritt für Schritt in Teilprobleme zerlegt bis man letztendlich eine Menge von Teilproblemen erhält, deren Lösungen trivial sind bzw. die verworfen werden können, weil sie sich als unbrauchbar erweisen (lokale Unterschranke $>$ globale Oberschranke).

Am anschaulichsten ist diese Vorgangsweise anhand eines weit verbreiteten Optimierungsproblemes darzustellen: dem Rucksackproblem. Hierbei handelt es sich um die Aufgabenstellung, einen Rucksack mit gegebenen Maximalgewicht auf solche Weise mit Gegenständen zu füllen, daß der Wert der eingepackten Gegenstände maximal ist. Für alle verfügbaren Gegenstände ist ihr Gewicht und ihr Wert gegeben. Das Problem besteht also aus einer Abfolge von Entscheidungen, welcher Gegenstand nun eingepackt wird und welcher nicht. Hier bietet sich ein Branch-and-Bound-Algorithmus an. Zuerst sortiert man die verfügbaren Gegenstände nach ihrem Gewicht/Wert-Verhältnis, sodaß das erste Element in der Liste jener Gegenstand ist, dessen Wert im Verhältnis zu seinem Gewicht am höchsten ist.

Für die Wurzel kann nun eine Schätzung für das beste Ergebnis berechnet werden: Man nimmt der Reihe nach die einzelnen Gegenstände und addiert deren Gewicht und Wert auf, bis das höchst zulässige Gesamtgewicht des Rucksacks erreicht ist. Dieser errechnete Wert stellt die Oberschranke des Gesamtproblems dar. Das erste Teilproblem ist nun, ob Gegenstand 1 eingepackt werden soll oder nicht. Für beide Alternativen berechnet man nun die Schranken und platziert sie in einer Priority-Queue. Wie bereits erwähnt, müssen die Teilprobleme im Speicher auf eine Weise abgelegt werden, daß man immer Zugriff auf das Element mit der höchsten Oberschranke hat. Eine Priority-Queue stellt eine Datenstruktur mit dieser erforderlichen Eigenschaft bereit.

Man entnimmt der Queue nun die Alternative mit der höchsten Schranke und betrachtet sie als nächstes und stellt sich der Entscheidung, ob Gegenstand 2 eingepackt werden soll oder nicht. Man berechnet wieder die Schranken für beide Alternativen und plaziert sie – sofern das Einpacken des Gegenstandes nicht das Maximalgewicht überschreitet – in der Queue. Auf diese Weise wurde ein Problem durch zwei weitere Teile zerlegt und der Algorithmus

beginnt von vorne. Erhält man beim Zugriff auf die Queue eine Lösung, die nicht weiter zerlegt werden kann, hat man die beste Lösung gefunden, denn die Queue enthält nur noch Lösungen, deren Schranken kleiner sind als die der eben entnommenen.

Dionne und Florian [6] präsentierten einen Algorithmus, welcher auf dem Branch-and-Bound-Prinzip basiert. Dieser Algorithmus stellt eine Modifikation des von Hoang [7] vorgestellten Ansatzes dar und betrachtet das Problem als eine Serie von Entscheidungen, welche Kante als nächstes in den Baum aufgenommen werden soll.

Ein Teilproblem des vorgestellten Algorithmus besteht also aus der Entscheidung, eine bestimmte Kante in den Baum aufzunehmen oder nicht. Dabei verwenden Dionne und Florian einen charakteristischen Vektor zur Darstellung eines Baums. Der Baum wird als binärer Vektor, dessen Länge gleich der Anzahl Kanten des Ausgangsgraphen entspricht, kodiert und eine 1 an der Stelle i bestimmt, ob die entsprechende Kante i in den Baum aufgenommen wird.

$$X = (x_1, x_2, ..., x_m) \,|\, x_k = 0 \,\text{or}\, 1, k \in M = \{1, 2, ..., m\}$$

Außerdem wird hier unterschieden, ob ein Element des Vektors variabel (noch nicht ausgewertet) oder bereits fixiert ist. Die Menge E ist eine Teilmenge des Lösungsraumes S und entspricht:

$$E = \left\{ (x_1, x_2, ..., x_m) \;\middle|\; \begin{array}{c} x_k \,\text{may take any value (either 0 or 1) for}\, k \in K \\ x_k \,\text{has a fixed value (either 0 or 1) for}\, k \in M - K \end{array} \right\}$$

K ist die Menge von Indizes der freien Variablen und $M - K$ die Menge der Indizes der fixierten Variablen. Ein Teilproblem verzeigt sich mit der Entscheidung, ob eine Kante r aufgenommen wird oder nicht:

$$E(x_r = 0) = E \cap \{(x_1, x_2, ..., x_m) \in S | x_r = 0\}$$

und

$$E(x_r = 1) = E \cap \{(x_1, x_2, ..., x_m) \in S | x_r = 1\}$$

wobei x_r die Branching-Variable darstellt. Um die Teilmenge E eliminieren zu können, muß nun eine Untergrenze LB für die Menge $\{F(X) | X \in E \,\text{and}\, X \,\text{feasible}\}$ berechnet werden. Sie wird durch Anwenden der Zielfunktion F auf das Teilproblem berechnet. Es muß dabei Rücksicht genommen werden, daß ungültige Lösungen entstehen können; In diesem Fall wird die Lösung verworfen und diese Alternative nicht weiter untersucht.

In jeder Teilmenge E sei X^pdas Netzwerk mit der größten Kantenanzahl:

$$X^p = (x_1^p, x_2^p, ..., x_m^p), \,\text{wobei}\, x_k^p = \left\{ \begin{array}{ll} 1, & \text{if}\, e_k \in E \\ 0, & \text{otherwise} \end{array} \right.$$

Für jedes $X \in E$ gilt $F(X^p) \leq F(X)$. Falls X^0die derzeit beste Lösung darstellt und $LB \geq F(X^0)$ gilt, dann kann die Teilmenge E eliminiert werden.

Für die Berechnung einer Untergrenze schlägt Hoang [7] folgende Formel vor:

$$LB = F(X^p) + \sum_{k \in K} \bar{x}_k^* f_k(X^p)$$

wobei $\bar{x} = 1 - x$ und $X^* = (x_1^*, x_2^*, ..., x_m^*)$ die optimale Lösung darstellt. Die Berechnung kann mit dem Algorithmus für das Rucksackproblem von Dantzig [23] erfolgen.

Dionne und Florian erzielten gute Ergebnisse mit diesem Algorithmus. Allerdings verwendeten sie relativ kleine Problemstellungen mit bis zu 29 Knoten. Bei größeren Problemen beansprucht der Algorithmus jedoch enorme Rechenzeit. Außerdem gingen Dionne und Florian bei ihren Untersuchungen nicht immer von kompletten Graphen aus und erlaubten auch zyklische Lösungsgraphen.

3.3 Ein exakter und ein heuristischer Algorithmus

Ahuja und Murty [3] stellen einen exakten und einen heuristischen Algorithmus für das OCSTP vor. Beide Lösungswege nehmen auf die spezielle Struktur des OCSTP Rücksicht.

3.3.1 Exakter Algorithmus

Der exakte Algorithmus ist ein Branch-and-Bound Algorithmus und für kleine und mittelgroße Probleme geeignet. Durch Optimierung der Subprobleme wird versucht, den Gesamtrechenaufwand zu minimieren.

Beim Branch-and-Bound Algorithmus wird für jede Kante entschieden, ob sie in den Baum aufgenommen werden soll oder nicht. Diese Definition des Teilproblems entspricht der Vorgehensweise in [6]. Allerdings verfolgen Ahuja und Murty eine andere Strategie bei der Lösung des Teilproblems. Die Berechnung der unteren Schranke erfolgt mit einer auf das OCSTP zugeschnittenen Methode, die Kanten in verschiedene Mengen unterteilt und damit arbeitet. Es existieren eine Kantenmenge I mit Kanten, welche in den Spannbaum inkludiert und eine Kantenmenge E mit Kanten, die ausgeschlossen (excluded) werden sollen. Weiters gibt es eine Menge U mit nicht zugeordneten Kanten (unassigned). Als erste Schätzung der unteren Grenze kann die Berechnung der kürzesten Wege über das Netzwerk bestehend aus Kanten der Mengen I und U herangezogen werden und sei $\mu_0 = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} r_{ij}u_{ij}$. Da ein OCST jedoch ein Spannbaum ist, können nicht alle Kanten aus U verwendet werden, da sonst Zyklen entstehen würden. Folglich führt das Eliminieren von Kanten aus U zu einer Verlängerung der kürzesten Wegstrecken und somit zu einer Erhöhung der unteren Grenze.

Eine Untergrenze für das OCSTP ist eine lineare Näherung des Wertes der Zielfunktion $L(\tilde{A} - X, Q)$ mit $\tilde{A} = I \cup \tilde{U}$ und $\tilde{U} \subseteq U$. Weiters sei $Q = \{(i,j) \in N \times N : 1 \leq i < j \leq n\}$, die Menge aller Knotenpaare aus N.

Für alle $(i,j) \in Q$ sei

$$\begin{aligned} \beta_{ij}^0 &= 0, \\ \beta_{ij}^k &= \max_{1 \leq p \leq k}\{\pi(a_k, (i,j))\}, \forall a_k \in \tilde{U}, \\ \omega_{ij}^k &= \max\{0, \pi(a_k, (i,j)) - \beta_{ij}^{k-1}\}, \forall a_k \in \tilde{U}. \end{aligned}$$

$h(x) = \mu_0 + \mu x$ ist eine Untergrenze für den OCST, wobei

$$\mu_k = \begin{cases} \sum_{(i,j)\in Q} r_{ij}\omega_{ij}^k, & \forall a_k \in \tilde{U}, \\ 0, & \forall a_k \in I. \end{cases}$$

daraus folgt:

$$\mu_0 + \mu x \leq L(\tilde{A} - X, Q).$$

Die Untergrenze erhält man durch Minimieren von $\mu_0 + \mu x$, sodaß $\tilde{A} - X$ ein Spannbaum ist. Durch Ersetzen $y_k = 1 - x_k, \forall a_k \in A$ erhält man:

$$\text{Minimize}\, \mu_0 + \sum_{k=1}^{m} \mu_k - \sum_{k=1}^{m} \mu_k y_k,$$

derart Y ein Spannbaum ist. Daraus folgt nun:

Wenn T^* der maximale Spannbaum von $\tilde{G} = (N, \tilde{A})$ mit μ_k als Gewicht der Kante a_k ist, dann ist $\mu_0 + \sum_{k=1}^{m} \mu_k - \sum_{a_k \in T^*} \mu_k$ eine Untergrenze für ein Teilproblem des Branch-and-Bound Algorithmus.

In [3] stellen Ahuja und Murty den Algorithmus zur Berechnung einer genauen Untergrenze dar. Der Aufwand dieser Prozedur beträgt $O(n^4)$.

Große Ausgangsgraphen erfordern jedoch sehr viel Rechenzeit. Für Aufgabenstellungen dieser Größenordnung schlagen Ahuja und Murty einen Heuristischen Algoritmus vor:

3.3.2 Heuristischer Algorithmus

Der heuristische Algorithmus ist ein Zwei-Phasen-Prozeß bestehend aus Tree-Building und Tree-Improvement. Beim Tree-Building wird mit einem Startknoten begonnen. Von diesem Knoten beginnend wird der Baum aufgebaut, indem passende Kanten und somit weitere Knoten hinzugefügt werden. Es wird immer jene Kante verwendet, welche den bestehenden Baum mit den restlichen Knoten in einer Weise verbindet, daß sich die Gesamtkosten des Baumes minimal erhöhen. Der auf diese Weise konstruierte Baum wird nun dem Tree-Improvement unterworfen. Hier wird versucht, jede Kante des Baumes durch eine andere Kante zu ersetzen, um die Gesamtkosten des Baumes zu senken.

Prinzipiell basiert der heuristische Algorithmus auf dem von Camerini, Fratta und Maffioli [5] entwickelten Verfahren (CFM Algorithmus) für einen speziellen Fall des OCST Problems. Dieser Algorithmus nimmt die Existenz einer Zahl b_i für jedes $i \in N$ an, sodaß $r_{ij} = b_i b_j, \forall (i,j), j \in N$, also für alle Knotenpaare, gilt. Der von Ahuja und Murty vorgestellte Algorithmus stellt eine Verallgemeinerung des CFM Algorithmus dar.

Tree-Building. Mit dieser Prozedur wird versucht, eine annähernd optimale Lösung zu generieren. Der Baum wird schrittweise aufgebaut, wobei immer jene Kante hinzugefügt wird, die eine Kommunikation zu den noch nicht durch Kanten verbundenen Knoten mit möglichst geringen Kosten ermöglicht.

Step 1 (Initialization). Determine all pair shortest path lengths u_{ij}'s. Set $s := \mathit{seed}$, $S := \{s\}$, $T := 0$, $\mathit{cost} := 0$, and $u^0_{ij} = 0$, $\forall (i,j) \in N \times N$.

Step 2 (Augmentation). Compute w_i and h_i for each $i \in N$ as follows:

$$w_i := \begin{cases} \sum_{j \in \overline{S}} r_{ij}, & \text{if } i \in S, \\ \sum_{j \in S} r_{ij}, & \text{if } i \in \overline{S}, \end{cases}$$

$$h_i := \begin{cases} \sum_{j \in S} w_j u^0_{ij}, & \text{if } i \in S, \\ \sum_{j \in \overline{S}} w_j u^0_{ij}, & \text{if } i \in \overline{S}. \end{cases}$$

Let $W := \sum_{i \in S} w_i$. Define a number α_{ij} for each $[i,j] \in (S \times \overline{S}) \bigcap A$ as

$$\alpha_{ij} := h_i + d_{ij}W + h_j.$$

Let $[p,q]$ be an arc satisfying

$$\alpha_{pq} = \min_{[i,j] \in (S \times \overline{S}) \bigcap A} \{\alpha_{ij}\}.$$

Set $T := T \bigcup \{[p,q]\}$.

Step 3 (Updating). Update $u^0_{iq} := u^0_{ip} + d_{pq}$ and $u^0_{qi} := u^0_{iq}$ for each $i \in S$. Update $cost := cost + \sum_{i \in S} r_{iq} u^0_{iq}$. Set $S := S \bigcup \{q\}$.

Step 4 (Termination). If T is a spanning tree, STOP; else go to Step 2.

In obiger Beschreibung ist S die Menge der Knoten im aktuellen Teilbaum T und u^0_{ij} die Länge der Pfade in T zwischen den Knotenpaaren $(i,j) \in S \times \overline{S}$, also jenen Knotenpaaren, bei welchen ein Knoten in S und der anderen nicht in S liegt. Die α_{ij} für jedes Paar $[i,j] \in (S \times \overline{S})$ geben die Kommunikationskosten aller Knoten in S mit den Knoten in $\overline{S}$ über die Kante $[i,j]$ an. Kante $[p,q]$ ist somit in bezug auf die Kosten die günstigste.

Schritte 1, 2 und 3 erfordern Rechenaufwand der Größenordnung $O(n^3)$, $O(n^2)$ und $O(n)$. Schritt 1 wird einmal ausgeführt, Schritte 2 und 3 hingegen $(n-1)$-mal. Der Gesamtaufwand der Prozedur ist somit $O(n^3)$.

Tree-Improvement. Diese Prozedur erhält als Eingabe das Ergebnis der Tree-Building Phase und untersucht jede Kante. Dabei wird diese Kante aus dem Baum entfernt und es entstehen wieder zwei Knotenmengen verbunden durch die entstandenen Teilbäume. Nun wird versucht, diese beiden Teilbäume mit einer anderen als die gerade entfernte wieder zu vereinigen. Verbessert sich die Lösung, so wird die entsprechende Kante aufgenommen und die Teilbäume wieder vereint. Die Verbesserungsprozedur wird beendet, wenn keine Verbesserung mehr eintritt.

Step 1 (Initialization). Let T be the initial spanning tree and u^0_{ij} be the distance in T between any node pair (i,j). Let $cost := \sum_{(i,j) \in Q} r_{ij} u^0_j$.

Step 2 (Selection). Examine arcs in T one by one. If all the arcs have been examined, go to Step 4; else consider an unexamined arc $[s,t]$. Let $S := \{i \in N : u^0_{is} + d_{st} = u^0_{it}\}$. For every $i \in N$ compute

$$w_i := \begin{cases} \sum_{j \in \overline{S}} r_{ij}, & \text{if } i \in S, \\ \sum_{j \in S} r_{ij}, & \text{if } i \in \overline{S}, \end{cases}$$

$$h_i := \begin{cases} \sum_{j \in S} w_j u^0_{ij}, & \text{if } i \in S, \\ \sum_{j \in \overline{S}} w_j u^0_{ij}, & \text{if } i \in \overline{S}. \end{cases}$$

Let $W := \sum_{i \in S} w_i$. Define a number α_{ij} for each $[i,j] \in (S \times \overline{S}) \bigcap A$ as $\alpha_{ij} := h_i + d_{ij}W + h_j$. Let $[p,q]$ be an arc satisfying $\alpha_{pq} = \min_{[i,j] \in (S \times \overline{S}) \bigcap A} \{\alpha_{ij}\}$. If $\alpha_{pq} < \alpha_{st}$ then go to Step 3; else repeat this step for another unexamined arc.

Step 3 (Improvement). Set $T := T \bigcup \{[p,q]\} - \{[s,t]\}$. Update $cost := cost + \alpha_{pq} - \alpha_{st}$. Update $u^0_{ij} := u^0_{ip} + d_{pq} + u^0_{qj}$ and $u^0_{ji} := u^0_{ij}$ for each $(i,j) \in S \times \overline{S}$. Go to Step 2.

Step 4 (Termination). The tree T is a locally optimum spanning tree. STOP.

Durch Entfernen der zu untersuchenden Kante zerfällt der Baum in zwei Teilbäume und die Knoten werden somit in zwei Mengen geteilt. Das Einsetzen einer Ersatzkante erfolgt wie bei der *Tree-Building*-Phase.

Der Aufwand von Schritt 2 beim Durchlauf für eine Kante beläuft sich auf $O(n^2)$. Da $(n-1)$ Kanten untersucht werden, bevor eine Verbesserung eintritt oder die Terminationsbedingung erfüllt ist, beträgt der Gesamtaufwand von Schritt 2 $O(n^3)$ pro Verbesserung. Schritt 3 benötigt $O(n^2)$ pro Verbesserung. Der Gesamtaufwand beträgt somit $O(n^3)$ pro Verbesserung des Ergebnisses.

