Technische Universität Wien

MAGISTERARBEIT

Multikriterielle Routenplanung mit Tabu-Suche

Ausgeführt am Institut für
Wirtschaftsmathematik
der Technischen Universität Wien

unter der Anleitung von

A.o. Univ.Prof. Dr. Alexander Mehlmann
und Dr. Josef Haunschmied

durch

Dipl.Ing. Leonhard Zehentner
Gärtnergasse 8/3B/5
2103 Langenzersdorf

Wien, im März 2006. .....

Leonhard Zehentner

Multikriterielle Routenplanung mit Tabu-Suche

(Magisterarbeit)

mailto:leonhard.zehentner@gmx.at

Ich danke Marion und Hans-Peter, die sich die Mühe gemacht haben, meine Entwürfe durchzulesen und Unklarheiten aufzudecken.

Ich danke meinem Betreuer, Herrn Dr. Josef Haunschmied für die ausgezeichnete Zusammenarbeit.

Kurzfassung

Was ist die *beste* Route von A nach B?
Antwort des Controllers: *die Billigste*
Antwort des Pendlers um 06:35 früh: *die Schnellste*
Antwort des Grünwählers: *hauptsache mit dem Bus*

Es gibt viele Möglichkeiten eine Route zu beurteilen. Nimmt man nur ein Kriterium als Entscheidungsgrundlage, dann ist die Berechnung der *besten* Route aus computationaler Sicht einfach. Sind mehrere Kriterien gleichzeitig zu beachten wird es schwieriger. Die Länge der Fahrzeit kann eine billige Route teuer machen, die kurze Route über einen Pass kann den Schadstoffausstoß empfindlich erhöhen. Bei vielen Kriterien bewirkt ein Verbessern des einen Kriteriums ein Verschlechtern des anderen. Es ergeben sich aus mathematischer Sicht folgende 3 Situationen:

1. Die Lösungen X und Y sind gleich gut, das heisst: X und Y unterscheiden sich in keinem Kriterium.

2. Die Lösung X ist besser als Y, das heisst: X ist in mindestens einem Kriterium besser als Y aber Y in keinem besser als X.

3. Die Lösungen X und Y sind unvergleichbar, das heisst: X ist in mindestens einem Kriterium besser als Y und umgekehrt.

Ist eine Lösung gegenüber allen anderen Lösungen unvergleichbar, dann wird sie auch *pareto-optimal* genannt. Alle pareto-optimalen Lösungen zusammen werden *Pareto-Menge* genannt. In dieser Arbeit versuche ich festzustellen, ob *Tabu-Suche* (siehe Kapitel 2) geeignet ist, eine möglichst große Menge von pareto-optimalen Lösungen zu finden. Diese Pareto-Menge kann für kleine Aufgaben in vertretbarer Zeit exakt bestimmt werden. Bei größeren Aufgaben ist das nicht mehr möglich (siehe Kapitel 1.3). Um die Effizienz des Algorithmus zu testen, habe ich ein C++Programm geschrieben und mit diversen mittelgroßen Problemen gefüttert.

Da sich eine frühere Diplomarbeit ([2]) mit demselben Problem über einen anderen Algorithmus (evolutionären) genähert hat, konnte ich entsprechende Vergleiche durchführen. In dieser erwähnten Arbeit wird auch die Problematik von hybriden Lösungen aufgeworfen. Hybride Lösungen sind konvexe Kombinationen von Lösungen. Kapitel 3.5 und 4.1 befassen sich damit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Multikriterielle Routenplanung

Multikriterielle Routenplanung befasst sich mit dem Problem, dass es in einem typischen geographischen Raum mehrere Möglichkeiten gibt, von einem Ort A zu einem Ort B zu gelangen. Das inkludiert sowohl die Möglichkeit, unterschiedliche Verkehrsmittel zu verwenden, als auch über verschiedene Zwischenstationen das Ziel zu erreichen. Jeder dieser unterschiedlichen Wege oder Verkehrsmittel hat gewisse Vor- und Nachteile. Auf dem einen Weg ist man schneller unterwegs, muss allerdings Maut zahlen. Ein anderer Weg ist kurz und ohne Maut, führt aber über eine Bergstraße und verursacht deshalb einen hohen Treibstoffverbrauch und hohen Schadstoffausstoß.

Wie können diese unterschiedlichen Kriterien auf ein handhabbares Modell reduziert und mit mathematischen Methoden untersuchen werden?

1.1 Das Modell

Als Modell zur Untersuchung der Multikriteriellen Routenplanung wird ein Graph verwendet. Die Verwendung eines Graphen als abstraktes Modell wurde bereits 1736 von Leonhard Euler zur Beschreibung des Königsberger Brückenproblems eingeführt bzw. verwendet. Auf eine detaillierte Beschreibung der Elemente eines Graphen, deren Bedeutung und Zusammenwirken werde ich an dieser Stelle verzichten. Ich gehe davon aus, dass es dem Leser vertraut ist und verweise z.B. auf [10].

Alle potentiellen Abzweigungen des Gebietes, Städte oder Autobahnkreuzungen, werden auf Knoten, die Straßen dazwischen werden auf Kanten abgebildet. Diese Kanten verbinden die Knoten und sind i.A. gerichtet. Für jede dieser Kanten (Straßen) wird eine Bewertung bzgl. der Kriterien erstellt. Jede Teilstrecke des zu untersuchenden Gebiets erhält eine Bewertung für jedes betrachtete Kriterium (siehe Abbildung 1.1). Für unterschiedliche Verkehrsmittel auf der gleichen Teilstrecke gäbe es die Möglichkeit mehrere Kanten zwischen Knoten zuzulassen. Diese Vorgehensweise wird hier allerdings nicht gewählt. Wenn die Nutzung mehrerer Verkehrsmittel möglich ist, werden an den Enden der Teilstrecke Schattenknoten gebildet. Zwischen den Originalknoten und den

Schattenknoten gehen jeweils Kanten. Diese Kanten tragen als Bewertungen die Umsteigkosten auf das jeweilige Verkehrsmittel. Man kann, um von Wien nach

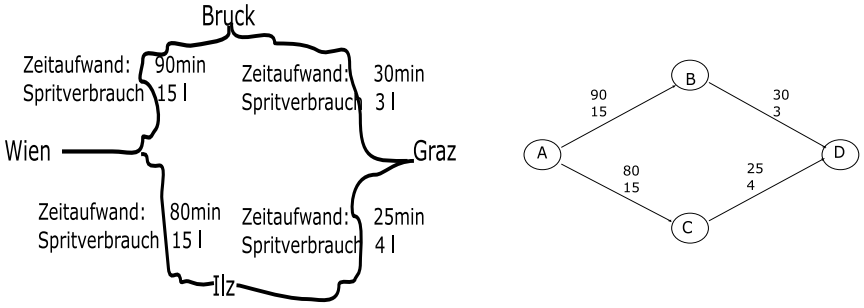

Abbildung 1.1: Wege mit 2 Bewertungen und das Modell

Graz zu gelangen, die Route über die Südautobahn wählen und damit schneller sein, oder man entscheidet sich für die Route über Bruck und spart Treibstoff.

Einwand (Kenner der regionalen Gegebenheiten): Die Route über Bruck ist auf jeden Fall länger und spart keinen Treibstoff!