Der mit der *Tree-Improvement*-Phase erhaltene Spannbaum ist eine lokal optimierte Lösung, die sich in der Praxis oft als sehr gute Lösung erweist. Voraussetzung für das Finden der global optimalen Lösung ist allerdings, daß eine gute Näherungslösung in der vorangegangenen *Tree-Building*-Phase erzeugt wurde. Diese wiederum hängt von der Wahl des Anfangsknotens (seed) ab. Ahuja und Murty schlagen folgende Vorgehensweise vor: Die Ebene des Netzwerkes soll in vier Quadranten unterteilt werden und innerhalb jedes Quadrants soll der Knoten mit dem höchsten Kommunikationsbedarf als Startknoten gewählt werden, also der Knoten mit $\sum_{j \in N} r_{ij}$ maximal. Nun soll der gesamte Algorithmus viermal mit den jeweiligen Startknoten durchlaufen werden. Mit dieser Methode soll eine gute Ausgangsbasis zur Findung des Optimums geschaffen werden. Die beste Lösung der vier Durchläufe stellt schließlich das Endergebnis dar.

Der Rechenaufwand beläuft sich auf $O(n^3)$ für das Tree-Building und ebenfalls $O(n^3)$ für jedes Tree-Improvement. Für große Aufgaben liefert der Algorithmus brauchbare Ergebnisse in annehmbarer Zeit.

3.3.3 Ergebnisse und Schlußfolgerung

Ahuja und Murty implementierten obige Algorithmen und verwendeten euklidische und nicht-euklidische Testinstanzen. Der Branch-and-Bound Algorithmus war in der Lage, Aufgabenstellungen mit 40 Knoten und 69 Kanten in ungefähr 40 Minuten (DEC 10 System) zu lösen. Rechenzeit und Anzahl der untersuchten Lösungen steigen exponentiell mit der Größe der Ausgangsgraphen. Spärliche Graphen sind aufgrund der einfacheren Berechnung der Untergrenze einfacher zu berechnen als dichte Graphen. Zufällig erzeugte, nicht-euklidische Aufgabenstellungen sind wiederum weitgehend einfacher bzw. schneller zu berechnen. Auch steigt der Aufwand mit wachsender Größe der Ausgangsgraphen nicht so stark an. Ahuja und Murty führen dies auf auf die große Anzahl an beinahe optimalen Lösungen bei euklidischen Problemen zurück. 90% der Rechenzeit fallen bei der Berechnung der Lower Bound an. Hier lokalisierten sie Verbesserungspotential: Bei dieser Berechnung kann auf die spezielle Struktur von dünnen Graphen eingegangen werden.

Der von ihnen vorgeschlagene heuristische Zwei-Phasen-Algorithmus ist betreffend Lösungsfindung und Rechenzeit sehr attraktiv. In der ersten Phase wird bereits eine gute Lösung gefunden, welche dann in der zweiten Phase noch optimiert wird. Bei allen 20 Testinstanzen wurde die optimale Lösung berechnet. Bei größen Graphen wurden keine Probleme bezüglich der Qualität der Lösung identifiziert. Auch bleibt die Rechenzeit bei größeren Problemen immer auf einem annehmbaren Niveau. Eine Lösung für einen Graphen mit 100 Knoten und 1000 Kanten wird in weniger als einer Minute berechnet.

Im Rahmen dieser Arbeit wurde der heuristische Algorithmus selbst sowie ein Genetischer Algorithmus mit dieser Heuristik implementiert.

3.4 Näherungsalgorithmen für Spezialfälle

Bang Ye Wu, Kun-Mao Chao und Chuan Yi Tang schlagen in [17] Näherungsalgorithmen für spezielle Problemstellungen des OCSTP vor. Das OCST-Problem mit der allgemeinen Zielfunktion $\min \sum_{u,v} \lambda(u,v)d(u,v)$ kann in das Product-Requirement Communication Spanning Tree (PROCT) und in das Sum-Requirement Communication Spanning Tree (SROCT) Problem unterteilt werden. Für beide Spezialfälle bieten sie einen Näherungsalgorithmus an.

Product-Requirement Communication Spanning Tree (PROCT): Es wird angenommen, daß jeder Knoten $v \in V$ ein Gewicht $r(v)$ besitzt und der Kommunikationsbedarf zweier Knoten u und v das Produkt dieser Gewichte $r(u)$ und $r(v)$ ist: $\lambda(u,v) = r(u) \cdot r(v)$. Die Kosten der Product-Requirement Communication (p.r.c.) sind somit definiert als: $C_p(T) = \sum_{u,v} r(u)r(v)d(u,v)$.

Stellen die Knoten des Graphen Städte dar, könnte man das Gewicht als Bevölkerung der Stadt betrachten. Der Kommunikationsbedarf zweier Städte wird als proportional zum Produkt der Bevölkerung beider Städte angenommen.

Sum-Requirement Communication Spanning Tree (SROCT): Der Kommunikationsbedarf zweier Knoten u und v wird als die Summe ihrer Gewichte $r(u)$ und $r(v)$ definiert: $\lambda(u,v) = r(u) + r(v)$. Die Kosten der Sum-Requirement Communication (s.r.c.) werden daher definiert als: $C_s(T) = \sum_{u,v}(r(u) + r(v))d(u,v)$.

Das SROCT-Problem kann in folgender Situation auftreten: Für jeden Knoten eines Netzwerkes gibt es eine Nachricht, welche an alle anderen Knoten gesendet werden soll. Die Datenmenge der Nachricht ist proportional zum Gewicht des Empfängerknotens. Unter dieser Annahme sind die Kommunikationskosten des Spannbaums T gleich $\sum_{u,v} r(v)d(u,v)$ und somit exakt die Hälfte von $C_s(T)$.

Minimum Routing Cost Spanning Tree (MRCT): Das MRCT-Problem stellt wiederum einen Spezialfall des PROCT und SROCT dar und entsteht, wenn die Gewichte aller Knoten gleich sind, zum Beispiel $r(v) = 1, \forall v \in V$.

In [17] schlagen die Autoren einen 1.577-Näherungsalgorithmus für das PROCT-Problem mit einem Aufwand $O(n^5)$ und einen 2-Näherungsalgorithmus für das SROCT-Problem mit einem Aufwand $O(n^3)$ vor. Beide Algorithmen basieren auf einem 1.577-Näherungsalgorithmus für das allgemeinere MRCT-Problem. Der Aufwand für den MRCT-Algorithmus beträgt $O(n^3)$.

3.5 Genetischer Algorithmus mit Pfad-Crossover

In [18] schlagen Li und Bouchebaba einen Genetischen Algorithmus für das OCST-Problem vor. Der Baum selbst stellt ein Chromosom des Genetischen Algorithmus dar. Die Anwendung des Rekombinations- und Mutationsoperators erfolgt also direkt auf dem Spannbaum. Dieser Algorithmus kann auch an andere graphentheoretische Optimierungsaufgaben angepaßt werden.

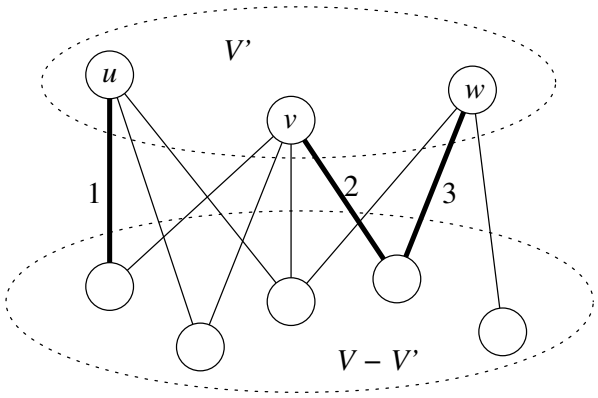


Abbildung 4: generateSpanningTree()

3.5.1 Initialisierung

Für die Erzeugung der Anfangspopulation verwenden die Autoren eine einfache Heuristik, welche auf Prim's Algorithmus für die Erstellung Minimaler Spannbäume basiert.

Sei T' ein verbundener Teilgraph eines in Konstruktion befindlichen Spannbaumes, welcher die Teilmenge von Knoten $V' \subset V$ verbindet. Das Gewicht $w(e)$ einer Kante $e = (u, v)$ sei die Distanz $d(u, v)$ der Knoten $u \in V'$ und $v \in V - V'$. Desweiteren sei $L(v) = d(u, v)$, für alle $v \in V - V'$, definiert als die kürzeste Distanz des Knotens v zu allen anderen Knoten $u \in V'$. Dies bedeutet also, daß $L(v)$ für jeden Knoten v die zu verwendende Kante zu einem Knoten u bestimmt. Der Algorithmus aktualisiert alle $L(v)$, sobald ein neuer Knoten der Menge V' hinzugefügt wird.

Bei jeder Iteration des Algorithmus wird nun eine neue Kante e zu T' hinzugefügt. Jene Kante e verbindet den zufällig gewählten Konten $u \in V'$ mit einem Knoten aus $v \in V - V'$, für den $w(e) = L(v)$ gilt. Dieser Teil ist unterschiedlich zu Prim's Algorithmus, bei welchem jene Kante gewählt wird, die einen Knoten aus $V - V'$ mit einem Knoten aus V' mit der kürzesten Distanz verbindet. Der Algorithmus terminiert, sobald $V = V'$ ist, also sobald alle Knoten verbunden sind.

Abbildung 4 zeigt ein konkretes Beispiel für das Vorgehen des Algorithmus bei der Auswahl einer Kante: Für die Knoten $u, v, w \in V'$ wurde jeweils die kürzeste Kante zu einem beliebigen Knoten aus $V - V'$ berechnet. Nun wird eine davon (Kante 1, 2 oder 3) zufällig ausgewählt und dem Baum hinzugefügt. Prim's Algorithmus hingegen würde immer die kürzeste Kante eines Knotens von V' zu einem Knoten nach $V - V'$ wählen. Algorithmus 2 stellt den Algorithmus als Pseudo-Code dar.

Dieser Algorithmus erzeugt zufällige Spannbäume mit relativ niedrigen Kosten.

3.5.2 Rekombination

Als Crossover-Operator wurde path-crossover verwendet. Bei diesem Operator werden zufällige Pfade, bestehend aus ein oder mehrere Kanten der Eltern, ausgewählt. Diese werden dann zwischen den Bäumen ausgetauscht. Auf diese Weise soll möglichst viel Genmaterial an die Nachkommen übergeben werden (Algorithmus 3).

Der Parameter $k \in [1, k_{max}]$ bestimmt die Anzahl der Pfade, welche von den Elternteilen übernommen werden sollen und muß entsprechend der Baumgröße gewählt werden. Es sollte

Algorithm 2 generateSpanningTree()

```
procedure generateSpanningTree()
begin
  randomly choose a u ∈ V
  T' := 0
  V' := {u}
  for all v ∈ (V − V') do
    L(v) := w((u, v))
  while V' ≠ V do
  begin
    randomly choose a v ∈ (V − V') and
      denote the associated edge from v to u ∈ V' by e
      for which L(v) = w(e)
    T' := T' ∪ {e}
    V' := V' ∪ {v}
    for all v ∈ (V − V') do
      if L(v) > w((v, u)) then L(v) := w((v, u))
  end
end.
```

Algorithm 3 pathCrossover()

```
pathCrossover(parent1, parent2, child1, child2)
begin
  choose a random number k
  for i = 1 to k do
  begin
    path1 = randomPath(parent1)
    path2 = randomPath(parent2)
    insertPath(path1, child2)
    insertPath(path2, child1)
  end
end.
```

Algorithm 4 insertEdge()

```
procedure insertEdge(tree, edge)
begin
  if edge = (v1, v2) is not in tree then
  begin
    path = deterministicPath(tree, v1, v2)
    delete an edge on path, which is not recently inserted
    add edge to tree
  end
end.
```

Algorithm 5 pathMutation()

```
procedure pathMutation(child)
begin
  choose a random numker k
  for i = 1 to k do
  begin
    path = randomPath(completeGraph)
    insertPath(path, child)
  end
end.
```

eine ausreichende Anzahl an Pfaden übernommen werden, damit genügend Erbinformation an die Abkömmlinge weitergegeben werden kann. Andererseits dürfen aber auch nicht zu viele Pfade eingefügt werden, da sonst viele bereits übernommene Pfade wieder zerstört werden und die Übernahme des Erbmaterials nicht effizient erfolgt. Der Wert k stellt somit einen strategischen Parameter dar, der mit Bedacht gewählt werden muß.

Das Einfügen eines Pfades in einen bestehenden Spannbaum erfolgt durch Einfügen der einzelnen Kanten des Pfades. Vor dem Einfügen einer Kante wird eine andere zufällig gewählte Kante des existierenden Pfades zwischen den beiden Knoten entfernt. Es sollte allerdings keine Kante entfernt werden, die erst gerade im Zuge der Rekombination eingefügt wurde. Dadurch wird die Bildung von Zyklen beim Einsetzen eines Pfades verhindert (Algorithmus 4).

Abbildung 5 illustriert die Übernahme zweier Pfade aus den Elternteilen in die Nachkommen bei der Rekombination mit path-crossover.

3.5.3 Mutation

Die Mutation wurde mit path-mutation verwirklicht. Hierbei wird eine Anzahl an Pfaden aus dem kompletten Graphen (Eingabegraph) zufällig ausgewählt, die dann in das Individuum eingefügt werden. Die Bildung von Zyklen wird wieder wie bei der Rekombination verhindert (Algorithmus 5 und entsprechende Abbildung 6).

Die Erzeugung eines zufälligen Pfades kann mittels einer modifizierten Tiefensuche (depth-first search) durchgeführt werden.

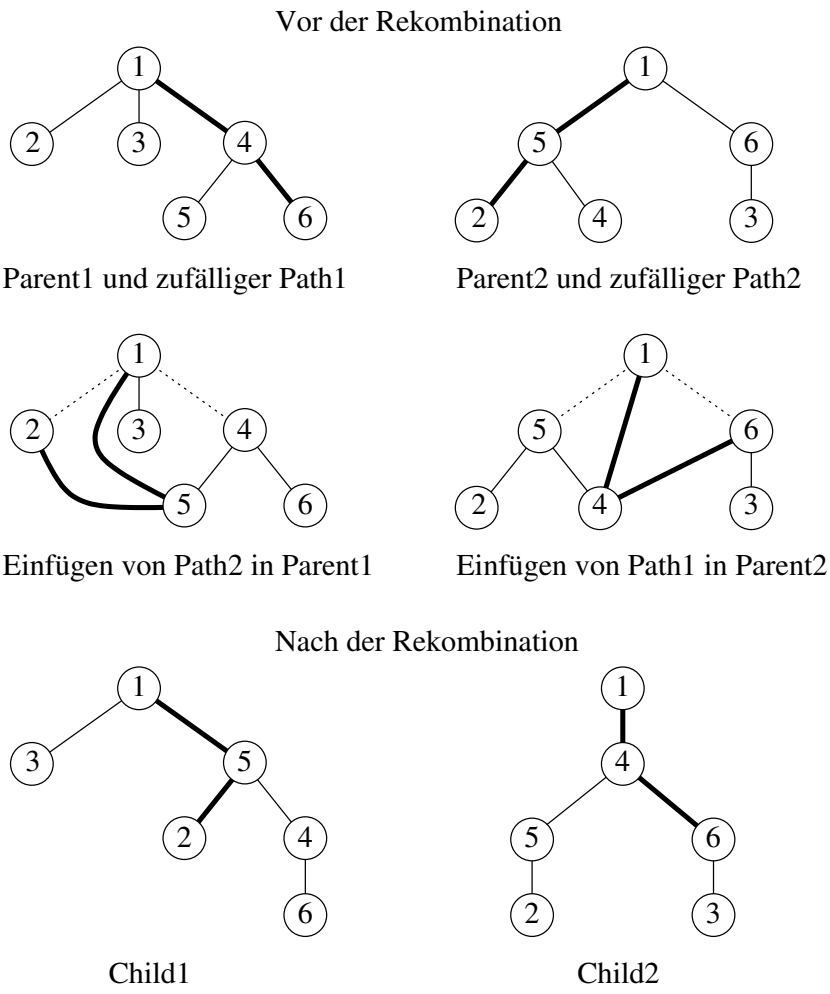
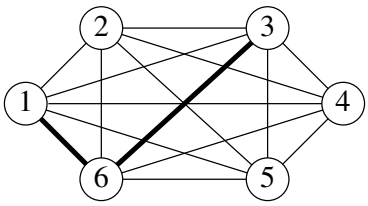
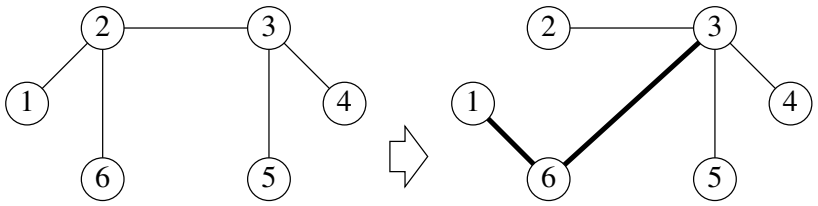


Abbildung 5: Path-crossover



Kompletter Graph and zufälliger Path1 (1–6–3)



Child Child nach dem Einfügen von Path1 (1–6–3)

Abbildung 6: Path-mutation

3.5.4 Ergebnisse und Schlußfolgerung

Der Hauptaufwand steckt im Suchen eines zufälligen oder bestimmten Pfades. Folglich ist der Gesamtaufwand der Crossover- und Mutationsoperatoren proportional zum Aufwand der Pfadsuche.

Li und Bouchebaba testeten den Genetischen Algorithmus mit fünf Beispielen, welche in Literatur zum OCST-Problem zu finden sind. Die Berechnung wurde mit folgenden Parametern durchgeführt:

- Crossover-Wahrscheinlichkeit $p_c = 0.6$
- Mutations-Wahrscheinlichkeit $p_m = 0.001$
- Populationsgröße = 200
- Maximale Anzahl an Generationen = 100

Aufgrund des Initialisierungsalgorithmus für die Erstellung der Ausgangspopulation werden relativ gute Lösungen bereits zu Beginn erzeugt. Somit waren weniger Generationen nötig, um die in der Literatur angegebenen Ergebnisse zu erzielen. In vielen Fällen wurde sogar eine bessere Lösung als die angegebene gefunden.