Der Einwand ist berechtigt und zeigt einen der Schwachpunkte dieser Vorgehensweise. Viele Bewertungen sind auf Schätzungen angewiesen. Vielleicht nicht gerade in diesem Fall, aber nimmt man z.B. das Attribut *abwechslungsreich* als weiteres (zugegeben eher touristisch angehauchtes) Kriterium einer Fahrt dazu, dann ist die Objektivität kaum mehr gegeben.

Ich gehe in meinem Algorithmus davon aus, dass die Bewertungen der Realität nahekommen und berücksichtige nicht die Möglichkeit mit Hilfe von statistischen Methoden die Abweichung bei Schätzungen besser handzuhaben.

1.2 Mathematischer Hintergrund

Wie kann man nun die einzelnen Wege des oben gezeigten Beispiels miteinander vergleichen? Der erste Schritt besteht darin, die Bewertungen aller Teilstrecken aufzusummieren. Für jede der Teilstrecken von A nach D (von Wien nach Graz) über C (Bruck) wird der Zeitaufwand und der Treibstoffverbrauch aufsummiert. Für die Strecke Wien-Graz über Bruck ergibt das:

$$\begin{pmatrix} Zeitaufwand \\ Spritverbrauch \end{pmatrix} = \begin{pmatrix} 90 \\ 15 \end{pmatrix} + \begin{pmatrix} 30 \\ 3 \end{pmatrix} = \begin{pmatrix} 120 \\ 18 \end{pmatrix}$$

Für die Strecke über Ilz ergibt das:

$$\begin{pmatrix} 80 \\ 15 \end{pmatrix} + \begin{pmatrix} 25 \\ 4 \end{pmatrix} = \begin{pmatrix} 105 \\ 19 \end{pmatrix}$$

Wenn man die beiden Werte vergleicht, ergibt sich folgende Situation: Der Zeitaufwand spricht für die Strecke über Ilz, der geringere Treibstoffverbrauch für die Strecke über Bruck.

Ich löse mich jetzt einmal von diesem sehr kleinen Beispiel und stelle mir vor was passiert, wenn sehr viele solcher unterschiedlicher Strecken analysiert und verglichen werden. Folgende Situationen können eintreten:

Beim Vergleich zweier Strecken (X und Y) ist eine der beiden Strecken (sagen wir die Strecke X) eindeutig besser. Eindeutig heisst in diesem Zusammenhang, in jedem Kriterium ist X mindestens so gut wie Y, aber in mindestens einem dieser Kriterien ist X echt besser als Y.
In diesem Zusammenhang sagt man auch X dominiert Y und schreibt: $X \succ Y$

Den zweiten Fall haben wir schon mit dem einfachen Beispiel kennengelernt. Wenn beide Strecken jeweils mindestens ein Kriterium besitzen, in dem sie besser sind als die andere Strecke. In diesem Fall kann man von keiner der beiden Strecken behaupten, sie sei besser als die andere. Es könnte gerade das eine Kriterium ausschlaggebend für eine Entscheidung zugunsten dieser Strecke sein. Da keine von beiden Strecken besser ist als die Andere, sind beide unvergleichbar ($X <> Y$).

Der letzte Fall ist der einfachste, beide Strecken sind gleich im Sinne der Kriterien. Das heisst, sie sind zwar nicht gleich im Sinne der Streckenführung, benutzen also unterschiedliche Kanten, die Endsumme ist allerdings in allen Kriterien gleich. In normalen Testgraphen ist dieses Ereignis eher selten und bei den Tests überhaupt nur bei sehr kleinen Beispielen aufgetreten.

Es gibt einen weiteren Fall von Dominanz. Dieser bezieht sich aber nicht auf das bilaterale Verhältnis zwischen zwei Strecken, sondern auf das Verhältnis mehrerer Strecken zu einer. Dabei dominiert die konvexe Kombination von mehr als einer Strecke eine Andere. Dieser Fall wird in den Kapiteln 3.5 und 4.1 behandelt.

Nachdem alle Wege analysiert wurden, bleibt eine Menge von Wegen übrig, die nicht von anderen dominiert wird. Nimmt man einen dieser Wege und vergleicht ihn mit einem beliebigen Anderen, dann dominiert keiner von beiden den jeweils Anderen. Diese Menge nennt man Pareto- Menge.

Ich möchte an dieser Stelle die entsprechende Stelle aus Wikipedia zitieren ([13], eigener Text kursiv gesetzt):

> Bei vielen Optimierungsaufgaben lassen sich mehrere, voneinander grundsätzlich unabhängige Zielsetzungen definieren, zum Beispiel bei *Wegstrecken die Zeit, der Spritverbrauch, der Schadstoffausstoß*. Es ist hier oft nicht möglich, alle Ziele gemeinsam zu optimieren, man kann sich zum Beispiel in der Situation befinden, dass man *den Spritverbrauch nur vermindern kann (eine Verbesserung), wenn gleichzeitig der Zeitaufwand steigt (eine Verschlechterung)* . Das übliche Vorgehen zur Behandlung solcher Aufgaben ist es, die interessierenden Ziele als Teilziele aufzufassen und sie mittels Gewichtungsfaktoren zu einer gemeinsamen Zielfunktion zusammenzufassen. Man erhält auf diese Weise ein monokriterielles Problem. Dies löst man mit einem der unter Operations Research genannten Verfahren und bestimmt eine optimale Lösung für die gemeinsame Zielfunktion.
>
> Bei nicht ineinander umrechenbaren Zielgrößen, wie etwa im

gegebenen Beispiel, sind die anzusetzenden Gewichtungsfaktoren willkürlich und in bestimmten Rahmen subjektiv. Hierdurch ergibt sich auch eine entsprechende Willkürlichkeit bei der Identifikation der gesuchten "besten"Lösung des Optimierungsproblems. Eine sinnvolle Vorgehensweise ist in solchen Fällen die separate Optimierung für alle möglichen Kombinationen von Gewichtungsfaktoren. Dabei wird man in der Regel nicht eine einzelne beste Lösung finden, da die Zielkriterien meist miteinander in Konflikt stehen (wie oben der minimale Spritverbrauch und der Zeitaufwand).

Da keine eindeutig beste Lösung definiert ist, bestimmt man eine Menge von Lösungen des Optimierungsproblems, bei der eine Verbesserung eines Zielfunktionswertes nur noch durch Verschlechterung eines anderen erreicht werden kann, also die Menge optimaler Kompromisse. Diese Lösungsmenge bezeichnet man als Pareto-Menge oder Pareto-Optimum des zugrunde liegenden Optimierungsproblems; deren Elemente als pareto-optimal. Es ist zu beachten, dass die Pareto-Menge im Allgemeinen nicht vollständig durch die Variation von Gewichtungsfaktoren bestimmt werden kann.

Ist die Pareto-Menge des gegebenen Optimierungsproblems erst einmal gefunden, können subjektive Einschätzungen über die Wichtigkeit der einzelnen Teilziele (verschiedene Gewichtungsfaktoren) angegeben werden. Die Paretomenge enthält dann für beliebige relative Teilzielgewichtungen jeweils mindestens eine Lösung, die bei dieser Gewichtung optimal ist.

Zusammengefasst bzw. anders formuliert heisst das:
Wenn beim Übergang von einer Lösung zu einer beliebigen anderen Lösung eine Verbesserung in einem Kriterium nur durch Verschlechterung in einem anderen erreicht wird, dann ist die Lösung pareto-optimal. Die Zusammenfassung aller pareto-optimalen Lösungen (Wege) heißt Pareto-Menge.