4 Spannbaum-Repräsentation in Evolutionären Algorithmen

Die Kodierung von Bäumen ist eine grundlegende Entscheidung beim Design eines Netzwerkes mit evolutionären Algorithmen. Die gewählte Darstellung der Graphen ist für die Anwendung

eines Algorithmus entscheidend. Nicht jeder Algorithmus kann mit einer beliebigen Darstellung bzw. Datenstruktur arbeiten. Jeder Algorithmus benötigt eigene Zustandsvariablen und Hilfsstrukturen.

4.1 Grundprinzipien

Beim Design der Datenstruktur für die Baumdarstellung gilt es, auf folgende Aspekte zu achten, die die Effizienz und Skalierbarkeit des evolutionären Algorithmus beeinflussen[24]:

- Speicherplatz: Ein Chromosom sollte nicht mehr Speicher belegen als notwendig, da eine ganze Population im Arbeitsspeicher gehalten werden muß.
- Zeitaufwand: Der Aufwand zum Kodieren und Dekodieren eines Baumes in seine Darstellung für die Rekombination, Mutation und Bewertung sollte möglichst klein gehalten werden, da diese Operatoren laufend ausgeführt werden.
- Gültigkeit: Alle Individuen, speziell die durch Rekombination und Mutation erzeugten, sollten nach Möglichkeit gültige Lösungen darstellen. Sind ungültige Lösungen darstellbar oder können ungültige Lösungen durch die Anwendung von Operatoren erzeugt werden, so muß entweder ein Reparaturalgorithmus oder eine sonstige, geeignete Maßnahme zur Behandlung ungültiger Lösungen verwendet werden.
- Abdeckung: Die gewählte Kodierung sollte alle möglichen Lösungen abdecken. Der gesamte Suchraum sollte mit der Kodierung dargestellt werden können. Nur so ist sichergestellt, daß die Operatoren des Genetischen Algorithmus alle möglichen Lösungen – und damit auch die optimale – auch erzeugen können.
- Bias: Wenn kein Wissen zu der optimalen Lösung vorhanden ist, dann sollten alle Lösungen mit der gleichen Anzahl von Darstellungen kodiert werden können. Nur auf diese Weise ist sichergestellt, daß alle möglichen Lösungen gleichwahrscheinlich erzeugt und keine Individuen bevorzugt werden. Allerdings kann gezielt eingesetztes Biasing auch Vorteile bringen, wenn dadurch gute Lösungen in den Vordergrund gebracht werden.
- Lokalität: Eine einzige Änderung eines Gens sollte keine globale Auswirkung auf die gesamte repräsentierte Lösung haben.
- Vererbung: Nachkommen sollten möglichst ausschließlich aus Eigenschaften ihrer Eltern bestehen. Es soll nicht sein, daß durch Rekombination erzeugte Nachkommen keine Ähnlichkeiten mit ihren Eltern aufweisen.
- Hybridisierung: Es sollten auf das Problem zugeschnittene Heuristiken verwendet werden können. Die Effizienz des Genetischen Algorithmus kann durch Anwendung dieser Heuristiken und Rücksichtnahme auf die Problemstruktur innerhalb der Rekombination oder Mutation bzw. als lokale Verbesserung erhöht werden.
- Randbedingungen: Die Operatoren sollten auf eventuelle problembezogene Einschränkungen Rücksicht nehmen können. Beispiele sind Gradbeschränkungen, maximale Pfadlängen und ähnliche Fälle für Spezialanwendungen.

Im Folgenden wird ein Überblick über die gebräuchlichsten Baumdarstellungen gegeben. Anschließend werden diese Kodierungen auf ihre Stärken und Schwächen miteinander verglichen.

4.2 Characteristischer Vektor

Ein Characteristischer Vektor ist ein Binär-String, dessen Bits das Vorkommen eines jeden Elements angeben. Jedes Bit gibt also Auskunft über das Vorhandensein eines Elements in einer Menge. Wenn ein Chromosom einen Graphen darstellt und die Elemente die Kanten des Graphen sind, so zeigt ein Bit das Vorhandensein einer Kante an. Der erforderliche Speicherplatz eines solchen Vektors ist proportional zur Anzahl der Elemente. Im Falle eines kompletten Graphen beträgt die Länge des Vektors $n(n-1)/2$, also die Anzahl seiner Kanten.

Stellt man Spannbäume auf diese Weise dar und verwendet naive Initialisierung, stößt man auf folgendes Problem: Es gibt $2^{n(n-1)/2}$ mögliche Chromosome, aber nur ein Bruchteil stellt gültige Spannbäume dar. Ein Graph besitzt nur n^{n-2}Spannbäume. Die Wahrscheinlichkeit, daß ein zufälliges Chromosom auch tatsächlich einen Baum darstellt, ist somit sehr gering. Rekombinations- und Mutationsoperatoren müssen daher Rücksicht auf diese Gegebenheit nehmen: Reparaturalgorithmen können aus diesen ungültigen Spannbäumen wieder gültige erzeugen, jedoch erfordert dies zusätzliche Rechenzeit und beeinflußt auch die Vererbung.

4.3 Predecessorkodierung

Eine alternative Kodierung ist die Predecessor-Darstellung für Bäume. Hierbei wird eine Wurzel r für einen Baum bestimmt. Für alle anderen Knoten i wird jeweils dessen Vorgängerknoten $Pred(i)$ im Pfad von r nach i notiert. Weiters kann immer ein bestimmter Knoten, zum Beispiel Knoten 0, die Wurzel darstellen, damit sie nicht explizit kodiert werden muß. Als Folge davon ist jeder mögliche Baum als eine einzigartige Knotenliste beschrieben.

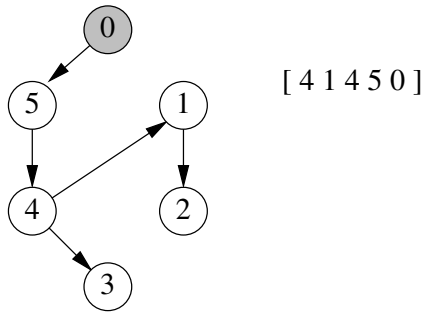


Abbildung 7: Gerichteter Baum und entsprechender Predecessor Vektor

Es gibt n^{n-1} solcher Vektoren. Da es $n^{(n-2)}$ Bäume mit dem Wurzelknoten 0 gibt, ist die Wahrscheinlichkeit, daß eine zufällig erzeugte Zahlenfolge einen Baum darstellt gleich $\frac{n^{(n-2)}}{n^{n-1}} = \frac{1}{n}$. Dies stellt zwar eine Verbesserung gegenüber dem Characteristischen Vektor dar, aber dennoch werden viele Graphen repräsentiert, die keine Bäume sind. Eine Verwendung als Chromosom mit Anwendung primitiver Operatoren ist daher nicht zu empfehlen. Der Zeitaufwand für das Kodieren eines Baumes in seine Predecessor-Darstellung beträgt $O(n)$.

4.4 Prüferzahlen

Mit Hilfe von Prüferzahlen [21] können Spannbäume speichereffizient kodiert werden. Für die Darstellung der n^{n-2} möglichen Spannbäume eines Graphen mit n Knoten wird nur ein

Vektor mit $n-2$ Stellen benötigt. Prüferzahlen können mit $O(n \log n)$ Aufwand kodiert und dekodiert werden.

Desweiteren stellt jede Prüferzahl einen gültigen Baum dar, wodurch keine Reparaturalgorithmen verwendet werden müssen. Im umgekehrten Sinne existiert für jeden Baum auch eine eindeutige Prüferzahl.

Sei T ein Baum mit n Knoten. Die Prüferzahl $P(T)$ ist ein $n-2$ stelliger Vektor, bestehend aus Ziffern zur Basis n, die durch folgenden Algorithmus definiert wird:

1. Sei i das Blatt (Knotengrad $= 1$) mit der niedrigstes Nummer in T. Sei j nun der Vorgängerknoten von i; Dann wird j die Ziffer an der äußersten rechten Stelle von P(T). Weitere Ziffern werden rechts an $P(T)$ angefügt.

2. Entferne den Knoten i und die Kante (i, j). i wird nun nicht mehr betrachtet. Falls i der einzige Nachfolgerknoten von j war, so ist j nun ein Blatt.

3. Falls nur noch zwei Knoten übrig sind, so ist $P(T)$ vollständig, ansonsten gehe zu Punkt 1.

Ein Beispiel: Knoten 2 ist das Blatt mit der niedrigsten Nummer und Knoten 3 ist sein Vorgänger. Die erste Ziffer der Prüferzahl ist somit 3 ($P(T) = 3$). Knoten 2 und die Kante (2,3) werden nun entfernt. Das Blatt mit der niedrigsten Ziffer ist somit Knoten 4. Dessen Vorgänger ist wieder Knoten 3 - die nächste Ziffer der Prüferzahl $P(T) = 33$. Knoten 4 und Kante (3,4) werden entfernt und Knoten 3 ist das Blatt mit der niedriegsten Ziffer. Knoten 1 ist sein Vorgänger - Ziffer 1 wird hinzugefügt ($P(T) = 331$). Nun ist Knoten 5 das Blatt mit der niedrigsten Nummer und dessen Vorgänger ist wiederum Knoten 1 - Ziffer 1 wird hinzugefügt ($P(T) = 3311$). Nach dem Entfernen des Knotens 5 und der Kante (1,5) sind nur noch zwei Knoten übrig - fertig.

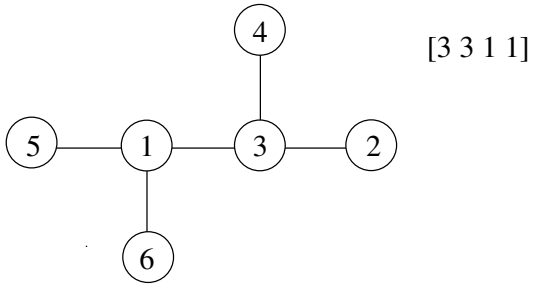


Abbildung 8: Ein Baum und seine Prüferzahl

Um eine gegebene Prüferzahl in einen Baum zu konvertieren, geht man wie folgt vor:

1. Sei $P(T)$ eine Prüferzahl und alle Knoten, welche nicht als Ziffern in der Prüferzahl erscheinen, seien Elemente der Menge V.

2. Falls die Prüferzahl keine Ziffer enthält, so gibt es exakt zwei Knoten i und j in der Menge V. Füge die Kante (i, j) zum Baum hinzu und terminiere.

3. Sei i der Knoten mit der niedrigsten Nummer in V und sei j die Ziffer an der linken äußersten Stelle von $P(T)$. Füge nun die Kante (i, j) dem Baum hinzu. Entferne die Ziffer j an der linken äußersten Stelle von $P(T)$ und entferne i aus V. Falls j nicht mehr in $P(T)$ vorkommt, füge j der Menge V hinzu.

4. Gehe zu Punkt 2.

Im Beispiel von vorhin sei $P(T) = 3311$. Die Menge V besteht aus den Knoten $\{2, 4, 5, 6\}$. Knoten 2 ist der Knoten mit der niedrigsten Nummer in V und 3 ist die linke Ziffer von $P(T)$. Kante (2,3) wird zum Baum T hinzugefügt, Knoten 2 aus V und die erste Stelle der Prüferzahl entfernt ($P(T) = 311$). Knoten 4 ist nun der nächste Knoten aus V. Folglich wird Kante (3,4) dem Baum hinzugefügt, Knoten 4 aus V entfernt und wieder die erste Stelle der Prüferzahl gelöscht ($P(T) = 11$). Knoten 3 ist nun nicht mehr in der Prüferzahl vorhanden und wird daher V hinzugefügt. Knoten 3 ist nun der Knoten mit der niedrigsten Nummer in V. Nun wird Kante (1,3) dem Baum hinzugefügt, Ziffer 1 aus $P(T)$ entfernt ($P(T) = 1$) und Knoten 3 aus V entfernt. Nun ist Knoten 5 der niedrigste in V: Kante (1,5) wird hinzugefügt, Ziffer 1 aus $P(T)$ und Knoten 5 aus V entfernt ($P(T) = leer$). Knoten 1 kommt nicht mehr in $P(T)$ vor und wird zur Menge V hinzugefügt. Da die Prüferzahl leer ist und V nur noch zwei Knoten (nämlich 1 und 6) enthält, wird Kante (1,6) dem Baum hinzugefügt und der Algorithmus terminiert. Das Ergebnis ist wieder der Baum aus Abbildung 8.

Der Nachteil von Prüferzahlen ist allerdings der Mangel an Lokalität. Obwohl ein aus Rekombination entstandener Nachkomme zweier Bäume wieder ein Baum ist, hat dieser oft wenig mit seinen Eltern gemein. Tatsächlich führt die Veränderung von nur einer Ziffer oft zu einem komplett anderen Ergebnis. Beispielsweise haben ein Baum T_1 mit 6 Knoten mit der Prüferzahl $P(T_1) = 3241$ und ein anderer Baum T_2 mit $P(T_2) = 3242$ nur zwei ihrer fünf Kanten gemeinsam.

In [22] legen die Autoren des Artikels Gründe und Untersuchungen dar, welche gegen die Verwendung von Prüferzahlen zur Kodierung von Spannbäumen in Genetischen Algorithmen sprechen. Nur Sterne und Bäume mit hoher Ähnlichkeit dazu besitzen in ihrer Prüferzahlendarstellung eine hohe Lokalität.

4.5 Link and Node Biased Kodierung

In [1] schlägt Palmer eine neue Kodierung für Bäume vor. Das Link-and-Node-Bias-Encoding (LNB) stellt eine allgemeine Baumrepräsentation dar. Diese Kodierung benötigt wenig Information über den Suchraum und ist äußerst allgemein gehalten, sodaß diese Repräsentation auch für andere Aufgabenstellungen verwendet werden kann.

Die Grundidee dabei ist, daß jeder Knoten ein Gewicht erhält, um seine Position innerhalb des Baums (Blatt oder Zentrum) zu bestimmen, da sich manche Knoten besser als Endknoten und andere sich besser als zentrale Knoten eignen.

Für jeden Knoten v_i existiert nun ein Gewicht b_i; für einen Graphen mit n Knoten existiert also ein Vektor $[b_1, b_2, ..., b_n]$. Mit diesen Gewichten wird nun die Kostenmatrix modifiziert, mit der ein einfacher Algorithmus zum Finden eines minimalen Spannbaumes arbeitet, um eine Lösung zu finden:

$$C'_{i,j} = C_{i,j} + P(b_i + b_j)C_{max}$$

Ein solcher Gewichtsvektor stellt nun das Chromosom im Genetischen Algorithmus dar. Zur Kostenberechnung des Lösungsbaumes wird wieder die originale Kostenmatrix C ver-

wendet. Der Parameter P definiert einen Faktor und C_{max}stellt das Maximum von C dar (also die teuerste Kante).

Grundsätzlich können mit diesem Ansatz nicht alle Bäume dargestellt werden. Um dieses Problem zu beheben ist erforderlich, auch die Kanten des Ausgangsgraphen mit Gewichten zu versehen. Desweiteren kann mit dieser zusätzlichen Modifikation der Tendenz des Algorithmus, vorzugsweise Sterne zu erzeugen, entgegengewirkt werden.

Ein kompletter Graph enthält $\frac{n(n-1)}{2}$ Kanten, somit besteht in dieser zweiten Kodierungsvariante das Chromosom aus $\frac{n(n+1)}{2}$ Gewichten. Der Genetische Algorithmus erhält zwei Parameter, P_1und P_2, die als Faktoren fungieren:

$$C'_{i,j} = C_{i,j} + P_1 b_{i,j} C_{max} + P_2(b_i + b_j)C_{max}$$

Parameter P_1wirkt sich auf die Kantengewichte $b_{i,j}$ aus, während P_2 Einfluß auf die Knotengewichte b_i und b_j ausübt. Die Werte für diese beiden Parameter bleiben während eines Durchlaufes unverändert. Palmer wählte $P_1 = 0$ und $P_2 = 1$ für seine Versuche. Dadurch wurde der Einfluß der Kantengewichte ausgeschaltet, um bei der Vorgangsweise aus dem ersten Ansatz zu bleiben.

Auch sollte nicht vernachlässigt werden, daß die Kantengewichte, welche im Falle $P_1 > 0$ einen Einfluß auf die Bestimmung eines Knoten (innen- oder außenliegend) hätten, im Chromosom nicht unbedingt als nebeneinander liegende Gene codiert werden. Dadurch würde ein naiver Crossover-Operator Building-Blocks mit großer Wahrscheinlichkeit verhindern.

4.6 Random Network Keys

Bean beschreibt in [19] Random Keys zur Kodierung von Permutationen. Diese Kodierung wurde von Rothlauf [20] zu Random Network Keys (RNK) erweitert, um Spannbäume darzustellen. Diese Kodierung ist eine Folge von Gleitkommawerten, die Gewichte für jede Kante repräsentieren.

Um einen Baum zu erhalten, werden die Kanten nach ihren Gewichten sortiert und an Kruskals Algorithmus zum Finden eines Minimalen Spannbaumes übergeben. Dieser verwendet die Kanten in der Reihenfolge ihrer Sortierung. Wie beim LNB-Encoding stellt hier jedes Chromosom einen gültigen Baum dar und es können Mutations- und Rekombinationsoperatoren verwendet werden.

Jedes Chromosom benötigt wegen der Sortierung der Kanten den Aufwand $O(|E| \log |E|)$. Daher sind Random Network Keys nur für kleine Aufgabenstellungen geeignet. Rothlauf [20] berichtet von guten Ergebnissen bis zu Problemstellungen für 26 Knoten.

4.7 Edge-Sets

In dieser Darstellung werden die Kanten eines Graphen als Menge kodiert. Diese natürliche Kodierung erleichtert die Verwendung der Daten in einem Algorithmus, da eine explizite Kodierung und Dekodierung entfällt. In [24] stellt Raidl diese Kodierung vor und beschreibt Operatoren und Algorithmen für diese Art der Darstellung.

Bei der Implementierung des in dieser Arbeit vorgestellten Algorithmus werden die Spannbäume mit Edge-Set-Kodierung verwirklicht. Die Anfangspopulation wird mit zufällig generierten Spannbäumen erzeugt: Es werden solange zufällig ausgewählte Kanten in den Baum eingebracht, bis alle Knoten verbunden sind und der Spannbaum somit komplett ist. Bei der

Rekombination werden die Kantenmengen der Eltern-Spannbäume geschnitten und jene in dieser Schnittmenge liegenden Kanten beim Erzeugen des Abkömmlings bevorzugt. Sollte sich hierbei kein kompletter Spannbaum ergeben, so wird er mit den restlichen Kanten der Eltern und den Kanten des Ausgangsgraphen vervollständigt. Diese intuitive Vorgehensweise beim Arbeiten mit Graphen ist sehr gut mit der mathematischen Mengenschreibweise darstellbar und kann daher bei der Implementierung unmittelbar angewandt werden. Die Abbildung des mathematisch beschriebenen Algorithmus kann sozusagen direkt bei der Kodierung des Algorithmus in einer Programmiersprache erfolgen.

Für die Implementierung ist jedoch eine Datenstruktur erforderlich, die den Aufwand zum Suchen, Einfügen und Löschen der Elemente optimiert. Dies kann mit Hilfe einer Hashtabelle erfolgen; In diesem Fall ist der Aufwand für diese Operationen im Erwartungsfall konstant und der benötigte Speicher ist proportional zur Größe des Graphen, sofern geeignete Operatoren verwendet werden.

Ein großer Vorteil dieser Kodierung ist die hohe Lokalität. Wenn eine Kante aus dem Baum entfernt und durch eine andere Kante ersetzt wird, verändert sich nicht der komplette Baum. Die durch die Entfernung einer Kante entstandenen Teilbäume werden lediglich durch das Hinzufügen einer anderen Kante wieder verbunden. Bei der Mutation wird dadurch eine Kante, die aufgrund schlechter Fitneß ausgeschieden ist, wieder eingebracht, ohne den Baum dadurch komplett neu zu erzeugen.

Die Rekombination kann Rücksicht auf die Übernahme des Genmaterials aus den Elternelementen nehmen und Kanten, welche in beiden Eltern vorkommen, beim Erzeugen der Nachkommen bevorzugen. Es ist nötig, einen neuen Rekombinationalgorithmus zu entwickeln, da beispielsweise 1-point crossover nicht auf Mengen angewendet werden kann. Da hierbei der Nachkomme komplett neu (mit dem Erbmaterial der Elten) aufgebaut werden muß, kann dabei auf Heuristiken zurückgegriffen werden, um auf die Problemstellung zu reagieren.

Zum Erstellen der Ausgangspopulation für den Genetischen Algorithmus ist eine Vorgehensweise erforderlich, die gleichwahrscheinlich alle möglichen Bäume aus einem Eingabegraphen erzeugt. Dennoch können, abhängig vom Ziel, bestimmte Typen wie Sterne oder dergleichen bevorzugt werden, wenn sich dadurch eine bessere Ausgangsbasis für den Algorithmus ergibt. Diese Bevorzugung von Darstellungen einzelner Typen muß jedoch sehr genau untersucht werden und darf nicht blindlings ohne Kenntnisse der Struktur des Lösungsraumes erfolgen. Falsche Annahmen bei der Erstellung der Ausgangspopulation können das Finden optimaler Lösungen beeinträchtigen oder im schlimmsten Fall sogar verhindern. In [25] wurde gezeigt, daß sich Minimale Spannbäume besser als zufällig erzeugte Spannbäume eignen, um das Optimum Communication Spanning Tree Problem anzugehen.

4.8 Zusammenfassung und Vergleich

Tabelle 1 zeigt die oben im Detail beschriebenen Darstellungen hier noch einmal im Vergleich in bezug auf Feasibility, Coverage, Bias, Locality/Heritability, Constraints sowie Sparse Graphs.

In Tabelle 2 sind die Aufwände in Speicher und Rechenzeit der einzelnen Kodierungen gegenübergestellt, wobei gilt: $n = |V|$, $m = |E|$.

Darstellung	Gültigkeit	Abdeckung	Bias	Lokal./Vererb.	Randbed.	Hybrid.	Dünne G
Char. Vektor	schlecht	ja	nein	hoch	mittel	gut	gut
Predecessor	mäßig	ja	nein	hoch	mittel	gut	schlecht
Prüferzahlen	ja	ja	nein	niedrig	schlecht	schlecht	schlecht
LNB	ja	ja	nein	mittel	gut	gut	gut
RNK	ja	ja	niedrig	hoch	gut	möglich	gut
Edge-Sets	ja	ja	abhängig	am höchsten	gut	gut	gut

Tabelle 1: Eigenschaften von Baumkodierungen

Darstellung	Speicheraufwand	Zeitaufwand
Characteristischer Vektor	$O(m)$	$O(m)$
Predecessor-Kodierung	$O(n-1)$	$O(n)$
Prüferzahlen	$O(n-2)$	$O(n \log n)$
Link-and-Node Biased	$O(m+n)$	$O(m+n \log n)$
Random Network Keys	$O(m)$	$O(m \log m)$
Edge-Sets	$O(n)$	$O(n)$

Tabelle 2: Zeit- und Speicheraufwand von Baumkodierungen

5 Ein Genetischer Algorithmus für das OCSTP

Zu Beginn werden alle Individuen (= Lösungen) der Population initialisiert (siehe Abschnitt 5.1). Die Rekombination erfolgt mit einem der in Abschnitt 5.2 beschriebenen Crossover-Verfahren. Es wurden mehrere heuristische sowie ein nicht heuristisches Verfahren für die Kantenauswahl beim Crossover implementiert.

Es wird ein Steady State Algorithmus verwendet. In jeder Generation wird also nur ein neues Individuum erzeugt, welches das schlechteste Individuum in der Population ersetzt. Das jeweils beste Individuum wird niemals durch einen Nachkommen ersetzt (Elitismus).

Des weiteren werden neu erzeugte Lösungen nicht in die Population aufgenommen, wenn es sich um Duplikate von existierenden Individuen handelt. Damit wird der Erhalt einer Mindestvielfalt an Lösungen in der Population unterstützt.

5.1 Initialisierung

Alle Individuen der Anfangspopulation werden initialisiert. Die Größe der Population wird in der Konfiguration des Genetischen Algorithmus festgelegt. Es handelt sich hierbei um eine zufällige Initialisierung.

1. Der neue Baum wird mit einer leeren Kantenmenge initialisiert. Weiters wird eine Liste aller Kanten des Eingabegraphen generiert, aus welcher Kanten für den Spannbaum ausgewählt werden.

2. Aus den verfügbaren Kanten des Eingabegraphen wird eine Kante zufällig ausgewählt und zum Baum hinzugefügt, sofern die beiden Knoten, die er verbindet, nicht bereits über andere Kanten des Baumes verbunden sind. Die betrachtete Kante wird aus der

Algorithm 6 initialize()

```
procedure TREE::initialize(GRAPH G)
begin
  E' :=local copy of G's edge list E
  T := {};
  while |T| < |V| − 1 do
  begin
    randomly select e ∈ E' and remove e from E';
    decode e = (n1, n2);
    if n1 and n1 not already connected in T then
    begin
      T := T + {e};
    end;
  end;
end.
```

Liste der verfügbaren Kanten entfernt, um zu verhindern, daß diese Kante nochmals ausgewählt und in Betracht gezogen wird.

3. Falls der Spannbaum ($|V| - 1$ Kanten) nicht vollständig ist, gehe zu (2), ansonsten beende.

Der Algorithmus weist einen maximalen Aufwand von $O(|E|) = O(n^2)$ auf, da maximal alle Kanten des Ausgangsgraphen einmal überprüft werden, bis der Spannbaum erzeugt ist.

Zur Verwaltung der Knoten wird eine Union-Find-Datenstruktur verwendet, welche die Überprüfung, ob zwei Knoten bereits verbunden sind, sowie das Hinzufügen neuer Verbindungen – also die Vereinigung von Knotenmengen – übernimmt. Eine Operation auf dieser Datenstruktur (Hinzufügen einer Kante, Überprüfung auf Verbundenheit) hat den Aufwand $O(\alpha(n^2, n))$, wobei $\alpha(M, N)$ die inverse Ackermannfunktion ist. Ihr Wert erhöht sich mit wachsendem M bzw. N nur extrem langsam. So ist $\alpha(M, N) \leq 4$ für alle praktisch auftretenden Werte von M und N (solange $M \leq 10^{80}$). Parameter M ist die Anzahl der Operationen auf der Datenstruktur (in unserem Falle n^2) und N stellt die Anzahl der Elemente dar (also n selbst). Die gemittelte Laufzeit innerhalb eines Algorithmus ist praktisch konstant. Die Implementierung der Union-Find-Datenstruktur ist in 6.2.1 beschrieben.

Algorithmus 6 stellt die Initialisierung im Pseudo-Code dar.

5.2 Crossover

Es wurden ein nicht heuristisches und mehrere heuristische Verfahren für die Kantenauswahl beim Crossover implementiert. Das für einen Durchlauf zu verwendende Verfahren wird beim Start des Algorithmus als Parameter mitgegeben.

5.2.1 Keine Heuristik

Bei dieser Methode wird der Spannbaum mit zufällig ausgewählten Kanten aus verschiedenen Kantenmengen der Eltern aufgebaut:

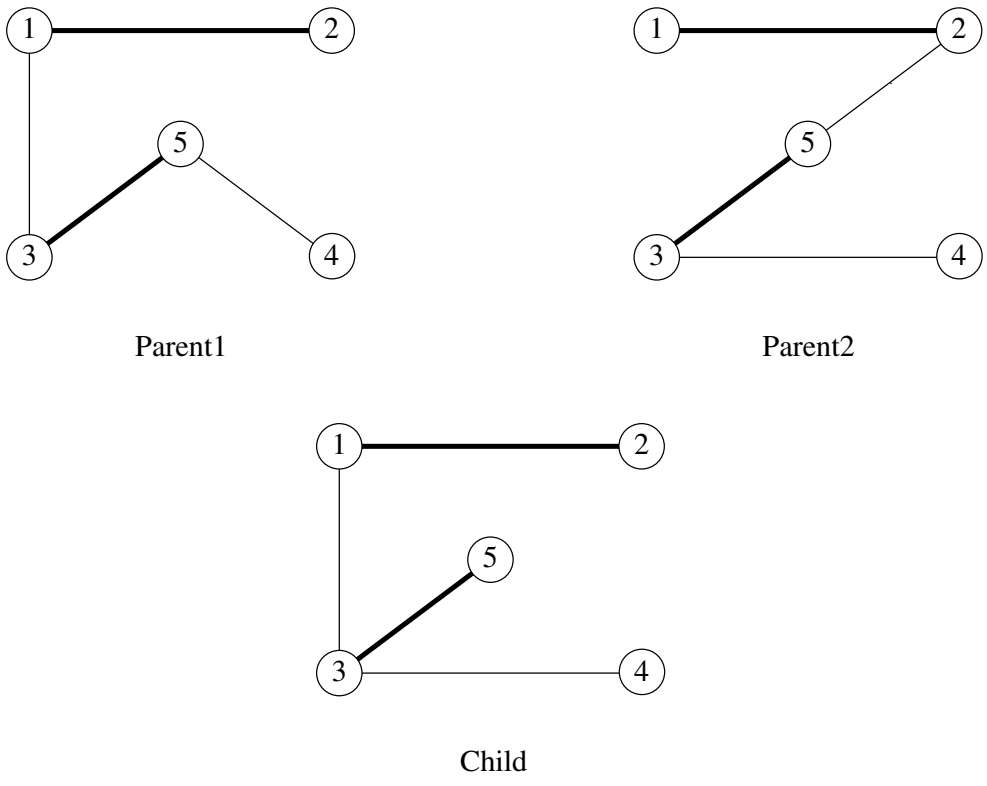


Abbildung 9: Crossover ohne Heuristik

1. Alle Kanten, welche in beiden Elternteilen vorkommen ($A \cap B$), werden unverändert in das Kind übernommen.

2. Der Spannbaum wird mit Kanten, welche in einem der beiden Elternteile vorkommen ($A \cup B - A \cap B$), vervollständigt.

Hierbei werden also Kanten bevorzugt, welche in beiden Elternteilen vorkommen, bevor auf die übrigen Kanten der Elternteile zurückgegriffen wird. Es ist nicht erforderlich, auf andere Kanten des Ausgangsgraphen zurückzugreifen, da mit den Kanten der Elternteile mit den Schritten 1 und 2 ein gültiger Baum erzeugt wird. Dies muß berücksichtigt werden, da durch die Rekombination kein neues Genmaterial in die Population eingebracht wird. Nicht in den Elternteilen vorkommende Kanten können somit nur mit Hilfe der Mutation wieder eingefügt werden.

Abbildung 9 illustriert dieses Verfahren: Zunächst wird der Abkömmling mit Kanten, welche in beiden Elternteilen vorkommen, aufgebaut (dicke Kanten). Danach wird der Baum mit Kanten, die nur in einem der Elternteile verwendet wurden, vervollständigt (dünne Kanten).

5.2.2 Heuristik

Zunächst wird eine Menge von Kanten, welche in mindestens einem Elternteil vorkommen, erzeugt ($A \cup B$). Nun wird versucht, aus dieser Menge den neuen Spannbaum zu generieren.

Zur Auswahl der Kanten wird eine Tournament-Selektion verwendet. Bei jedem Schritt werden k_{sel} Kanten zufällig selektiert und mit Hilfe einer Heuristik wird bestimmt, welche dieser Kanten nun tatsächlich zur Konstruktion des Baumes verwendet wird. Hier wird die

Unterscheidung vorgenommen, welches spezielle Crossover-Verfahren gewählt wurde (siehe nachfolgende Abschnitte). Die Tournament-Selektion erlaubt es, durch das Festlegen der zu verwendenden Kantenanzahl k_{sel} den Einfluß der Heuristik zu bestimmen. Das Verfahren ist effektiv und arbeitet mit linearem Aufwand pro hinzugefügter Kante. Der Gesamtaufwand bei der Erzeugung eines neuen Baumes beläuft sich somit auf $O(n)$.

Seien s, t, u, v Knoten und $e_{s,t}, e_{u,v}$ zwei zufällige Kanten aus $A \cup B$. Außerdem wird $k_{sel} = 2$ angenommen. Für alle $k_{sel} > 2$ gilt: Es wird das Minimum bzw. Maximum über alle k_{sel} Kanten bestimmt und die beste Kante wird verwendet.

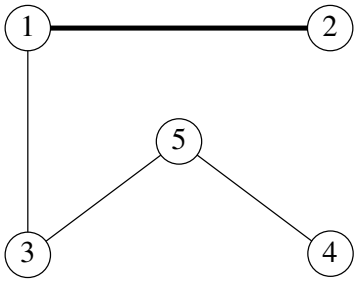
1. *Heuristik 1*: Bei dieser Heuristik ist $d_{s,t}$ – also die Distanz zweier Knoten – das Auswahlkriterium. Diese Heuristik bevorzugt Kanten mit geringer Distanz.
   Es wird Kante $e_{s,t}$ verwendet, wenn gilt: $d_{s,t} < d_{u,v}$.

2. *Heuristik 2*: Hier ist $r_{s,t}$ – also der Kommunikationsbedarf zweier Knoten – das Auswahlkriterium. In diesem Fall geht man davon aus, daß Knoten, welche sehr umfangreich miteinander kommunizieren, mit größerer Wahrscheinlichkeit verbunden werden sollten.
   Es wird Kante $e_{s,t}$ verwendet, wenn gilt: $r_{s,t} > r_{u,v}$.

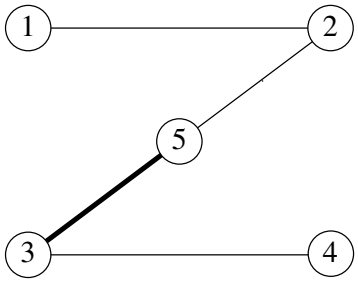
3. *Heuristik 3*: Diese Variante verwendet die Distanz und den Kommunikationsbedarf zur Auswahl einer Kante. Die Distanz wird mit dem Kommunikationsbedarf in Relation gesetzt, um Kanten mit großer Länge aber auch großem Bedarf bessere Chancen gegenüber Kanten mit kleiner Länge und niedrigem Bedarf einzuräumen. Es wird der mittlere Kommunikationsbedarf verwendet, da dieser einen von der Größe der Aufgabenstellung abhängigen Bezugswert darstellt und somit bei jeder Testinstanz verwendet werden kann.
   Sei R ist der mittlere Kommunikationsbedarf $\frac{\sum_{s,t \in V(G)} r_{s,t}}{|G(V)|}$.
   Als Auswahlkriterium für eine Kante dient: $\frac{d_{s,t}}{r_{s,t}+R}$.
   Es wird Kante $e_{s,t}$ verwendet, wenn gilt: $\frac{d_{s,t}}{r_{s,t}+R} < \frac{d_{u,v}}{r_{u,v}+R}$.
   Die Addition des Kommunikationsbedarfs $r_{s,t}$ mit dem mittleren Kommunikationsbedarf R des Graphen bewirkt eine Abschwächung des Einflußes von $r_{s,t}$ gegenüber dem Einfluß von $d_{s,t}$ beim Vergleich mit einer anderen Kante.

4. *Heuristik 4*: In dieser Variante wird die mittlere Distanz aller Knoten berechnet und bei der Auswahl einer Kante als Entscheidungshilfe herangezogen. Es werden somit Kanten bevorzugt, die großen Kommunikationsbedarf haben, auch wenn sie über eine große Länge verfügen. Auch hier bewirkt die Addition der durchschnittlichen Distanz D eine Abschwächung des Einflusses der Distanz im Verhältnis zum Kommunikationsbedarf.
   Sei D ist die mittlere Distanz zweier Knoten $\frac{\sum_{s,t \in V(G)} d_{s,t}}{|G(V)|}$.
   Als Auswahlkriterium für eine Kante dient: $\frac{r_{s,t}}{d_{s,t}+D}$.
   Es wird Kante $e_{s,t}$ verwendet, wenn gilt: $\frac{r_{s,t}}{d_{s,t}+D} > \frac{r_{u,v}}{d_{u,v}+D}$.