Ziel dieser Arbeit ist es, möglichst effizient die Pareto-Menge zu bestimmen. Im nächsten Kapitel sehen wir, dass es meist nicht einfach und in manchen Fällen sogar extrem aufwendig ist, die gesamte Pareto-Menge zu bestimmen.

1.3 Komplexität

Bei der Analyse eines Problems und der Beurteilung eines Algorithmus zur Lösung des Problems ist es wesentlich, die Komplexität des zugrundeliegenden Problems zu kennen. In unserem Fall ist es also wichtig zu wissen, mit wie vielen pareto-optimalen Lösungen zu rechnen ist. Ein recht einfaches Beispiel (Abbildung 1.2 zeigt, dass die Anzahl sehr groß werden kann.

Die Abbildung ist folgendermaßen zu verstehen: Ich versuche den besten Weg vom linken Knoten (0) zum rechten Knoten (2n+1) zu finden. Pro Kante sind zwei Bewertungen zu berücksichtigen. Die Bewertungen der Kanten verhalten sich nach folgendem Muster: Eine Kante die von einem Knoten der Ebene i zu einem Knoten der Ebene i+1 der oberen Linie führt, hat im ersten Kriterium eine Bewertung 2^i. Im zweiten Kriterium genau das doppelte (z.B. die

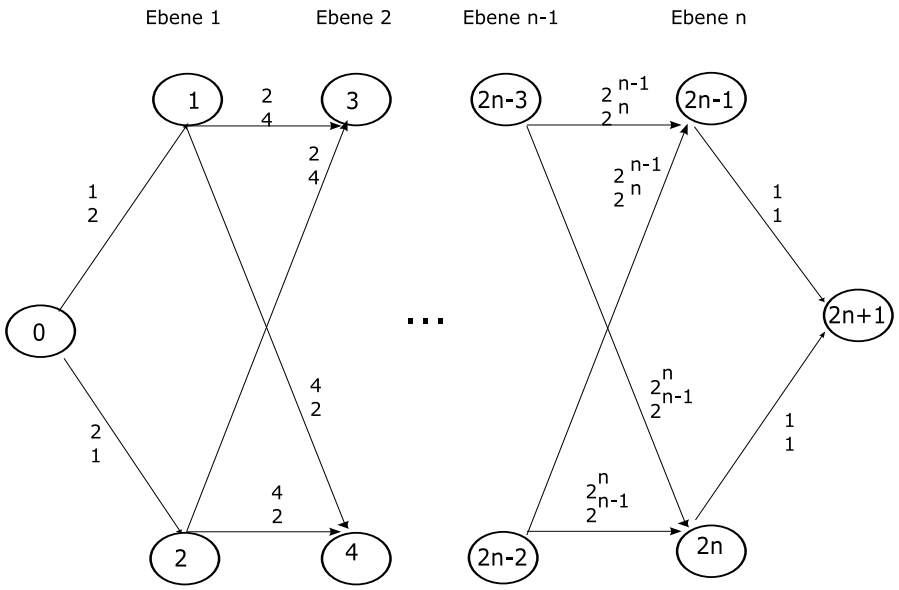

Abbildung 1.2: Komplexität des Problems

Kanten der Ebene 1, nämlich $1 \rightarrow 3$ und $2 \rightarrow 3$, haben die Bewertung 2^1 bzw. 2^2). Für Kanten die zu einem Knoten der unteren Linie führen gilt genau das Umgekehrte. Sie haben im ersten Kriterium eine doppelt so große Bewertung wie im zweiten Kriterium.

Daraus ergibt sich, dass der beste Weg bzgl. des ersten Kriteriums über die obere Linie führt und der beste Weg bzgl. des zweiten Kriteriums über die untere.

Wenn man den gesamten Weg auf der oberen Linie wandert, dann ergeben sich für die Gesamtbewertung folgende Daten:

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 2 \\ 4 \end{pmatrix} + ... + \begin{pmatrix} 2^{n-1} \\ 2^n \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2^n \\ 2^{n+1} - 1 \end{pmatrix}$$

Für die Bewertungen auf der unteren Linie ergeben sich die selben Werte nur sind die Einträge in den Vektoren vertauscht:

$$\begin{pmatrix} 2 \\ 1 \end{pmatrix} + \begin{pmatrix} 4 \\ 2 \end{pmatrix} + ... + \begin{pmatrix} 2^n \\ 2^{n-1} \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2^{n+1} - 1 \\ 2^n \end{pmatrix}$$

Ich betrachte jetzt nur die Summe der ersten Bewertung auf einem beliebigen Weg. Wenn ich mich vom linken zum rechten Knoten bewege, dann führt mein Weg entweder über einen Knoten auf der oberen Linie oder auf der unteren Linie. Jedesmal wenn ich auf Ebene i statt den oberen den unteren Knoten nehme, dann bekomme ich statt 2^i den doppelten $2 * 2^i$ Wert dazu. Letztendlich spiegelt die Endsumme exakt wieder, wo ich auf dem Weg vom linken Knoten zum rechten Knoten die obere Linie benutzt habe und wo die untere.

Die Summe der Bewertungen der unterschiedlichen Lösungen geht von 2^n bis $2^{n+1} - 1$. Das sind insgesamt 2^n verschiedene Bewertungen.

Mit der zweiten Bewertung verhält es sich genau umgekehrt. Diese beträgt bei ausschließlicher Benutzung der oberen Linie $2^{n+1} - 1$ und reduziert sich je nach Anzahl der Knoten im Weg, die sich auf der unteren Linie befinden und nach Ebene auf 2^n.

Wenn ich jetzt zwei verschiedene Wege betrachte, dann stelle ich fest, dass der erste Weg entweder nach dem ersten Kriterium besser ist, dann muss er nach dem zweiten Kriterium schlechter sein, oder er ist nach dem ersten Kriterium schlechter, dann muss er nach dem zweiten Kriterium besser sein.

Beide Wege sind also zueinander pareto-optimal. Letztendlich sind *alle* Wege in diesem Graphen pareto-optimal. Wie leicht ersichtlich, gibt es in diesem Graphen 2^n verschiedene Wege.

Resumee: Die Anzahl pareto-optimaler Wege kann exponentiell groß sein in der Anzahl der Knoten und Kanten.

Das ist keine gute Ausgangsposition für die Auswahl eines geeigneten Algorithmus. Nicht zuletzt ist damit die Anwendung eines Algorithmus, der sich auf eine komplette Auflistung aller Wege stützt, nur bei relativ kleinen Graphen möglich.

In solchen Fällen wird dann meist auf Algorithmen aus der Klasse der Heuristiken zurückgegriffen. Im nächsten Kapitel erkläre ich einen oft verwendeten Vertreter dieser Klasse, die Tabu-Suche. Eine vereinfachte Form dieser Tabu-Suche wird dann im Kapitel 3 verwendet, um die Pareto-Menge effizient zu bestimmen.

Im Kapitel 5, "Ergebnisse und Vergleich", werde ich nochmal auf das eben besprochene Beispiel zurückkommen, da es bei diesem Beispiel möglich ist, den Aufwand des Suchalgorithmus zum (genau bekannten) Ergebnis, trotz exponentieller Anzahl von Lösungen in Relation zu setzen.