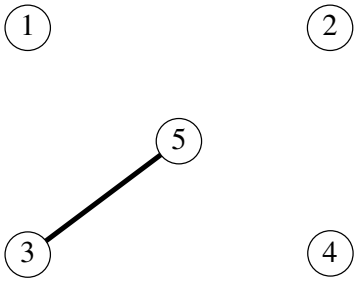
5. *Heuristik 5*: Bei dieser Heuristik wird sowohl der mittlere Kommunikationsbedarf aus (3) als auch die mittlere Distanz aus (4) verwendet. Eine Kante wird aufgrund des Distance-Requirement-Verhältnisses ausgewählt.
   Als Auswahlkriterium für eine Kante dient: $\frac{d_{s,t}+D}{r_{s,t}+R}$.
   Es wird Kante $e_{s,t}$ verwendet, wenn gilt: $\frac{r_{s,t}+R}{d_{s,t}+D} > \frac{r_{u,v}+R}{d_{u,v}+D}$.



Parent1: Kante (1,2) ausgewählt



Parent2: Kante (3,5) ausgewählt



Child: Kante (3,5) wurde hinzugefügt

Abbildung 10: Crossover mit Heuristik

6. *Zufällige Heuristik*: Für jede Kantenauswahl wird zufällig (gleichwahrscheinlich) eine der obigen Heuristiken verwendet.

7. *Heuristik nach Ahuja und Murty*: Die Heuristik nach Ahuja und Murty, welche in Kapitel 3.3.2 beschrieben ist, wurde im Rahmen dieser Arbeit ebenfalls implementiert. Damit diese Heuristik für den Genetischen Algorithmus verwendbar ist, mußte er modifiziert werden, um das Erbmaterial der Elternteile zu berücksichtigen. Hierbei wird wie folgt vorgegangen: Wird ein Nachkomme erzeugt, so werden in erster Linie die Kanten der beiden Elternteile verwendet. Wird hier keine geeignete Kante gefunden, wird auf die restlichen Kanten des Eingabegraphen zurückgegriffen.
   Abgesehen von dieser Modifikation wurde das Verfahren unverändert übernommen. Diese Heuristik ist relativ rechenaufwändig für einen Genetischen Algorithmus: Der Aufwand beträgt $O(n^3)$ für die Erzeugung eines Individuums und $O(n^3)$ für jede Verbesserung zum Finden des lokalen Optimums.

Abbildung 10 zeigt die Vorgehensweise bei der Auswahl einer Kante zum Aufbau des Nachkommens: Zwei Kanten werden zufällig aus der Vereinigungsmenge der Kanten der Elternteile selektiert (dicke Linien). Nun werden diese beiden Kanten entsprechend der jeweiligen Heuristik verglichen und die beste der beiden wird dem neuen Baum hinzugefügt. Im gezeigten Beispiel wurde die Kante (3,5) ausgewählt und verwendet.

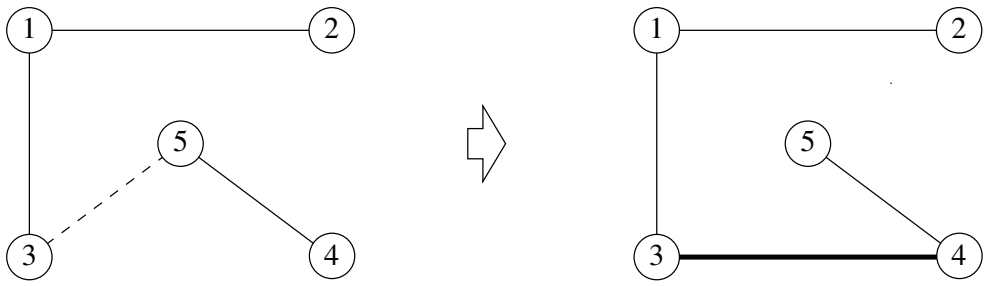


Abbildung 11: Mutation

5.3 Mutation

Es wird eine zufällige Kante aus dem Spannbaum entfernt und durch eine andere Kante aus G, welche ebenfalls zufällig ausgewählt wird, zu ersetzen. Da der Ausgangsgraph G vollständig ist, gibt es immer alternative Kanten, die die entfernte Kante ersetzen können.

1. Entfernen einer zufällig gewählten Kante e aus der Kantenmenge E des Spannbaumes T. Es befinden sich nun $n-1$ Kanten in E.

2. Aufbau der Union-Find-Structure durch Einfügen der verbliebenen Kanten aus E. Es existieren nun zwei Knotenmengen bzw. Teilbäume, die durch das Entfernen der Kante e entstanden sind.

3. Zufällige Auswahl eines Knoten u aus einem Teilbaum und eines Knoten v aus dem anderen Teilbaum. Diese beiden Knoten werden durch die Kante $e' = (u, v)$ wieder verbunden. Kante e' existiert, da der Ausgangsgraph vollständig ist.

Dieser Operator weist eine hohe Lokalität (der neue Baum wird nicht zu sehr verändert) auf und bringt eventuell selten verwendete Kanten (wegen zu großer Distanz oder zu geringem Kommunikationsbedarf benachteiligte Kanten) wieder in die Population ein. Der Mutationsoperator ist unbedingt erforderlich, da die Rekombination kein neues Genmaterial in die Population einbringt (siehe 5.2).

Abbildung 11 veranschaulicht diese Vorgehensweise: Kante (3,5) wird entfernt und die entstandenen Teilbäume werden durch eine andere Kante (3,4) wieder zusammengefügt.

Der Operator weist einen Aufwand $O(n)$ auf. Nach dem Entfernen einer Kante aus dem Baum muß die Kantenliste wieder neu aufgebaut werden (Schritt 2), damit die beiden entstandenen Teilbäume identifiziert werden können. Sodann kann eine neue Kante, welche die beiden Teilbäume wieder verbindet, gefunden und eingefügt werden. Schritte 1 und 3 haben konstanten Aufwand $O(1)$.

6 Implementierung

Der Algorithmus wurde auf einem System mit Pentium-IV CPU (1.8 GHz) unter Linux (SuSE Linux-Distribution 9.0) implementiert. Als C++ Compiler wurde gcc version 3.3.1 (SuSE Linux) verwendet.

6.1 Verwendete Bibliotheken

6.1.1 LEDA

Bei der Implementierung des Genetischen Algorithmus wurde die *Library of Efficient Data types and Algorithms* (LEDA) verwendet. Diese Bibliothek wurde vom Max Planck Institut für Informatik entwickelt und wird von Algorithmic Solutions Software GmbH vertrieben.

Die Bibliothek bietet eine Vielzahl an Algorithmen und Datenstrukturen für die Implementierung mathematischer und graphischer Lösungen. Von der großen Auswahl, die diese Bibiliothek bietet, wurden die *sets* verwendet, um die Kantenmenge des Spannbaumes abzubilden.

Sets sind mit Suchbäumen implementiert. Die Operationen `insert()`, `delete()` und `member()` haben den Aufwand $O(\log n)$, `empty()` und `size()` den Aufwand $O(1)$, und `clear()` den Aufwand $O(n)$, wobei n die Anzahl der Elemente im Set darstellt.

Für die Operatoren `join()`, `intersect()` und `diff()` gilt: Seien S1 und S2 zwei Sets des Types T mit |S1| = n1 und |S2| = n2, dann benötigt S1.join(S2) und S1.diff(S2) den Aufwand $O(n2 \log(n1 + n2))$, S1.intersect(S2) benötigt $O(n1 \log(n1 + n2))$.

6.1.2 EALib

Die EALib wurde am Institut für Algorithmen und Computergraphik an der Technischen Universität Wien entwickelt und stellt das Grundgerüst für einen Genetischen Algorithmus bereit. Das Hauptprogramm besteht aus einer Schleife mit dem Algorithmus (siehe 2). Die Klasse `chromosome` stellt die Basisklasse aller Chromosomen dar. Jedes problemspezifische Chromosom wird nun von dieser Klasse abgeleitet. Sie verfügt über die notwendigen Eigenschaften und Methoden für Rekombination, Mutation, Selektion und Bewertung, damit der Genetische Algorithmus damit arbeiten kann.

Die Bibliothek bietet auch die Möglichkeit, alle Einstellungen des Algorithmus in einer Konfigurationsdatei zu speichern und diese beim Start zu verwenden. Weiters können Implementierungen ebenfalls diesen Mechanismus nutzen und ihre problemspezifische Parameter in der Konfigurationsdatei ablegen oder als Startparameter angeben. Die Konfigurationsdatei und andere verwendete Dateiformate werden im Abschnitt 6.3 beschrieben.

6.2 Klassenübersicht

In der vorliegenden Implementierung des Genetischen Algorithmus für das OCSTP dient die von `chromosome` abgeleitete Klasse `TreeChrom` als Wrapper für einen Spannbaum – einem Objekt der Klasse `TREE`, welche weiter unten beschrieben ist.

Abbildung 12 zeigt das UML-Diagramm der Klassenhierarchie. Die Klasse `EA` stellt den Genetischen Algorithmus selbst dar. Jene Klasse besitzt eine Referenz auf ein Populationsobjekt der Klasse `EA_POP`. Diese wiederum beinhaltet die Population an Individuen, welche von der abstrakten Klasse `chromosome` abgeleitet sein müssen. Im vorliegenden Fall wird dies durch die abgeleitete Klasse `TreeChrom` erreicht, welche die Implementierung der Individuen übernimmt – und zwar derart, daß jedes Objekt der Klasse `TreeChrom` ein Objekt der Klasse `TREE` besitzt, welches den konkreten Spannbaum verwaltet.

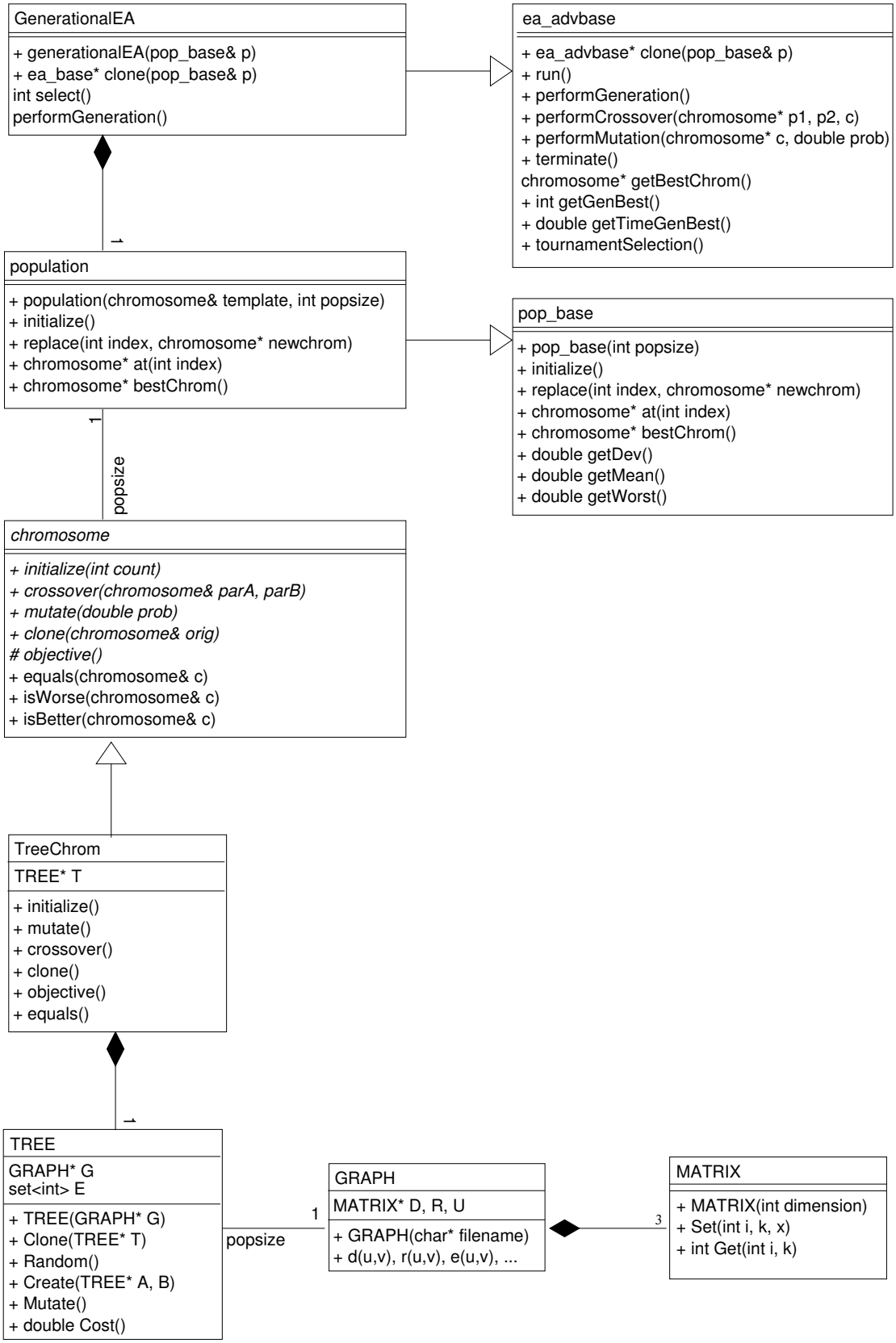


Abbildung 12: Klassendiagramm

6.2.1 Unit NODE

Diese Datei stellt einen Datentyp zur Verfügung, mit dem eine Knotenliste gebildet werden kann. Nach der Initialisierung der Liste mit

```
Initialize(NODE* N, int node_count);
```

kann damit gearbeitet werden. Die Initialisierung hat den Aufwand $O(nodecount)$. Nun können die einzelnen Knoten verbunden werden. Dies geschieht mit Hilfe der Funktion

```
bool Connect(NODE* N, int n1, int n2);
```

welche die beiden Knoten n1 und n2 in der Knotenliste N verbindet und bei Erfolg `true` zurückliefert. Sind die beiden Knoten bereits verbunden, wird `false` zurückgeliefert. Es wird eine union-find-structure mit path compression verwendet, um festzustellen, welche Knoten bereits verbunden sind. Der Aufwand dieser Funktion ist amortisiert $O(1)$, d.h. jeder Test, ob Knoten verbunden sind, benötigt konstanten Aufwand. Beim Hinzufügen des neuen Knoten wird jedoch eine zusätzliche Routine durchlaufen, um die interne Datenstruktur zu aktualisieren (path compression). Dieser zusätzliche Aufwand ist im Verhältnis aber sehr gering, sodaß er keine nennenswerte Auswirkung auf den Gesamtaufwand hat.

6.2.2 Klasse LIST

Diese Klasse ist ein Hilfswerkzeug zur Verwaltung von Kantenlisten. Eine Liste vom Typ `LIST` beinhaltet ein Array von Integerwerten, aus dem mit Aufwand $O(1)$ Kanten zufällig ausgewählt (und/oder entfernt) werden können.

6.2.3 Klasse MATRIX

Klasse MATRIX implementiert eine Dreiecksmatrix für die Requirements R bzw. Distances D des Eingabegraphen. Alle Zugriffe auf die Werte der Matrix haben den Aufwand $O(1)$.

6.2.4 Klasse GRAPH

Diese Klasse kapselt den Eingabegraphen G. Die Repräsentation des Graphen erfolgt über dessen Requirementsmatrix R und Distancematrix D. Die Angabe der Eingabedatei und das Einlesen der Daten erfolgt im Konstruktor der Klasse:

```
GRAPH* G = new GRAPH('filename');
```

Zusätzlich zu diesen Stammdaten werden die kürzesten Pfade zwischen allen Knoten berechnet und in der Matrix U gespeichert. Die Berechnung der Matrix U hat den (einmaligen) Aufwand $O(n^3)$. Weiters wird der durchschnittliche Kommunikationsbedarf und die durchschnittliche Distanz berechnet - der Aufwand beträgt jeweils $O(n)$. Alle Funktion, die nach der Erzeugung des Objekts aufgerufen werden, haben den Aufwand $O(1)$.

6.2.5 Klasse TREE

Diese Klasse enthält eine Lösung des Algorithmus, also einen Spannbaum. Sie ist somit auch ein Chromosom des Genetischen Algorithmus und verfügt über die erforderlichen Methoden zur Selektion, Recombination und Mutation.

Dem Konstruktor der Klasse wird der Eingabegraph übergeben:

```
TREE* T = new TREE(GRAPH* G);
```

Methode Random() erzeugt einen Zufallsspannbaum.

```
T->Random();
```

Methode `Create(TREE* A, TREE* B)` führt die Rekombination aus den Elternelementen A und B ohne Heuristik durch:

```
T->Create(TREE* A, TREE* B);
```

Alternativ zu dieser Methode bietet sich die folgende Methode `Create(TREE* A, TREE* B, int hmode, float hprob, int hksel)` an, bei welcher mit dem Parameter `hmode` die zu verwendende Heuristik angegeben werden kann. Die implementierten Heuristiken sind die unter Absatz 5.2.2 beschriebenen Verfahren. Der Parameter `hprob` akzeptiert Werte von 0 bis 1 und gibt die Wahrscheinlichkeit an, mit welcher die Heuristik bei der Auswahl einer Kante verwendet wird. Je kleiner der Wert, umso mehr wird der Einfluß der Heuristik abgeschwächt. Der Parameter `hksel` definiert die Anzahl an Kanten, welche zur Tournament-Selektion herangezogen und mit Hilfe der Heuristik ausgewertet werden. Je größer der Wert, umso stärker ist der Einfluß der Heuristik:

```
T->Create(TREE* A, TREE* B, int hmode,
   float hprob, int hksel);
```

Methode Mutate() führt eine Mutation des Spannbaumes durch:

```
T->Mutate();
```

Methode Cost() liefert die Kosten des Spannbaumes zurück:

```
double cost = T->Cost();
```

Die Aufrufe der Methoden erfolgen aus der Klasse `TreeChrom`, abgeleitet von der Klasse `chromosome` der EALib, welche die Steuerung des Algorithmus (Verwaltung der Population, Selektion, Erzeugung der neuen Generation und Termination) kontrolliert.

6.3 Dateiformat

Die Eingabe des Ausgangsgraphen erfolgt über eine Textdatei, welche die beiden den Graphen definierenden Matrizen R und D enthält. Schließlich wird das Ausgabeformat beschrieben, welches die beste gefundene Lösung präsentiert.