1.4 Der Gewichtsvektor

Wie bereits im Wikipedia-Zitat (1.2) angedeutet, wird i.A. eine *beste* Lösung aus der Menge aller pareto-optimalen Lösungen bestimmt, indem jeder Bewertung ein Gewicht innerhalb des Bewertungsvektors zugeordnet wird.

In dem von mir im Kapitel 3 beschriebenen Algorithmus wird zuerst versucht, die gesamte Lösungsmenge der pareto-optimalen Wege zu bestimmen. Danach wird der Vektor der besten Bewertungen pro Kriterium bestimmt. Die relative Abweichung einer Lösung zu diesem optimalen Vektor (Zielvektor) in der jeweiligen Komponente wird dann mit dem Gewicht dieser Komponente multipliziert. Die Summe dieser Abweichungen bestimmt die Qualität der Lösung. Bezogen auf das Beispiel in Abbildung 1.1:

Der Zielvektor aus $\begin{pmatrix} 120 \\ 18 \end{pmatrix}$ und $\begin{pmatrix} 105 \\ 19 \end{pmatrix}$ beträgt $\begin{pmatrix} 105 \\ 18 \end{pmatrix}$

Bei einer Gewichtung von z.B. jeweils 0.5 beträgt die Summe der relativen

Abweichungen von Lösung 1:

$$(120 - 105)/120 \ * \ 0.5 \ + \ (18 - 18)/18 \ * \ 0.5 \ = \ 0.0625$$

Für die zweite Lösung ergibt sich der Wert:

$$(105 - 105)/105 \ * \ 0.5 \ + \ (19 - 18)/18 \ * \ 0.5 \ = \ 0.0277$$

Somit wäre bei dieser Gewichtung die zweite Lösung die bessere.

Die eben beschriebene Methode ist eine Möglichkeit, um aus vielen Wegen einen auszuwählen.

Erstes Ziel dieser Arbeit ist allerdings nicht, diese Auswahl zu treffen, sondern eine möglichst große Anzahl pareto-optimaler Wege zu suchen und zu finden. Ziel des C++Programmes, das zu dem Zwecke der Evaluierung der Tabu-Suche im Kontext der multikriteriellen Routenplanung geschrieben wurde, ist also nicht von vorneherein die beste Route (Weg) zu finden, sondern eine große Auswahl vorzubereiten. Diese Vorgehensweise wird in Kapitel 3 beschrieben. Der klassische Tabu-Suche-Algorithmus, der das Finden einer besten Lösung nach *irgendeinem* vorgegebenen Bewertungskalkül vornimmt, wird im nächsten Kapitel gezeigt.

Kapitel 2

Klassische Tabu-Suche

In diesem Kapitel wird eine kurze Einführung in die Tabu-Suche für klassische NP-harte Probleme ([12] gemacht. In klassischen NP-Problemen geht es darum festzustellen, ob eine Lösung für ein Problem existiert, die gewisse Schranken einhält. Die Problematik für alle Algorithmen, ist dabei meistens die sehr große Anzahl möglicher Kandidaten, die untersucht werden müssen. Die Anzahl dieser Kandidaten wächst exponentiell in der Größe der Probleminstanz. Charakteristisch für diese Art von Problemen ist, dass sehr schnell nach Überschreiten einer Schranke, der Rechenaufwand für die exakte Lösung nicht mehr getätigt werden kann.

In diesem Kapitel beschreibe ich Tabu-Suche, wie sie allgemein bei NP-schwierigen Problemen angewandt wird. In meinem Algorithmus zur Bestimmung von pareto-optimalen Lösungen ohne festgelegten Gewichtsvektor habe ich auf einige, der in diesem Kapitel erwähnten Charakteristika, verzichtet.

Tabu-Suche ist eine Metaheuristik zur Lösung von komplexen und rechenintensiven Problemen. Beispiele für diese Art von Problemen sind neben vielen Anderen: Quadratic Assignment Problem (QAP), Travelling Salesperson Problem, Rucksackproblem (Kapitel 2.4). Bei Problemen dieser Art wird eine Zielfunktion definiert die maximiert/minimiert werden soll:

Rucksackproblem Finde jene Items, die in den Rucksack passen und ein *Maximum* an Profit ergeben.

Travelling Salesperson Problem Finde eine Route durch einen Graphen, die alle Knoten besucht und ein *Minimum* an Weg verbraucht.

Im Allgemeinen werden diese Probleme als NP-hart bezeichnet. NP-hart bedeutet, dass ein Algorithmus, welcher eine Probleminstanz dieser Klasse exakt löst, mindestens vom Grade nichtdeterministisch polynomiell ist und daher vermutlich auch kein deterministisch polynomieller Algorithmus existiert. Was heißt das konkret? Nehmen wir ein QAP-Problem mit einer Instanzgröße von ca. 50. Wenn wir in der Lage sind, diese Probleminstanz exakt zu lösen, dann kann es passieren, dass bereits eine ähnliche Instanz der Größe 100 auf absehbare Zeit unlösbar bleibt. Die Anzahl der zu untersuchenden möglichen Lösungen

steigt exponentiell in der Größe des zugrundeliegenden Problems und ist damit in vernünftiger Zeit nicht mehr berechenbar.

Für Probleme dieser Art muss man sich Algorithmen ausdenken, die zwar das Problem nicht exakt lösen können, aber in vertretbarer Zeit *gute* Lösungen liefern. Eines dieser Verfahren ist Tabu-Suche.

2.1 Arbeitsweise der Tabu-Suche

Basis der Tabu-Suche ist eine Heuristik, wie z.B. hill climbing (lokale Suche). Hill climbing geht von einer beliebigen Lösung aus und schaut sich in der Nachbarschaft um. Gibt es in der Nachbarschaft eine Lösung die besser ist als die momentane, dann wird an dieser Stelle (bei dieser Lösung) mit der Suche fortgesetzt. Auf diese Weise wird ein (lokales) Maximum erreicht.

Setzt man jetzt die Suche fort, dann muss man in Kauf nehmen, dass die Lösung wieder schlechter wird. Ausserdem muss man verhindern, dass man sich nach dem einmaligen Wegbewegen vom lokalen Maximum sich diesem aufgrund der "hill climbing Philosophie" wieder nähert. Beides wird erreicht durch Verwenden von Tabu-Listen (2.5).

Tabu-Suche kann man also vereinfacht folgendermaßen beschreiben: Irgendwo im Suchraum wird gestartet. Es wird das nächste lokale Maximium angestrebt. Ist dieses erreicht, wird trotzdem weitergesucht. Durch geschicktes Steuern mit Hilfe von Tabu-Listen wird einerseits verhindert, dass bereits besuchte lokale Maxima nochmals angesteuert werden, andererseits in hoffnungsvollen Gebieten intensiver gesucht.

2.2 Suchraum

Der Suchraum über dem die Heuristik abläuft, ist der Raum aller möglichen Lösungen, die während der Suche erreicht werden können ([3]). Angenommen eine Lösung wird durch einen binären Vektor der Dimension n repräsentiert. Der Suchraum ist dann die Menge aller möglichen Vektoren, also $\{0,1\}^n$. Eigentlich müssten dabei alle Vektoren ausgeklammert werden, die eine ungültige Lösung darstellen. Da es aber auch Varianten von Tabu-Suche gibt, bei denen temporär auch bei ungültigen Lösungen Station gemacht wird, ist i.A. die Abschätzung $\{0,1\}^n$ für die Größe des Suchraumes akzeptabel.

2.3 Lokale Suche als Basisheuristik

Lokale Suche ist ein Algorithmus, bei dem auf möglichst effiziente Weise das nächste lokale Maximum angesteuert wird. Dabei werden alle ausgewählten Nachbarn (siehe 2.4) zur derzeit bestehenden Lösung betrachtet. Ist der *geeignetste* aller dieser Nachbarn nicht in einer der Tabu-Listen, so wird bei diesem Nachbarn fortgesetzt. Nach einigen Schritten (von der Art der Nachbarschaft und vom Suchraum abhängig) wird auf diese Weise ein (lokales) Maximum erreicht.

Dabei gibt es mehrere Möglichkeiten den geeignetsten Nachbarn zu bestimmen ([5]):

2.3.1 best improvement

Alle Nachbarn werden betrachtet bzw. berechnet und der Beste wird genommen.

2.3.2 next improvement

Die Nachbarschaft wird nach einer festen Reihenfolge durchsucht. Die erste Lösung, die mindestens so gut ist wie die derzeitige, wird genommen.

2.3.3 random neighbour

Die Nachbarschaft wird zufällig durchsucht. Der erste Lösung, die mindestens so gut ist wie die Aktuelle, wird genommen.

2.4 Nachbarschaft

Wie in [3] beschrieben, ist der Begriff der Nachbarschaft einer der zentralen Punkte bei allen Heuristiken und Metaheuristiken.

Nachbarn zu einer bestimmten Lösung sind alle jene Lösungen, die in einem Schritt, in einer Transformation erreicht werden können. Es gibt viele Möglichkeiten eine Transformation zu definieren. Umgelegt auf das Beispiel des binären Vektors, wäre etwa das Kippen eines einzigen Bits eine geeignete (und vermutlich die kleinstmögliche) Transformation.

Betrachten wir als Beispiel das berühmte Rucksackproblem. Der Bit-Vektor indiziert das Vorhandensein eines bestimmten Items im Rucksack. Es ist offensichtlich sinnlos, aus einer gültigen Lösung ein Item rauszunehmen, da es nur die Lösung verschlechtert. Es bietet sich der Austausch von zwei Items an. Ein Item wird aus dem Rucksack entfernt und durch ein Neues ersetzt. Am Beispiel des Bit-Vektors: ein Bit switcht auf 0, im Gegenzug wird ein anderes von 0 auf 1 verändert.

Nachteil: aus einer Nachbarschaft der Größenordnung $O(n)$ wird eine der Größenordnung $O(n^2)$.

Vorteil: Mit einer Transformation, einem lokalen Schritt wird eine Verbesserung erreicht.

Im Fall der Nachbarschaft als reines Entfernen oder Hinzufügen eines Items, muss entweder eine temporäre Verschlechterung (zuerst fällt ein Item weg) in Kauf genommen werden, oder temporär eine ungültige Lösung (zuerst kommt ein Item dazu).

Alles in allem heißt das: Die konkrete Definition der Nachbarschaft beeinflusst in erheblichem Ausmaß den Erfolg bzw. Fortschritt des Algorithmus.

2.5 Tabu-Listen

Mit Hilfe von Tabu-Listen, auch *short memory* genannt, wird verhindert, dass bereits besuchte Lösungen wiederholt besucht werden.

Selbstverständlich kann eine Tabu-Liste aus Speicherplatzgründen nicht beliebig groß werden. Aus diesem Grund wird meistens nicht eine komplette Lösung mit allen Attributen gespeichert. Oft gilt auch, dass bei Speicherung aller Attribute schon das Feststellen, ob eine Lösung tabu ist oder nicht, sehr lange dauert. Im Beispiel Bit-Vektor immerhin $O(n)$. Dabei ist es zumeist einfacher, platzsparender und schneller, wenn man sich nur jenes Item merkt, das erst vor kurzem entfernt/aufgenommen wurde. Diese Items (Bits) dürfen dann eine gewisse Zeit lang nicht verändert werden.

Nachteil dieser Vorgehensweise: Gewisse Lösungen sind temporär nicht erreichbar.

2.6 Terminationskriterien

Beliebte Terminationskriterien sind die folgenden aus [4] übernommenen.
Abbruch des Algorithmus erfolgt:

- nach einer fixen Anzahl von Iterationen
- nach einer Anzahl von Iterationen ohne Verbesserung der Zielfunktion
- nach dem Erreichen eines bestimmten Wertes in der Zielfunktion

Oft wird auch nach einer bestimmten Anzahl von Phasen (z.B.: Multistart 2.8) abgebrochen, wobei jede einzelne Phase durch ein oben genanntes Kriterium beendet wird.

2.7 Aspirationskriterium

Während der Suche nach einem lokalen Maximum kann es passieren, dass eine Lösung tabu ist, obwohl sie besser ist als alle bisher Gefundenen. In diesem Fall wird natürlich der Tabu-Status übergangen und die Lösung akzeptiert.

2.8 Multistart

Eine Methode, die bei praktisch allen Heuristiken möglich ist. Man bestimmt mehrere Startpunkte als Anfangslösung und startet sukzessive von diesen. (Diversifizierung, wie sie im Kapitel 3.4 vorgestellt wird, ist im Grunde genommen eine Art Multistart) Durch geeignete Wahl dieser Punkte wird eine größere Streuung innerhalb des Suchraums erreicht.

2.9 Probabilistische Tabu-Suche

Im Regelfall wird beim Durchsuchen der Nachbarschaft jeder Kandidat bewertet. Dies kann mit großem computationalem Aufwand verbunden sein (siehe $O(n^2)$ in 2.4). Um dem zu entgehen, wird bei einer Variante von Tabu-Suche nur eine zufällige Auswahl aus der gesamten Nachbarschaft gewählt. Das vermindert erheblich den Rechenaufwand, kann allerdings dazu führen, dass gerade die bestmögliche Lösung nicht gefunden wird (Ausweg siehe 2.10). Der probabilistische Anteil dieser Variante dient oft als antizyklische Prävention und erlaubt damit kürzere Tabu-Listen ([3]).

2.10 Intensivierung

Intensivierung in Tabu-Suche heißt, man merkt sich im Laufe des Algorithmus Konstellationen, die vielversprechend sind, z.B.: die besten n Lösungen. Gewisse Attribute dieser Lösungen werden fixiert und eine neue Suche gestartet.

Eine weitere Möglichkeit der Intensivierung ist die Änderung der Nachbarschaft im Zuge der Intensivierung. Die besten Lösungen werden im Schnellverfahren (randomisierte Nachbarschaft) gefunden. Anschließend wird von diesen Lösungen ausgehend die jeweils komplette Nachbarschaft durchsucht.

2.11 Diversifizierung

Im Prinzip das Gegenteil von Intensivierung. Tabu-Suche ist anfällig dafür, im Suchraum zu lokal zu agieren. D.h., große Teile des Suchraums werden überhaupt nicht erforscht.
Es gibt mehrere Möglichkeiten dem zu begegnen ([3]):

- restart diversification
  Fixieren von selten benutzten Komponenten in Lösungen z.B.: Bits im Bit-Vektor, die selten in Lösungen vorkommen, oder Kanten, die bei der Bildung von pareto-optimalen Wegen bisher nicht berücksichtigt wurden
- continuous diversification
  Bestrafung von oft verwendeten Komponenten bekannter Lösungen
- strategic oscillation
  Anpassen der Gewichte bei ungültigen Lösungen (2.12)

2.12 Suche über ungültige Lösungen

Im Laufe des Suchprozesses geschieht es oft, dass Lösungen nicht berücksichtigt werden können, weil sie eine absolute Bedingung verletzen. Das kann ein Bit-Vektor sein, bei dessen Umsetzung in Items der Rucksack zu voll wird. Das kann eine Rundreise beim TSP-Problem sein, wo ein Knoten zweimal oder nie besucht wird. Verzichtet man jetzt komplett auf diese *ungültigen Lösungen*, dann schränkt man u.U. den Suchraum so ein, dass andere gültige Lösungen

kaum oder nur schwer zu erreichen sind. Diese Lösungen fehlen möglicherweise auf dem Weg zur besten Lösung, die damit nie erreicht wird.