6.3.1 Eingabeformat

Das Format für Eingabegraphen ($n \times n$ Matrizen R und D) ist wie folgt definiert:

```
0 d1.2 d1.3 d1.4 .... d1.n<LF>
0    0 d2.3 d2.4 .... d2.n<LF>
0    0    0 d3.4 .... d3.n<LF>
....                   ....
0    0    0    0 ....    0<LF>
0 r1.2 r1.3 r1.4 .... r1.n<LF>
0    0 r2.3 r2.4 .... r2.n<LF>
0    0    0 r3.4 .... r3.n<LF>
...                    ....
0    0    0    0 ....    0
```

Die erste Matrix in der Datei ist die Entfernungsmatrix (distance matrix). Darauf folgt die Bedarfsmatrix (requirement matrix). Die einzelnen Ganzzahlwerte sind durch Whitespaces getrennt. Ein LineFeed-Zeichen <LF> (= 0x10) signalisiert einen Zeilenumbruch. Als Beispiel sei die Eingabedatei für die Testinstanz Berry6 gegeben:

```
0 3 6 5 9 7
0 0 3 2 4 8
0 0 0 3 7 2
0 0 0 0 9 2
0 0 0 0 0 1
0 0 0 0 0 0
0 5 13 12  8  9
0 0  7  4  2  6
0 0  0  3 10 15
0 0  0  0 11  7
0 0  0  0  0 12
0 0  0  0  0  0
```

Die Knoten werden von 1 bis n durchnumeriert. Die Numerierung der Kanten wird bei 1 – beginnend von links oben nach rechts unten – für alle Werte größer 0 wie folgt vorgenommen:

```
(1,2) (1,3) (1,4) (1,5) ... (1,n)
      (2,3) (2,4) (2,5) ... (2,n)
            (3,4) (3,5) ... (3,n)
                  ...       ...
                          (n-1,n)
```

Ein kompletter Graph mit n Knoten besteht aus $\frac{n(n-1)}{2}$ Kanten. Für jeden Graphen existiert also eine Liste mit Kanten von 1 bis $|E| = \frac{n(n-1)}{2}$:

Kante 1 = (1,2) = (2,1)
Kante 2 = (1,3) = (3,2)
...
Kante $n-1 = (1,n) = (n,1)$
Kante n = (2,3) = (3,2)
...
Kante $|E| = (n-1,n) = (n,n-1)$.

6.3.2 Ausgabeformat

Die Ausgabe des Lösungbaumes erfolgt über die Standard-Ausgabe (stdout). Das Format ist wie folgt definiert:

(node)-edge-(node)<LF>
(node)-edge-(node)<LF>
...
(node)-edge-(node)

Jede Zeile entspricht einer Kante im Spannbaum. Jede Ausgabedatei besitzt somit $n-1$ Zeilen. Jede Zeile (=Kante) besteht aus dem ersten Knoten (in runden Klammern), der Kantennummer (zwischen den Bindestrichen) und dem zweiten Knoten (in runden Klammern). Als Beispiel der Spannbaum mit 6 Knoten von Berry:

```
(1)-1-(2)
(2)-7-(4)
(3)-12-(6)
(4)-14-(6)
(5)-15-(6)
```

Die Numerierung der Kanten des Eingabegraphen ist unter Kapitel 6.3.1 beschrieben.

6.4 Aufrufparameter

Die hier beschriebenen Einstellungen werden vom Genetischen Algorithmus verwendet und bestimmen dessen Verhalten. Diese Parameter können dem Programm als Kommandozeilenparameter beim Aufruf mitgegeben oder aus einer Konfigurationsdatei gelesen werden:

`eamod` Benutztes EA-Modell: SteadyState (0) oder Generational (1). Beim SteadyState-Modell wird immer nur ein Individuum pro Generation erzeugt, während beim Generational-Modell die gesamte Population neu erzeugt wird.

`elit` Elitismus: Ein (1) oder Aus (0). Wird Elitismus verwendet, so wird das jeweils beste Individuum niemals durch ein neu erzeugtes Individuum überschrieben. Auf diese Weise kann eine einmal gefundene beste Lösung nicht wieder verloren gehen.

`dupelim` Duplikate eliminieren: Ja (1) oder Nein (0). Werden beim Crossover bereits existierende Individuen erzeugt (Duplikate), so werden diese nicht in die Population aufgenommen.

`logext` Erweiterung der Logdatei. In diese Datei (`oname+logext`) werden Informationen zur Laufzeit des Algorithmus geschrieben. Bei jeder Änderung der Population werden die Generationsnummer, der beste und der schlechteste Fitneßwert, der Mittelwert und die Standardabweichung der Fitneßwerte der Gesamtpopulation ausgegeben.

`maxi` Maximieren: Ja (1) oder Nein (0 = Minimieren). Dieser Parameter bestimmt, ob das Problem ein Maximierungsproblem oder ein Minimierungsproblem darstellt. Im Falle des OCSTP haben wir es mit einem Minimierungsproblem zu tun, daher setzen wir maxi = 0.

`oname` Name der Ausgabedatei(en). Dieser Parameter definiert den Basisnamen für Ausgabedateien. Abhängig von der Konfiguration der Bibliothek werden verschiedene Ausgabedateien erzeugt, welche diesen Dateinamen, aber unterschiedliche Erweiterungen besitzen. Siehe Parameter `outext` und `logext` für weitere Informationen.

`outext` Erweiterung der Ausgabedatei. In diese Datei (`oname+outext`) werden die Konfiguration des Algorithmus und Informationen über das Ergebnis geschrieben. Enthalten sind unter anderem die CPU-Zeit zum Finden der besten Lösung, die Anzahl der Generationen, der ausgeführten Selektionen, Crossover-Operationen und Mutationen sowie die Anzahl der eleminierten doppelten Individuen (falls diese Option gewählt wurde).

`pcross` Crossover-Wahrscheinlichkeit im Bereich von 0 bis 1.

`pmut` Mutations-Rate/Wahrscheinlichkeit für ein erzeugtes Chromosom. Ist dieser Wert negativ, so wird der Absolutwert des Parameters als durchschnittliche Mutationsrate pro Individuum interpretiert (Die tatsächliche Anzahl der Mutationen wird mit einer Poisson-Verteilung berechnet).

`popsize` Populationsgröße (Vorgabe = 100). Dieser Wert bestimmt die Anzahl von Chromosomen in der Population.

`repl` Ersetzungsstrategie: Ein neu erzeugtes Chromosom ersetzt ein beliebiges bestehendes Chromosom (0) oder das schlechteste in der Population (1). Wird ein Wert repl > 1 angegeben, so wird das zu ersetzende Individuum mit Hilfe einer Tournament-Selektion aus repl Individuen bestimmt (Vorgabe = 1).

`tcgen` Konvergenz-Abbruchbedingung: Der Algorithmus bricht ab, wenn nach tcgen Generationen keine bessere als die bis dahin gefundene Lösung berechnet werden konnte.

`tselk` Auswahlgröße für Tournament-Selektion: Bei jeder Rekombination werden die Eltern aus einer Gruppe mit tselk Individuen ausgewählt (Vorgabe = 2).

Es folgen nun die Parameter, die speziell mit dem OCST-Problem zusammenhängen und das Verhalten der Heuristiken beeinflussen:

`infile` Datei mit den Matrizen des Eingabegraphen (siehe Kapitel 6.3.1).

`hmode` Benutzte Heuristik für die Kantenauswahl (0 bis 7):
0 = Keine Heuristik: Kanten werden zufällig ausgewählt und verwendet.
1 bis 6 = Die unter Kapitel 5.2.2 beschriebenen Heuristiken werden zur Kantenauswahl verwendet.
7 = Heuristik von Ahuja und Murty (siehe Kapitel **??**).

`hprob` Heuristik-Wahrscheinlichkeit (im Bereich von 0.0 bis 1.0, Vorgabewert = 1.0).
Dieser Parameter legt fest, mit welcher Wahrscheinlichkeit die mit hmode gewählte Heuristik zur Auswahl einer Kante benutzt wird. Bei jedem Konstruktionsschritt des Spannbaumes (Hinzufügen einer Kante) wird also entschieden,

ob eine Heuristik verwendet oder eine Kante zufällig ausgewählt werden soll. Dadurch kann der Einfluß der Heuristik auf den Rekombinationsprozeß abgeschwächt werden.
Hinweis: Der Wert `hprob = 0.0` (Wahrscheinlichkeit = 0%) entspricht somit `hmode = 0` (= keine Heuristik verwenden).

`hksel` Anzahl der Kanten für Tournament-Selektion (Vorgabewert = 2).
Dieser Parameter legt fest, wieviele Kanten bei jedem Konstruktionsschritt gegeneinander antreten und mit Hilfe der gewählten Heuristik ausgewertet werden, um die beste Kante zu bestimmen, die dem Baum hinzugefügt wird. Je höher dieser Wert ist, umso größer ist der Einfluß der Heuristik.
Hinweis: Im Falle `hmode = 0` ist dieser Parameter bedeutungslos.

Alle Aufrufparameter können in einer Konfigurationsdatei abgelegt werden, welche beim Start des Programms automatisch eingelesen wird. In der Kommandozeile angegebene Parameter überschreiben dabei die in der Konfigurationsdatei gespeicherten Werte. Das Format der Konfigurationsdatei 'ea.conf' ist einfach:

```
<parameter><white_space(s)><wert><LF>
...
```

Wenn die Konfigurationsdatei den selben Namen wie das Programm besitzt (aber mit der Erweiterung '.conf'), so wird sie automatisch beim Programmstart eingelesen.

7 Experimentelle Ergebnisse

Der in dieser Arbeit vorgestellte Algorithmus für das Optimum Communication Spanning Tree Problem wurde wie beschrieben implementiert. Dieser Abschnitt beschreibt die Vorgehensweise beim Testen der Implementierung.

Nachfolgend werden die verwendeten Testinstanzen, die Einstellungen des Algorithmus und die Untersuchungsergebnisse vorgestellt. Weiters sind die Ergebnisse der Untersuchung des Selektionsdruckes und des Konvergenzverlaufs dargestellt.

7.1 Testinstanzen

Für die Durchführung der Tests wurden die in Tabelle 3 aufgelisteten Testinstanzen verwendet.

Alle Testinstanzen sind vollständige Graphen. Die Spalte 'Bestes Resultat' enthält die besten Lösungen für die jeweilige Instanz, welche in anderen Quellen [18] gefunden wurden. Für die Testinstanzen von Raidl existieren noch keine Vergleichswerte.

Die Instanzen von Berry (Berry6, Berry35 und Berry35u) stammen aus dem Jahre 1995 und wurden von Berry, Mutagh, Sugden und McMahon [32] verwendet.

Die Instanzen von Palmer (Palmer12 und Palmer24) stammen aus dem Jahre 1995. Seine Testdaten repräsentieren amerikanische Städte und die Distanzen sind reale Werte. Palmer benutzte in [1] außerdem Instanzen mit 47 und 98 Knoten, welche aber leider nicht mehr verfügbar sind.

Testinstanz	Beschreibung	Bestes bekanntes Resultat
Berry6	Graph mit 6 Knoten von Berry	534
Berry35	Graph mit 35 Knoten von Berry	16273
Berry35u	Wie Berry35, jedoch mit einheitlicher Distanz $d_{u,v} = 1, \forall u, v \in V$	16915
Palmer12	Graph mit 12 Knoten von Palmer	3428510
Palmer24	Graph mit 24 Knoten von Palmer	1091500
Raidl10	Graph mit 10 Knoten von Raidl	-
Raidl20	Graph mit 20 Knoten von Raidl	-
Raidl35	Graph mit 35 Knoten von Raidl	-
Raidl50	Graph mit 50 Knoten von Raidl	-
Raidl75	Graph mit 75 Knoten von Raidl	-
Raidl100	Graph mit 100 Knoten von Raidl	-

Tabelle 3: Testinstanzen

Die Instanzen von Günther Raidl (Raidl10, -20, -35, -50, -75 und -100) aus dem Jahre 2001 sind direkt von Günther Raidl[1] erhältlich.

Außerdem sind alle Instanzen auch von Franz Rothlauf[2] erhältlich, wobei diese auch in [20] abgedruckt sind. Der Appendix dieses Buches, welcher die Testinstanzen enthält, kann kostenlos unter www.bwl.uni-mannheim.de/wifo1/de2/lehre_geabook_de.html heruntergeladen werden.

7.2 Einstellungen und Ablauf

Folgende Einstellungen wurden für den Genetischen Algorithmus verwendet:

Parameter	Wert	Beschreibung
`eamod`	0	Steady-State Algorithmus wird verwendet
`dupelim`	1	Identische doppelte Lösungen werden eliminiert
`elit`	1	Elitismus wird verwendet
`maxi`	0	Maximize OFF (=Minimierungsproblem)
`tcgen`	50000	Termination nach 50000 Iterationen ohne Verbesserung
`tgen`	1000000	Maximal 1 Million Iterationen insgesamt durchführen
`popsize`	100	Populationsgröße = 100 Individuen
`pmut`	-1	1 Mutation pro Chromosom
`pcross`	1	Rekombinationswahrscheinlichkeit = 100%
`tselk`	2	Tournament-Selektion mit 2 Individuen

Für jede Testinstanz wurden mit jeder Heuristik (`hmode` = 0 bis 7) jeweils 40 Durchläufe durchgeführt. Hierbei wurde immer mit folgender Einstellung gearbeitet:

[1] Günther Raidl, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria, E-Mail: raidl@ads.tuwien.ac.at

[2] Franz Rothlauf, Department of Information Systems, University of Mannheim, Germany, E-Mail: rothlauf@bwl.uni-mannheim.de

Parameter	Wert	Beschreibung
`hprob`	1,0	Die gewählte Heuristik wird immer verwendet = 100%
`hksel`	2	Tournament-Selektion mit 2 Kanten bei jeder Auswahl

Die jeweils gewählte Heuristik wurde also bei jeder Auswahl einer Kante, die dem Baum hinzugefügt wurde, verwendet. Weiters wurden immer 2 Kanten aus der Menge der Elternkanten gewählt, miteinander verglichen und der Gewinner zur Konstruktion verwendet.

Der Parameter `hmode` definiert die verwendete Heuristik. An dieser Stelle sei nochmals eine Aufstellung der Heuristiken gegeben:

Heuristik (hmode)	Beschreibung
None (0)	Keine Heuristik (Kanten werden zufällig ausgewählt)
Dist (1)	Wähle (s,t) wenn gilt: $d_{s,t} < d_{u,v}$
Req (2)	Wähle (s,t) wenn gilt: $r_{s,t} > r_{u,v}$
Dist-Rel (3)	Wähle (s,t) wenn gilt: $\frac{d_{s,t}}{r_{s,t}+R} < \frac{d_{u,v}}{r_{u,v}+R}$
Req-Rel (4)	Wähle (s,t) wenn gilt: $\frac{r_{s,t}}{d_{s,t}+D} > \frac{r_{u,v}}{d_{u,v}+D}$
Req&Dist (5)	Wähle (s,t) wenn gilt: $\frac{r_{s,t}+R}{d_{s,t}+D} > \frac{r_{u,v}+R}{d_{u,v}+D}$
Random (6)	Zufällige Heuristik 1-5 bei jeder Kanteauswahl
G-Ahuja (7)	Heuristik nach Ahuja und Murty

Die Heuristiken sind unter 5.2.2 beschrieben.

Weiters werden für Vergleichszwecke die Ergebnisse anderer Algorithmen – falls für diese Testinstanz vorhanden – angegeben:

Algorithmus	Beschreibung
Li	Genetischer Algorithmus von Li und Bouchebaba [18]
Palmer	Genetischer Algorithmus von Palmer [1]
Ahuja	Heuristischer Algorithmus von Ahuja und Murty [3]
Berry	Algorithmus von Berry (Ergebnisse aus [18])

7.3 Ergebnisse

Nachfolgend sind alle Ergebnisse in Tabellenform dargestellt. Die Spalte "Quote" gibt an, wieoft das Minimum gefunden wurde (in Prozent). Spalte "Iteration" enthält die durchschnittliche Iteration, in welcher das beste Ergebnis gefunden wurde. Der Zeitaufwand (Spalte "Zeit") ist in Sekunden (CPU-Zeit) für die beste Lösung angegeben.

Die Angabe der Iteration beim Algorithmus von Li ergibt sich aus der verwendeten Populationsgröße = 200 mal der in [18] angegebenen Generation und stellt somit die Anzahl der erzeugten und evaluierten Lösungen dar.