Besser ist es, diesen ungültigen Lösungen bzw. Teilen dieser Lösungen (Attributen) Strafkosten aufzubrummen. Mit Hilfe dieser Strafkosten erreicht man, dass sich der Algorithmus eher wieder in einen gültigen Bereich begibt.

Eine Möglichkeit die Größe der Strafkosten zu steuern ist, sie dem Suchprozess anzupassen. Je länger der Algorithmus im ungültigen Bereich operiert, desto größer werden die Kosten und umgekehrt.

2.13 Hilfsfunktionen zur Performanceverbesserung

Bei manchen Problemen ist die Berechnung der Zielfunktion einer Lösung sehr aufwendig. Die Berechnung der Funktion für alle Nachbarn der momentanen Lösung kann dabei die entscheidende Performancebremse sein. In diesem Fall kann sich die Verwendung einer Ersatz- bzw. Hilfsfunktion positiv auswirken. Diese Funktion soll gewissermaßen die Tendenz der eigentlichen Funktion widerspiegeln. Sie dient zur Vorselektion potentieller Kandidaten. Für solche, vorselektierte Kandidaten, wird dann die eigentliche Zielfunktion berechnet.

Kapitel 3

Der Algorithmus

Bevor ich mit dem Erklären des Algorithmus beginne, möchte ich noch einmal die Grundlagen in Erinnerung rufen. Gegeben ist ein Graph mit Knoten, die i.A. Städte bzw. Kreuzungspunkte von Verbindungsstraßen repräsentieren. Weiters sind Kanten gegeben, die Knoten verbinden. Nicht jeder Knoten hat mit jedem anderen eine Verbindung, aber ich gehe davon aus, dass zumindest jeder Knoten von jedem anderen Knoten erreichbar ist. Jede Kante wird nach einer fixen Anzahl von Kriterien bewertet. Das sind normalerweise gut objektivierbare Kriterien wie:

- Zeitaufwand um die Strecke zu bewältigen
- Treibstoffverbrauch
- Schadstoffausstoß, den man auch detaillierter aufschlüsseln kann
- Zusatzkosten für Maut oder Spezialausrüstung

Weiters sind ein Ausgangsknoten und ein Endknoten gegeben. Nun werden Wege vom Ausgangsknoten zum Endknoten gesucht, deren Gesamtbewertung möglichst gut, in unserem Fall also möglichst klein ist. Wenn möglichst viele (alle) gute Wege (pareto-optimale) gefunden wurden, dann werden sie noch mittels einer Gewichtung der Kriterien auf einen Wert reduziert. Dieser Wert gibt dann Auskunft, welcher der Wege der beste im Sinne der Gewichtung ist.

Der Algorithmus gliedert sich in 3-5 Phasen. Die erste und letzte Phase können auch übersprungen werden. Sie werden nur aktiviert, wenn das Programm mit den entsprechenden Optionen aufgerufen wird. Die erste Phase dient zur Analyse des eigentlichen Algorithmus. Die letzte zur Elimination von Wegen, die durch hybride Lösungen (4.1) dominiert werden, bzw. um diese "non-supported" Lösungen überhaupt zu erkennen.

Der Algorithmus läuft folgendermaßen ab:

1. Komplette Suche im Suchraum für eine begrenzte Zeit
2. Bestimmung der kürzesten Wege mit dem Dijkstra-Algorithmus
3. Suchen in der Nachbarschaft dieser n Wege - Intensivierung

4. Bilden neuer Wege über bisher nicht benutzte Kanten - Diversifizierung
5. Erkennen und Eliminieren (falls gewünscht) von non-supported Lösungen

3.1 Phase 1: Komplette Suche

In vielen Beispielen mit Praxisbezug ist es gar nicht nötig eine Heuristik zu implementieren. Es gibt gar nicht so viele Möglichkeiten, über verschiedene Wege von einem Punkt zum anderen zu gelangen. In diesen Fällen ist es möglich, alle Wege zu bestimmen und die pareto-optimalen herauszufinden. Dass diese Vorgehensweise aber relativ rasch an ihre Grenzen stößt, wird im Kapitel 5 dargelegt. Wichtig ist diese Phase deshalb, weil ihre Ergebnisse zum Bestimmen der Effizienz der anderen Phasen dienen.

3.2 Phase 2: Dijkstra Wege

In der schon erwähnten Vorgängerdiplomarbeit ([2]) wurde ein Algorithmus zur Bestimmung kürzester Wege erläutert. Ein Spezialfall davon ist der Dijkstra-Algorithmus (nachzulesen in [8]). In diesem Algorithmus geht es darum, den kürzesten Weg von einem Anfangsknoten zu einem (beliebigen) Endknoten in einem Graphen zu bestimmen. Kürzester Weg heißt in diesem Fall, der nach nur *einem* Kriterium beste Weg vom Anfangs- zum Endknoten. Nachdem es b Kriterien gibt, ist i.A. mit b verschiedenen Wegen zu rechnen.

Im Programm habe ich den Algorithmus nicht selber implementiert, da er von der externen Software zur Verwaltung des Graphen bereits zur Verfügung gestellt wird (Kapitel 4.2). Jeder Weg, der mittels dieser Methode gefunden wird, ist automatisch pareto-optimal, da es keinen anderen Weg geben kann, der in diesem Kriterium besser ist.

Achtung: Der Algorithmus ist nicht eindeutig in dem Sinne, dass er bei Auswahl von mehreren gleich guten Kanten eine *beliebige* Kante auswählt. Dadurch kann es vorkommen, dass es mehrere Wege gibt, die alle in dem betrachteten Kriterium optimal sind. Der Algorithmus weiß normalerweise nichts von den übrigen Kriterien und kann diese deshalb (und vor allem aus technischen Gründen) nicht berücksichtigen. Somit kann es mehrere Wege geben, die in dem speziell betrachteten Kriterium optimal sind. Der Algorithmus wählt eine Kante *zufällig* aus mehreren optimalen aus. Der Gesamtweg ist zwar in genau diesem Kriterium optimal, es kann aber auch noch andere geben, die gleich gut sind. Wenn einer dieser nach Kriterium 1 gleich guten Wege im Kriterium 2 besser ist, dann dominiert er den zuerst gefundenen. Dem könnte man vorbeugen, indem man nach dem Durchlauf von Dijsktra sukzessive jede einzelne Kante des gefundenen Weges verschlechtert und danach Dijkstra noch einmal aufruft. Wenn weitere Dijkstra-Wege existieren, dann werden sie auf diese Weise ermittelt und man kann sie bzgl. der anderen Kriterien bewerten.