Die Angabe der gefundenen Ergebnisse mit den Algorithmen von Berry für die Testinstanzen Berry6, Berry35 und Berry35u wurde aus [18] entnommen. Das Ergebnisse von Palmer12 und Palmer24 stammen aus [1]. Weiters sei bemerkt, daß die Angaben der Ergebnisse für Palmer12 und Palmer24 aus der besagten Quelle doppelt so hoch sind wie oben angegeben. Dies ist auf eine Skalierung der Werte zurückzuführen. Daher wurden die Werte halbiert, um eine bessere Vergleichsmöglichkeit zu gewährleisten. Außerdem sind die Werte in den Quellen nur auf die Tausenderstelle genau angegeben; Diese Werte sind in obigen Tabellen mit "≈"

Heuristik	Mittelwert	Std.Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	534	0,00	0,0	534	100,0	159	0,02
Dist	534	0,00	0,0	534	100,0	71	0,02
Req	534	0,00	0,0	534	100,0	243	0,03
Dist-Rel	534	0,00	0,0	534	100,0	177	0,02
Req-Rel	534	0,00	0,0	534	100,0	140	0,02
Req&Dist	534	0,00	0,0	534	100,0	114	0,02
Random	534	0,00	0,0	534	100,0	136	0,02
G-Ahuja	534	0,00	0,0	534	100,0	1	$< 0{,}01$
Ahuja			0,0	534			$< 0{,}01$
Li			0,0	534		100	
Berry			0,0	534			

Tabelle 4: Berry6

Heuristik	Mittelwert	Std.Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	17557	681,89	3,8	16915	32,5	17300	16,32
Dist	17251	446,72	2,0	16915	45,0	7100	6,76
Req	17374	415,90	2,7	16915	25,0	17194	0,03
Dist-Rel	17404	452,58	2,9	16915	25,0	33423	0,02
Req-Rel	17308	401,05	2,3	16915	25,0	18241	17,10
Req&Dist	17297	456,44	2,3	16915	37,5	16366	15,28
Random	17267	347,78	0,1	16915	27,5	11952	11,47
G-Ahuja	16915	0,00	0,0	16915	100,0	287	3,37
Ahuja			0,0	16915			$< 0{,}01$
Li			0,0	16915		620	
Berry			80,1	30467			

Tabelle 5: Berry35

Heuristik	Mittelwert	Std. Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	16361	307,30	1,2	16167	15,0	45329	43,55
Dist	16192	12,32	0,2	16167	10,0	49479	48,50
Req	16191	10,46	0,1	16167	10,0	50375	48,96
Dist-Rel	16189	11,87	0,1	16167	17,5	47138	45,99
Req-Rel	16188	14,80	0,1	16167	25,0	51369	49,93
Req&Dist	16191	13,41	0,1	16167	15,0	48642	47,15
Random	16188	13.02	0,1	16167	20,0	47538	46,83
G-Ahuja	16627	419,48	2,8	16204	5,0	60782	561,30
Ahuja			16,0	18756			$< 0{,}01$
Li			1,6	16420		780	
Berry			4,6	16915			

Tabelle 6: Berry35u

Heuristik	Mittelwert	Std. Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	3428510	0,00	0,0	3428510	100,0	6335	0,91
Dist	3428510	0,00	0,0	3428510	100,0	5308	0,83
Req	3428510	0,00	0,0	3428510	100,0	5186	0,80
Dist-Rel	3428510	0,00	0,0	3428510	100,0	5141	0,79
Req-Rel	3429246	4654,87	0,0	3428510	97,5	12077	1,80
Req&Dist	3428510	0,00	0,0	3428510	100,0	12489	1,84
Random	3428510	0,00	0,0	3428510	100,0	7244	1,12
G-Ahuja	3428510	0,00	0,0	3428510	100,0	752	0,45
Ahuja			0,1	3431693			$< 0{,}01$
Li			0,0	3428510		260	
Palmer			0,0	3428510			$\approx$180,00

Tabelle 7: Palmer12

Heuristik	Mittelwert	Std.Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	1086660	0,00	0,0	1086660	100,0	17338	8,03
Dist	1086660	0,00	0,0	1086660	100,0	15794	7,53
Req	1086660	0,00	0,0	1086660	100,0	13406	6,48
Dist-Rel	1086660	0,00	0,0	1086660	100,0	17136	8,17
Req-Rel	1086723	225,71	0,0	1086660	92,5	48882	23,10
Req&Dist	1086747	266,24	0,0	1086660	90,0	47491	21,88
Random	1086660	0,00	0,0	1086660	100,0	21418	10,39
G-Ahuja	1086660	0,00	0,0	1086660	100,0	265	1,12
Ahuja			0,1	1087612			$< 0{,}01$
Li			0,0	$\approx$1087000		280	
Palmer			65,9	$\approx$1803000			$\approx$660,00

Tabelle 8: Palmer24

Heuristik	Mittelwert	Std. Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	53674	0,00	0,0	53674	100,0	1061	0,13
Dist	53674	0,00	0,0	53674	100,0	736	0,10
Req	53674	0,00	0,0	53674	100,0	2383	0,27
Dist-Rel	53674	0,00	0,0	53674	100,0	1500	0,18
Req-Rel	53674	0,00	0,0	53674	100,0	1462	0,18
Req&Dist	53674	0,00	0,0	53674	100,0	1324	0,16
Random	53674	0,00	0,0	53674	100,0	988	0,13
G-Ahuja	53674	0,00	0,0	53674	100,0	12	0,02
Ahuja			0,0	53674			$< 0{,}01$

Tabelle 9: Raidl10

Heuristik	Mittelwert	Std. Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	157570	0,00	0,0	157570	100,0	9833	3,38
Dist	157570	0,00	0,0	157570	100,0	23149	8,19
Req	164279	994,79	4,3	159884	2,5	77044	26,98
Dist-Rel	157646	479,09	0,0	157570	97,5	58140	20,23
Req-Rel	157649	500,27	0,1	157570	97,5	56965	19,48
Req&Dist	157628	353,13	0,0	157570	95,0	60164	20,55
Random	157570	0,00	0,0	157570	100,0	59664	20,92
G-Ahuja	157570	0,00	0,0	157570	100,0	369	0,90
Ahuja			1,0	159118			$< 0{,}01$

Tabelle 10: Raidl20

Heuristik	Mittelwert	Std. Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	412167	0,00	0,0	412167	100,0	26012	24,86
Dist	412228	383,58	0,0	412167	97,5	64044	61,69
Req	414340	3811.52	0,5	412167	57,5	423284	403,91
Dist-Rel	412291	341,30	0,0	412167	87,5	89641	86,00
Req-Rel	412464	1374,58	0,1	412167	90,0	88080	84,65
Req&Dist	412352	498,19	0,0	412167	85,0	87070	83.59
Random	412402	762,18	0,1	412167	90,0	95300	92,88
G-Ahuja	412167	0,00	0,0	412167	100,0	312	3,60
Ahuja			1,4	417949			$< 0{,}01$

Tabelle 11: Raidl35

Heuristik	Mittelwert	Std. Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	812637	19402,46	0,7	806864	75,0	72215	142,65
Dist	808840	3131,67	0,2	806864	43,3	317334	616,92
Req	892713	45154,05	10,6	806864	3,3	775043	1506,15
Dist-Rel	810300	8511,23	0,4	806864	40,0	345517	669,62
Req-Rel	808349	2805,82	0,2	806864	50,0	356136	687,34
Req&Dist	810893	9762,51	0,5	806864	33,3	346882	666,40
Random	817780	20513,66	1,4	806864	16,7	418605	191,18
G-Ahuja	806864	0,00	0,0	806864	100,0	356	11,23
Ahuja			0,0	806864			

Tabelle 12: Raidl50

Heuristik	Mittelwert	Std.Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	1783502	101318,13	3,8	1717490	50,0	202020	906,90
Dist	1808114	80551,26	5,3	1717490	5,0	583961	2625,30
Req	2071928	131823,72	20,6	1895760	5,0	961182	4324,20
Dist-Rel	1861302	48848,35	8,4	1801140	5,0	982432	4423,60
Req-Rel	1900308	71913,60	10,6	1815310	5,0	955522	4283,83
Req&Dist	1888426	81652,60	10,0	1768900	5,0	958300	4268,15
Random	1835113	78222,28	6,8	1736320	5,0	971147	4396,20
G-Ahuja	1717490	0,00	0,0	1717490	100,0	474	45,89
Ahuja			1,0	1734143			$< 0{,}01$

Tabelle 13: Raidl75

Heuristik	Mittelwert	Std.Abw.	%-gap	Minimum	Quote	Iteration	Zeit
None	2686549	118743,11	4,9	2561540	15,0	291959	2377,97
Dist	2689185	95528,83	5,0	2564550	5,0	779308	6396,63
Req	3966779	578089,76	54,9	3033430	5,0	993427	8072,12
Dist-Rel	2872671	117677,07	12,1	2742700	5,0	960496	7836,96
Req-Rel	2814336	130995,99	9,9	2630320	5,0	916729	7452,12
Req&Dist	2787375	91022,16	8,8	2649320	5,0	920103	7471,38
Random	2776892	77895,78	8,4	2609240	5,0	969787	7924,16
G-Ahuja	2600596	74826,41	1,5	2561540	5,0	26389	5384,14
Ahuja			7,5	2752423			$< 0{,}01$

Tabelle 14: Raidl100

gekennzeichnet.

Zur Berechnung des %-gaps werden die Mittelwerte herangezogen. Der %-gap wird wie folgt berechnet:

$$\%\text{-gap} = \frac{Mittelwert - Optimum}{Optimum} \cdot 100.$$

In der Arbeit von *Li* sind keine Mittelwerte angegeben und beim Algorithmus *Palmer* entspricht der Mittelwert dem Minimum, d.h. der Mittelwert ist auf die Tausenderstelle genau angegeben und entspricht dem ebenfalls auf die Tausenderstelle angegebenen Minimum. *Ahuja* liefert immer das gleiche Ergebnis und daher ist hier der Mittelwert mit dem Minimum identisch.

Bei den Instanzen Raidl75 und Raidl100 wird der Algorithmus nach der maximalen Iterationsanzahl (Parameter `tgen`) abgebrochen, obwohl der Algorithmus noch nicht konvergiert hat. Da jedoch bei diesen Beispielen das Verfahren *None* (keine Heuristik) bereits erfolgreich konvergiert, besteht keine Veranlassung, bei den anderen Verfahren auf eine Termination zu warten. Eine Heuristik soll die Konvergenz des Algorithmus beschleunigen sowie der Qualitätsverbesserung der Lösungen dienen und das ist hier nicht der Fall. Daher ist eine Fortführung des Durchlaufs bis zur Konvergenz nicht erforderlich.

7.4 Auswertung und Vergleich

Nach der Durchführung der Testläufe und der Darstellung der Ergebnisse in obigen Tabellen konnten folgende wiederkehrende Muster beobachtet werden:

- Die Heuristiken *Dist* und *Dist-Rel* konvergieren besser als die Heuristiken *Req* und *Req-Rel*. Die von der Distanz abhängigen Heuristiken weisen eine geringere Anzahl an Iterationen und eine kleinere Standardabweichung der Ergebnisse auf als die vom Kommunikationsbedarf gesteuerten Heuristiken.

- Die Heuristik *Dist-Rel*, welche den Einfluß der Distanz mit Hilfe von Mittelwerten relativiert, konvergiert deutlich langsamer als ihr Verwandter *Dist*. Dies war zu erwarten, da die Relativierung den Einfluß der Heuristik abschwächt.

- Im Falle der Heuristik *Req-Rel* bewirkte die Relativierung jedoch eine Verbesserung des Ergebnisses. Durch die Relativierung wurde der Einfluß der irreführenden Heuristik *Req* abgeschwächt und der Algorithmus konnte daher früher konvergieren. Dies untermauert die Annahme, daß eine Heuristik, welche sich ausschließlich auf den Kommunikationsdedarf konzentriert, nicht für das OCST-Problem geeignet ist.

- *Ahuja* stellt eine starke Heuristik dar und liefert bereits bei nur einem Durchlauf eine gute Ausgangslösung. Durch die Anwendung dieser Heuristik im Genetischen Algorithmus – *G-Ahuja* – konnte die Wirkung noch verbessert werden. Dieses Verfahren ist zwar äußerst rechenaufwändig, findet jedoch nach nur wenigen Iterationen sehr gute Lösungen.
  Der Algorithmus hat immer die beste bekannte Lösung gefunden und dabei die höchste Trefferquote erzielt. Eine Ausnahme bildet die Testinstanz Berry35u, bei welcher das Minimum mit einer geringeren Trefferquote gefunden werden konnte. Die schlechtere Quote bei Berry35u kann auf die Besonderheit dieser Testinstanz zurückgeführt werden: Da hier alle Distanzen auf 1 gesetzt sind, wird der Einfluß der Distanz bei der

	None	Dist	Req	Dist-Rel	Req-Rel	Req&Dist	Random	G-Ahuja
Berry6	100	100	100	100	100	100	100	100
Berry35	33	45	25	25	25	38	28	100
Berry35u	15	10	10	18	25	15	20	5
Palmer12	100	100	100	100	98	100	100	100
Palmer24	100	100	100	100	93	90	100	100
Raidl10	100	100	100	100	100	100	100	100
Raidl20	100	100	3	98	98	95	100	100
Raidl35	100	98	58	88	90	85	90	100
Raidl50	75	43	3	40	50	33	17	100
Raidl75	50	5	5	5	5	5	5	100
Raidl100	15	5	5	5	5	5	5	5

Tabelle 15: Trefferquoten für jede Instanz in Prozent

Berechnung abgeschwächt bzw. obsolet. Dies bewirkt wiederum, daß ausschließlich der Kommunikationsdedarf zur Bewertung herangezogen wird und dies ist, wie bereits in obigem Punkt festgestellt, keine gute Ausgangsbedingung für eine Heuristik.
Dennoch konnte hier eine bessere Lösung als der Algorithmus von Berry und Li erzielt werden.

- Beim Vergleich der benötigten Zeit zum Finden des Minimums stellt sich heraus, daß *G-Ahuja* trotz eines Aufwandes sehr gute Leistungen zeigt. Die Lösung wird bereits nach wenigen Iterationen und dementsprechend kurzer Zeit gefunden. Dies ist auch bei großen Probleminstanzen (Raidl75 und Raidl100) zu beobachten. Obwohl *G-Ahuja* bei Raidl100 nur eine geringe Trefferquote vorweisen kann, sei auf den niedrigen Mittelwert und die relativ geringe Standardabweichung (Tabelle 14) verwiesen, die auf ein stabiles Verhalten dieser Heuristik hinweisen.

- Heuristik *Random* liefert durchschnittliche Werte und stellt somit einen soliden Mittelwert dar. Dieses Ergebnis ist auch in der Graphik der Trefferquoten zu erkennen (Abbildung 13).

- Es darf nicht übersehen werden, daß auch das zufallsgesteuerte Verfahren *None* (keine Heuristik) ebenfalls gute Ergebnisse liefert. Die Heuristik *Dist* konvergiert zwar schneller als *Req* , aber im Vergleich zur zufälligen Kantenauswahl *None* ist durchaus Verbesserungspotenial zu vermuten.

Die Tabelle 15 faßt die Trefferquoten aus den obigen Tabellen zusammen und bietet eine übersichtliche Darstellung. Die Prozentwerte wurden aus Gründen der besseren Lesbarkeit gerundet. Bei Raidl75 und Raidl100 wurden keine Werte eingetragen, da der Algorithmus bei diesen Instanzen nach Ablauf der maximalen Anzahl an Iterationen nicht konvergierte und das bis dahin erreichte Resultat nicht dem von *G-Ahuja* gelieferten entsprach (siehe entsprechende Tabellen 13 und 13).

Abgesehen von Testinstanz Berry35u hat die Heuristik eine Trefferquote von annähernd 100 Prozent für alle Durchläufe und Instanzen. Aber auch den anderen Heuristiken bereitet diese Instanz offensichtlich Schwierigkeiten. Man kann vermuten, daß bei dieser speziellen Problemstellung – alle Distanzen haben den Wert 1 – das Zusammenfassen von Knoten mit

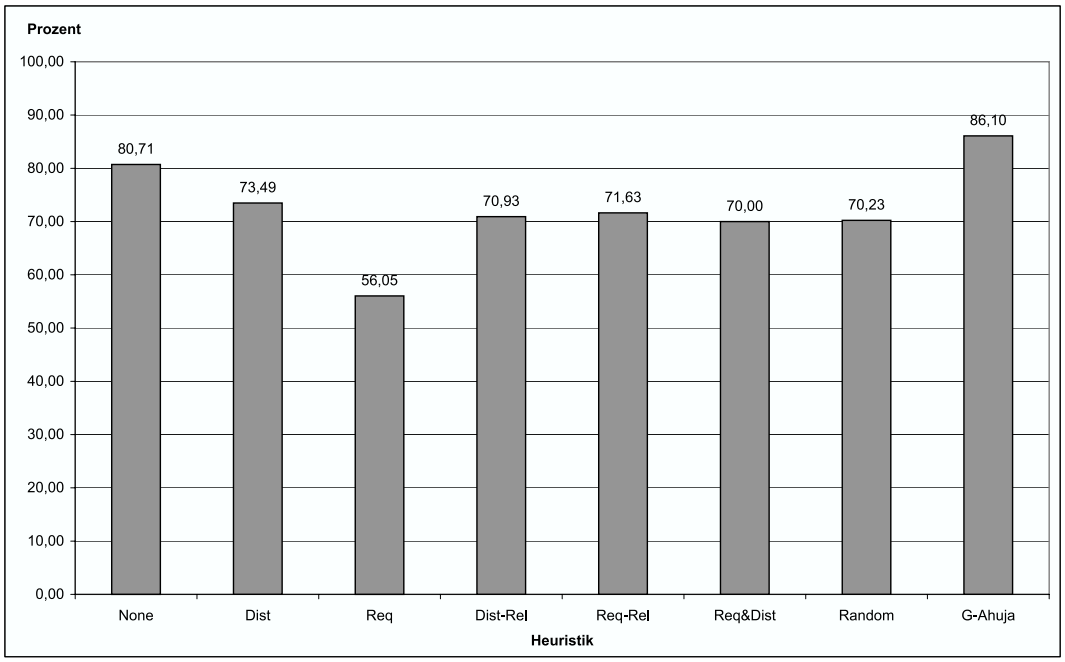


Abbildung 13: Trefferquote

hoher Kommunikation (lokale Zentren mit kurzen Distanzen und hohem Kommunikationsbedarf) nicht möglich ist und der Algorithmus daher zur Zufallsuche degeneriert.

Abbildung 13 zeigt eine zusammenfassende Gegenüberstellung der einzelnen Heuristiken. Hier wird die Trefferquote über alle Testinstanzen dargestellt. Heuristik *G-Ahuja* hat in 86,10% aller Durchläufe das jeweilige Minimum erreicht. An zweiter Stelle ist das Verfahren ohne Heuristik. *Req* bildet das Schlußlicht mit den wenigsten Treffern.

Der Umstand, daß der Algorithmus für Raidl75 und Raidl100 nicht konvergiert sondern nach Erreichen der Maximalanzahl an Iterationen abgebrochen wurde, läßt über die Trefferquote dieser Instanzen keine gültigen Aussagen zu. Eine Ausnahme bildet wieder die Heuristik *G-Ahuja*, die hier ebenfalls gute Lösungen präsentiert und dieses Ergebnis bei wiederholten Durchläufen erreicht. Dies zeigt, daß diese Heuristik sehr verläßlich arbeitet und auch für große Problemstellungen gute Lösungen in einer kurzen Zeit liefert.

Zusammenfassend kann festgestellt werden, daß im Vergleich zu anderen in der Literatur auffindbaren Verfahren der in dieser Arbeit implementierte Algorithmus das dort angegebene oder meist ein besseres Resultat liefert. Dies gilt insbesondere für die Problemstellung Berry35u, welche sich als die schwierigste Instanz herausgestellt hat: Auch hier konnte eine bessere als die bisher dokumentierten Lösungen gefunden werden.