3.3 Phase 3: Nachbarschaft/Intensivierung

Nach dem Aufruf des Dijkstra-Algorithmus wird eine Intensivierung genannte Suche rund um die bisher gefundenen pareto-optimalen Wege gestartet. Diese Suche ist bereits Teil der als Grundlage dieser Arbeit verwendeten Tabu-Suche zu sehen.

Die wesentliche Grundlage dieser Intensivierung ist der Begriff der Nachbarschaft. Ich nehme als Ausgangspunkt einen der bisher gefundenen pareto-optimalen Wege. Von diesem ausgehend, versuche ich einen neuen Weg zu finden, indem ich in der *Nähe* des Ausgangsweges suche. Die Nähe ist folgendermaßen definiert: Nimm einen beliebigen Knoten des Ausgangsweges. Verfolge* eine *neue* adjazente Kante zu dessen inzidenten Knoten. Wenn sich dieser Knoten im restlichen Teil des alten Weges befindet, dann ist ein neuer Nachbarweg gefunden. Diesen neuen Weg der Tiefe 0 (0 neue Knoten wurden verwendet) würde man allgemein wohl als *Abkürzung* bezeichnen. Wenn sich durch Verwendung einer neuen Kante nicht unmittelbar ein neuer Weg ergibt, dann setze mit dem neuen Knoten fort. Von diesem neuen Knoten können alle adjazenten Kanten betrachtet werden und mit diesen Kanten wird der mit * gekennzeichnete Teil des Algorithmus rekursiv aufgerufen.

Der rekursive Aufruf wird abgebrochen, sobald eine vorgeschriebene Tiefe entlang des neuen Weges erreicht wird. Viele Tests (z.B. 5) haben gezeigt, dass die Tiefe 3 in den meisten Fällen ausreicht um die Umgebung des alten Weges intensiv zu erforschen. Oft ist sogar ausreichend, nur bis zur Tiefe 2 suchen zu lassen. In diesem Zusammenhang muss selbstverständlich erwähnt werden, dass die Größe dieses Parameters erheblich die Performance des gesamten Algorithmus beeinflusst. Je größer die zu durchsuchende Nachbarschaft definiert wird, desto mehr potentielle Wege müssen verfolgt werden.
Abbildung 3.1 soll das Beschriebene verdeutlichen.

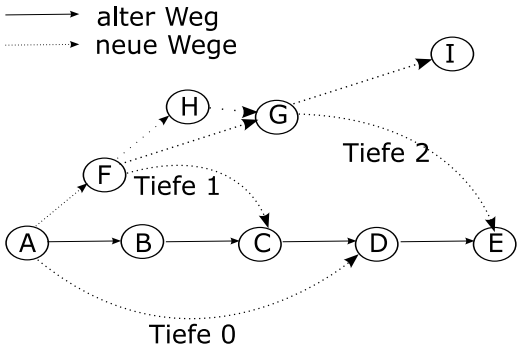

Abbildung 3.1: Wege in der Nachbarschaft

Erklärung: Der alte Weg besteht aus den Knoten A B C D E.

Ein neuer Weg der Tiefe 0 wird gefunden, indem man sich von einem beliebigen Knoten des alten Weges (in diesem Fall vom Knoten A) wegbewegt und 0 fremde Knoten verwendet, um auf den alten Weg zurückzukommen. Der neue

Weg der Tiefe 0 lautet also: A D E.

Ein neuer Weg der Tiefe 1 wird erreicht indem man sich von einem beliebigen Knoten wegbewegt und über einen neuen Knoten (F) den alten Weg betritt. Der neue Weg der Tiefe 1 lautet also: A F C D E. Das wird fortgesetzt bis zur maximalen Tiefe, in diesem Fall 2. Der neue Weg der Tiefe 2 lautet: A F G E. Andere neue Wege gibt es in diesem Beispiel nicht. Der Weg über Knoten H führt zwar auch auf den alten Weg, aber erst in der Tiefe 3 und Knoten I führt gar nicht auf den alten Weg.

Anders formuliert: Beweg dich vom alten Weg fort und versuche innerhalb einer gewissen Anzahl von Kanten wieder auf den alten Weg zurückzukommen.

Nachdem die Nachbarschaft aller Dijkstra-Wege durchsucht wurde und dabei neue pareto-optimale Wege in die Pareto-Menge aufgenommen wurden, wird die Nachbarschaftssuche auch auf die neuen Wege ausgedehnt. Diese Vorgangsweise wird solange beibehalten, bis alle Wege einmal als Ausgangspunkt der Nachbarschaftssuche verwendet wurden, also kein neuer Weg mehr gefunden wird.

Die Nachbarschaftssuche bricht ab, wenn das Maximum an Suchzeit erreicht wird. Wie ich in Kapitel 1.3 dargelegt habe, kann die Anzahl von pareto-optimalen Wegen exponentiell in der Größe der Knoten/Kanten sein. Das bedeutet: dieser Nachbarschaftsalgorithmus, angewendet auf das Beispiel aus Kapitel 1.3 würde zwar alle pareto-optimalen Wege finden, nachdem es aber exponentiell viele davon gibt, sehr lange dafür brauchen.

3.4 Phase 4: Diversifizierung

Nachdem die Nachbarschaft aller Wege intensiv untersucht wurde, bleiben immer noch Bereiche des Graphen unerforscht. Abbildung 3.4 soll das veranschaulichen. Der schraffierte Bereich ist jener Bereich, der in Phase 2 und 3 untersucht

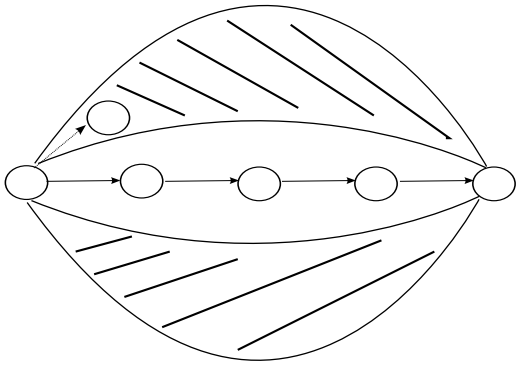

Abbildung 3.2: nicht erreichbare Wege

worden ist. Der Weg in der Mitte wird bzw. kann nicht gefunden werden, sofern er nicht bereits in der Menge der kürzesten Wege liegt und damit vom Dijkstra-Algorithmus erkannt wird.

Das passiert immer dann, wenn der Weg Teilstrecken beinhaltet, die keine Verbindung zu einem der bisher gefundenen Wege haben und mehr Kanten beinhalten, als die maximal vorgegebene Tiefe bei der Nachbarschaftssuche. Jedesmal, wenn sich der Algorithmus von einem Ausgangsweg entfernt und entlang des neuen Weges bewegt, dann wird abgebrochen, bevor er zu einem Knoten des alten Weges gelangt.
Jetzt kommt die Diversifizierung zur Anwendung.

Während des normalen Ablaufs des Algorithmus wird bei jeder Kante notiert, ob sie bereits Teil eines pareto-optimalen Weges ist. Zu diesem Zweck wird für jede Kante ein Zähler mitgeführt. Nach Durchlauf von Phase 2 und 3 werden diese Zähler ausgewertet (derzeit wird nur zwischen 0 und $\neq 0$ unterschieden). Alle Kanten mit Zählerstand 0 werden einer eigenen *Behandlung* unterzogen.