Im folgenden Kapitel werden noch einige Untersuchungen über den Einfluß der Heuristiken auf den Konvergenzverlauf des Algorithmus durchgeführt und diskutiert.

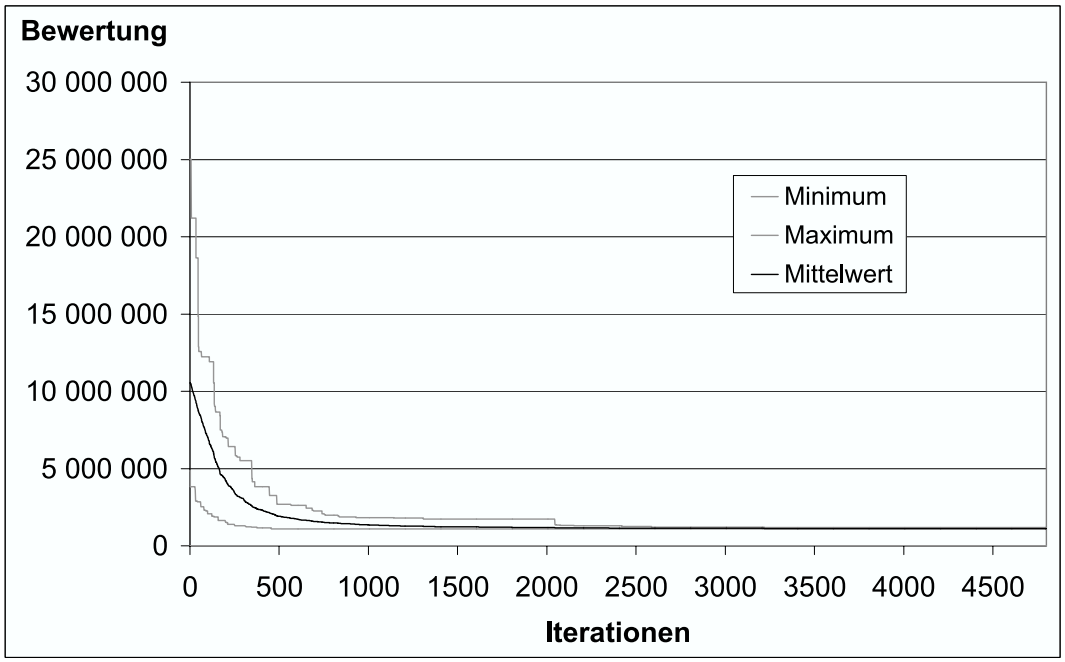


Abbildung 14: Durchschnittlicher Konvergenzverlauf mit `hksel` = 2, `hprob` = 1.0

7.5 Selektionsdruck und Konvergenzverlauf

Abbildung 14 stellt den Konvergenzverlauf des Algorithmus für die Testinstanz Palmer24 dar. Es wurden 40 Durchläufe für die Heuristik *Dist* (`hmode` = 1) aufgezeichnet und für die Darstellung gemittelt. Auf der X-Achse sind die Iterationen eingetragen; Die Y-Achse enthält die Fitneßwerte. Die grauen Linien zeigen das jeweilige Maximum und das Minimum an, während die schwarze Linie den Mittelwert darstellt.

Anschließend wurde der Parameter `hksel` = 4 gesetzt. Dies bewirkt einen höheren Selektionsdruck, da nun 4 Kanten pro Konstruktionsschritt betrachtet werden. Das Ergebnis ist in Abbildung 15 dargestellt. Man erkennt, daß die Mittelwertkurve steiler nach unten verläuft, die Population also schneller zum Minimum konvergiert. Dies ist auch an der Kurve der Minimalwerte ersichtlich: Ungefähr ab Generation 250 verläuft die Linie annähernd waagerecht, während dies in obiger Graphik erst ab Generation 500 der Fall ist.

Anschließend wurde versucht, der Einfluß der Heuristik abzuschwächen. Dazu wurde Parameter `hprob = 0.5` gesetzt. Dies bewirkt, daß die Heuristik nur mit 50%er Wahrscheinlichkeit angewendet wird (Abbildung 16). Parameter `hksel` wurde wieder auf 2 zurückgesetzt.

Hier ist das Verhalten des Algorithmus bei niedrigem Selektionsdruck zu beobachten. Erst ab etwa Iteration 3000 verläuft die Mittelwertkurve horizontal. Besonders gut ist an der Maximalwertkurve zu erkennen, wie schlechte Lösungen in der Population verbleiben und dann plötzlich verschwinden, also durch eine bessere Lösung ersetzt werden (siehe Iteration 1000 bis 1500).

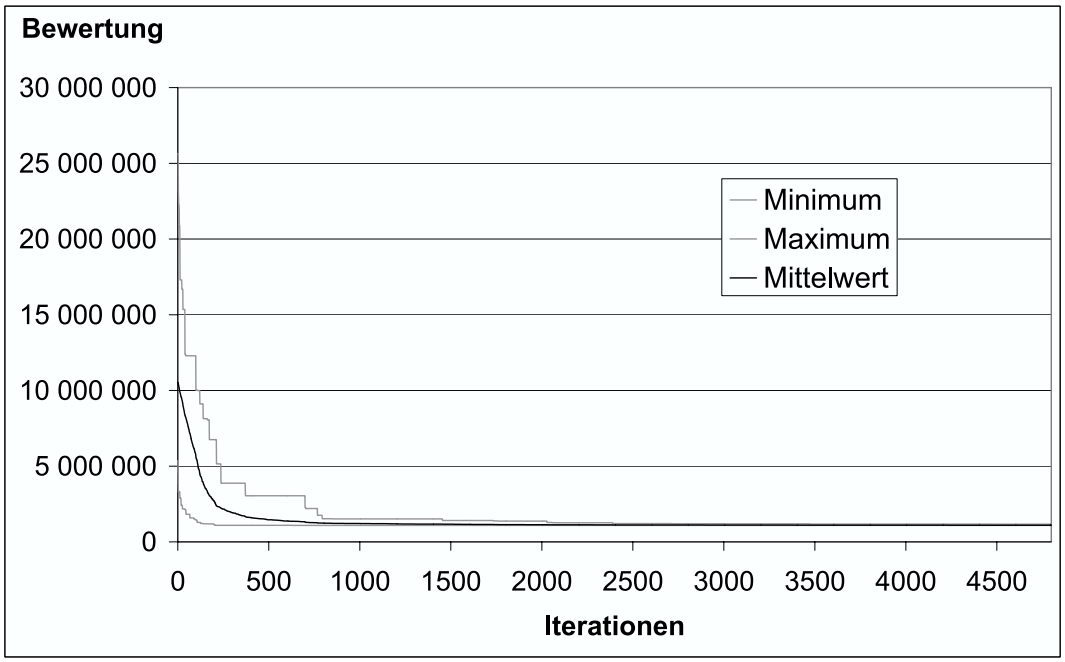


Abbildung 15: Durchschnittlicher Konvergenzverlauf mit `hksel` = 4

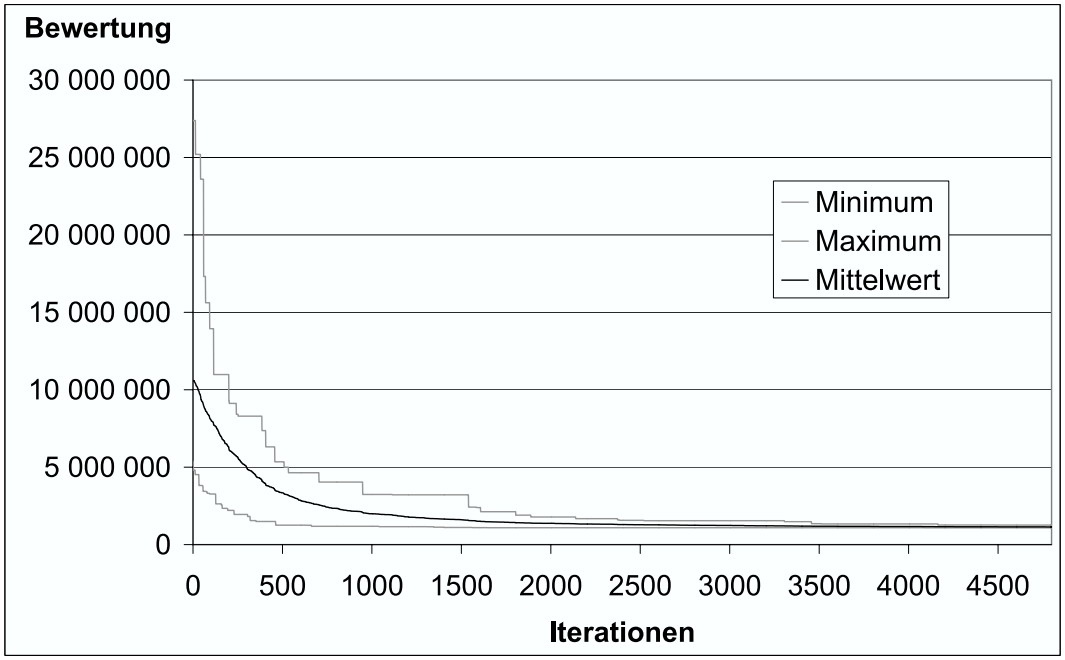


Abbildung 16: Durchschnittlicher Konvergenzverlauf mit `hprob` = 0.5

8 Zusammenfassung

In dieser Diplomarbeit wurde ein Genetischer Algorithmus für das Optimum Communication Spanning Tree Problem (OCSTP) entwickelt. Dabei wurden auch bisherige Arbeiten untersucht. Verschiedene Baumkodierungen wurden vorgestellt und für die Implementierung eines neuen Algorithmus in Betracht gezogen.

Der Genetische Algorithmus kodiert die Spannbäume als Edge-Sets und die Kanten stellen die Gene des Chromosoms dar. Dadurch wird das Arbeiten mit den Kanten und der Umgang mit dem Baum erleichtert. Eine direkte Implementierung mathematischer Algorithmen ist möglich, da nicht erst eine Konvertierung vom kodierten Chromosom in eine Kantenmenge erforderlich ist. Bei der Rekombination können Kanten aus beiden Elternteilen problemlos mit Mengenoperatoren berücksichtigt und an die Nachkommen vererbt werden.

Es wurden verschiedene Heuristiken untersucht, um eine Leistungssteigerung des Algorithmus zu erreichen. Diese Heuristiken werden bei der Kantenauswahl zur Konstruktion eines Spannbaumes verwendet. Eine starke Heuristik (nach Ahuja und Murty) wurde ebenfalls implementiert und im Genetischen Algorithmus sowie auch als eigenständiges Verfahren verwendet.

Es zeigt sich, daß eine Heuristik, welche Kanten mit geringer Distanz bevorzugt, weitaus effizienter arbeitet als eine Heuristik, die Kanten mit hoher Kommunikation vorzieht. Konzentriert man sich zu sehr auf den Kommunikationsbedarf, so wird der Algorithmus in die Irre geführt und konvergiert langsamer. Die ausführlich behandelte Heuristik nach Ahuja und Murty zeigt auch bei großen Probleminstanzen ein stabiles Verhalten und liefert gute Lösungen. Wird diese Heuristik in einem Genetischen Algorithmus verwendet, liefert dieser ausgezeichnete Ergebnisse. Der Rechenaufwand steigt zwar mit der Problemgröße enorm an, allerdings sind weniger Iterationen als mit einer einfachen Heuristik notwendig.

Der implementierte Genetische Algorithmus zeigt generell ein stabiles und reproduzierbares Verhalten und findet für einige Instanzen deutlich bessere Lösungen als die in der Literatur gefundenen Resultate [1, 18]. Speziell unter Verwendung der Heuristik nach Ahuja und Murty findet der Algorithmus mit hoher Trefferquote und in kurzer Zeit ausgezeichnete Lösungen für alle Probleminstanzen. Für zwei Instanzen wurden bessere Ergebnisse gefunden als die in der Literatur angegebenen Resultate. Bei den Testläufen stellte sich Problemstellung Berry35u (uniform distance – alle Kanten des Ausgangsgraphen haben dieselbe Länge) als die schwierigste Aufgabe heraus, aber auch hier kann der Algorithmus eine bessere Lösung als Berry und Li anbieten.

Zusammenfassend stellen Genetische Algorithmen eine gute Strategie für Spannbaumprobleme dar, da sie in der Lage sind, komplexe Suchräume auf der Suche nach dem globalen Optimum effizient zu durchforsten. Die Verwendung von Heuristiken beschleunigt den Algorithmus beim Finden einer guten Lösung. Kenntnisse über den Suchraum und über die Eigenschaften guter Lösungen stellen wichtige Hilfsmittel beim Design von Heuristiken dar. Ist dieses Wissen nicht vorhanden, so sollte man im ersten Schritt auf eine Heuristik verzichten, bis man gute Vergleichslösungen gefunden hat. Eine schlechte Heuristik schadet dem Algorithmus und wirkt sich in langen Laufzeiten aus.

Die Kodierung von Bäumen als Edge-Sets stellt eine effiziente und elegante Variante dar, durch die Algorithmen (in mathematischer Notation) ohne vorherige Umwandlung direkt angewandt und implementiert werden können. Zukünftige Arbeiten könnten sich auf das Finden geeigneter Heuristiken konzentrieren, um Genetische Algorithmen effizienter zu gestalten.

Literatur

[1] C. C. Palmer, An Approach to a Problem in Network Design using Genetic Algorithms, 1995

[2] C. C. Palmer, A. Kershenbaum, Representing Trees in Genetic Algorithms, 1994

[3] R. K. Ahuja, V. V. S. Murty, Exact and Heuristic Algorithms for the Optimum Communication Spanning Tree Problem, Transportation Science, Vol. 21, No. 3, August 1987

[4] R. K. Ahuja, V. V. S. Murty, New Lower Planes for the Network Design Problem

[5] P. M. Camerini, L. Fratta, F. Maffioli, Some Results on the Design of Tree-Structured Communication Networks, Proceedings of the Ninth International Teletraffic Congress, Spain 1979

[6] R. Dionne, M. Florian, Exact and Aproximate Algorithms for Optimal Network Design, Networks 9, pp. 37-59, 1979

[7] H. H. Hoang, A Computational Approach to the Selection of an Optimal Network, Management Science, Vol. 19, No. 5, pp. 488-498, 1973

[8] J. B. Kruskal, On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem, Proceedings of the ACM, Vol. 7, pp. 48-50, 1956

[9] R. C. Prim, Shortest Connection Networks and Some Generalizations, Bell Systems Technology Journal, Vol 36, pp. 1389-1401, 1957

[10] S. C. Narula, C. A. Ho, Degree-Constrained Minimum Spanning Tree, Computer Operations Research, Vol. 7, pp. 239-249, 1980

[11] M. Savelsbergh, T. Volgenant, Edge Excanges in the Degree-Constrained Spanning Tree Problem, Computer Operations Research, Vol. 12, pp. 241-348, 1985

[12] Gengul Zhou, Mitsuo Gen, Approach to Degree-Constrained Minimum Spanning Tree Problem Using Genetic Algorithm, Engineering Design & Automation 3, pp. 157-165, 1997

[13] G. R. Raidl, B. A. Julstrom, A Weighted Coding in a Genetic Algorithm for the Degree-Constrained Minimum Spanning Tree Problem, 2000

[14] H. Takahashi A. Matsuyama, An Approximate Solution for the Steiner Problem in Graphs, Mathematica Japonica 24, pp. 573-577, 1980

[15] V. J. Rayward-Smith, The Computation of Nearly Minimal Steiner Trees in Graphs, Int. Jour. Math. Educ. Sci. Technol. 14, pp. 15-23', 1983

[16] H. Esbensen, Computing Near-Optimal Solutions to the Steiner Problem in a Graph using a Genetic Algorithm, Networks 26, pp. 173-185, 1995

[17] Bang Ye Wu, Kun-Mao Chao, Chuan Yi Tang, Approximation Algorithms for some Optimum Communication Spanning Tree Problems, Discrete Applied Mathematics 102, pp. 245-266, 2000

[18] Y. Li, Y. Bouchebaba, A New Genetic Algorithm for the Optimal Communication Spanning Tree Problem, Proceedings of Artificial Evolution: Fourth European Conference, Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, Edmund Ronald, and Marc Schoenauer, Eds. 1999, Vol. 1829 of LNCS, pp. 162-173, Springer.

[19] J. C. Bean, Genetic Algorithms and Random Keys for Sequencing and Optimization, ORSA Journal on Computing, Vol. 6, No. 2, pp. 154-160, 1994

[20] F. Rothlauf, Representations for Genetic and Evolutionary Algorithms, Studies in Fuzziness and Soft computing. Physica, Heidelberg, 2002

[21] H. Prüfer, Neuer Beweis eines Satzes über Permutationen, Archiv für Mathematik und Physik, Vol. 27, pp. 141-144, 1918

[22] G. Raidl, F. Rothlauf, B. A. Julstrom, J. Gottlieb, Prüfer Numbers: A Poor Representation of Spanning Trees for Evolutionary Search

[23] Descrete Variable Extremum Problems, C. B. Dantzig, Operations Research, Vol. 5, No. 2, pp. 266-277, 1957

[24] G. R. Raidl, B. A. Julstrom, Edge-Sets: An Effective Evolutionary Coding of Spanning Trees, IEEE Transactions on Evolutionary Computation, 7(3), pp. 225-239, IEEE Press, 2003

[25] F. Rothlauf, J. Gerstacker, A. Heinzl, On the Optimal Communication Spanning Tree Problem, Working Papers in Information Systems, Nr. 10/2003, 2003

[26] J. H. Holland, Adaption in Natural and Artificial Systems, Univ. of Michigan Press, Ann Arbor, MI, 1975

[27] T. Bäck, Evolutionary Algorithms in Theory and Practice, IOP Press, PA, 1996

[28] T. Bäck, D. Fogel, Z. Michalewicz, Handbook of Evolutionary Computation, Oxford University Press, 1997

[29] L. Davis, Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991

[30] D. B. Fogel, Evolutionary Computation, IEEE Press, New York, 1995

[31] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, MA, 1989

[32] L. Berry, B. Mutagh, S. Sugden, G. McMahon, Application of genetic-based algorithm for optimal design of tree-structured communication networks. In Proceedings of the Regional Teletraffic Engineering Conference of the Int. Teletraffic Congress, Sourth Africa, pp. 361-370, 1995