Es werden auf folgende Art neue Wege gebildet: Zuerst wird der Dijkstra-Weg vom Anfangsknoten zum Anfang der Kante gesucht. Danach wird vom Ende der Kante bis zum Endknoten im Graph ein Dijkstra-Weg gesucht. Zusammen bilden diese beiden Teilwege plus Kante einen neuen Weg vom Anfangs- zum Endknoten. Abbildung 3.3 veranschaulicht die Zusammensetzung des neuen Weges.

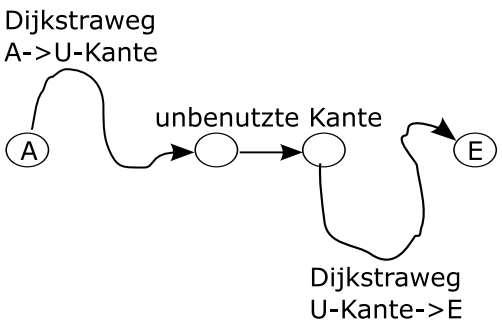

Abbildung 3.3: Neue Wege über unbenutzte Kanten

Nachdem der neue Weg gebildet wurde, gibt es mehrere Möglichkeiten weiterzumachen:

1. Der neue Weg ist pareto-optimal und wird aufgenommen.
2. Der neue Weg ist nicht pareto-optimal und es gibt keine Fortsetzung.
3. Der neue Weg ist nicht pareto-optimal aber seine Nachbarschaft wird auf pareto-optimale Wege abgesucht.
4. Für jedes einzelne Kriterium wird Punkt 1 bis 3 durchgeführt

Ad 3 und 4: Bei oberflächlicher Betrachtung scheint es offensichtlich zu sein, dass Punkt 3 und 4 die logischsten Optionen für eine Fortsetzung des Algorithmus darstellen. Nach Durchlauf vieler Testfälle hat sich allerdings herausgestellt, dass beide Punkte zwar sehr negativ die Laufzeit beeinflußen, aber kaum bessere

Ergebnisse bringen. In der Umsetzung des Programmes habe ich deshalb wieder davon Abstand genommen und folgende vereinfachte Variante realisiert:

Es wird nur ein Dijkstra-Weg gebildet und das Kriterium ist einzig die Länge des Weges. Wenn der so entstandene Weg nicht pareto-optimal ist, dann wird auch nicht in seiner Nachbarschaft gesucht.

Eine weiterführende Analyse, warum die Verwendung aller Kriterien bzw. die Suche in der Nachbarschaft nicht pareto-optimaler Wege nicht sehr erfolgreich war, wurde in dieser Arbeit nicht gemacht.

Bei welcher Art von Graphen es möglicherweise trotzdem sinnvoll ist diese Fortsetzung zu wählen, bleibt also vorerst offen. Nach einer freien Interpretation des berühmten "no free lunch" Theorems ([11]), gibt es Graphen, für die gerade dieser Algorithmus der beste wäre.

Nachdem mit Hilfe der Diversifizierung auch *entlegenere* Bereiche des Graphen untersucht wurden, wird noch einmal Phase 3, die Nachbarschaftssuche, mit den neu hinzugekommenen Wegen initiert.

Damit komme ich zum letzten Abschnitt des Algorithmus, dem Erkennen von "non-supported" Lösungen in der Pareto-Menge.

3.5 Phase 5: Konvexe Hülle

Was sind "non-supported" Lösungen?

Nicht jede pareto-optimale Lösung ist per se eine gute Lösung. Um das zu veranschaulichen, sind in Abbildung 3.4 die Bewertungsvektoren von 3 Wegen eingezeichnet, die jeweils zueinander pareto-optimal sind. Bei genauerer Betrachtung stellt sich allerdings heraus, dass der dritte Weg in Relation zu den beiden anderen Nachteile hat.

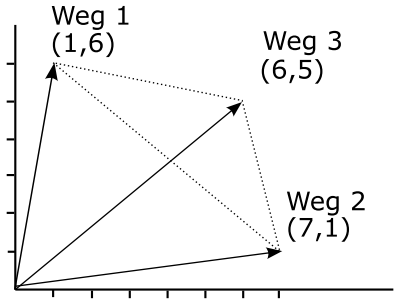

Abbildung 3.4: Bewertungsvektor eines non-supported pareto-optimalen Wegs

Weg 3 ist zwar jeweils in einem Kriterium besser als Weg 1 bzw. Weg 2 und deshalb pareto-optimal, allerdings sagt einem schon das Gefühl, dass Weg 3 im Kontext der beiden anderen betrachtet nicht gut ist. Dieses Gefühl kann durch folgende mathematisch/praktische Interpretation bestätigt werden.

Betrachtet man eine Lösung als Weg, der nicht nur einmal für einen Transport verwendet wird, sondern mehrmals oder auch beliebig oft, dann ist es

sinnvoll, Kombinationen von Wegen und deren Kriterien zu berücksichtigen. Nimmt man im Fall der Skizze oben (Abbildung 3.4) Weg 1 und Weg 2 jeweils einmal und kombiniert das Ergebnis, so ergibt sich als Gesamtbewertung: $(8, 7)$ für zwei Transporte. Zweimal den Transport über Weg 3, ergibt: $(12, 10)$. Das ist eindeutig schlechter als die Kombination der Wege 1 & 2, und wird von dieser dominiert. Wie kann man diesem Umstand Rechnung tragen und die Problematik allgemein formulieren?

Bewertungen von pareto-optimalen Wegen sind Vektoren im $b - dimensionalen$ Raum. Ich betrachte beliebige konvexe Kombinationen dieser Vektoren. Eine konvexe Kombination von Vektoren $\vec{p_1}, ..., \vec{p_n}$ sind alle Vektoren $\vec{x}$, die folgende Eigenschaft besitzen:

$$\vec{x} \; = \; \sum_{i=1}^{b} \lambda_i * p_i \; mit \; \sum \lambda_i \; = \; 1 \; und \; \forall \lambda_i : \lambda_i \geq 0$$

Im 2-dimensionalen Raum zwischen zwei Vektoren ist das einfach die Verbindungsstrecke zwischen beiden. Im dreidimensionalen mit drei Vektoren ist es die Ebene die zwischen den dreien aufgespannt wird. Sind mehr als zwei/drei Vektoren beteiligt, dann ist es die umschließende Hülle um alle Vektoren. Wenn nun eine dieser Kombinationen einen anderen Vektor dominiert, dann befindet sich der dominierte Vektor i.A. innerhalb dieser konvexen Hülle.

Was ist in Abbildung 3.4 mit "Weg 3" schiefgegangen? Warum befindet sich der Bewertungsvektor "Weg 3" nicht innerhalb der (strichliert angedeuteten) Hülle? Es liegt daran, dass die konvexe Hülle keine Rücksicht darauf nimmt, ob etwas gut oder schlecht im Sinne von möglichst weit weg vom Ursprung oder möglichst nahe beim Ursprung ist. Es ist nötig, die konvexe Hülle in Richtung *schlecht* abzugrenzen. Das geschieht, indem ich einen weiteren Vektor dazu nehme, der, im Sinne der Bewertung so schlecht ist, dass er den anderen Lösungen (Wegen) die Möglichkeit nimmt, Teil der konvexen Hülle zu sein. Dieser Vektor ist sozusagen das "Gegenteil"des Zielvektors, nämlich der jeweils schlechteste Wert pro Kriterium, genommen aus den Bewertungsvektoren der Pareto-Menge. In Abbildung 3.5 ist der beschriebene Zusammenhang dargestellt.

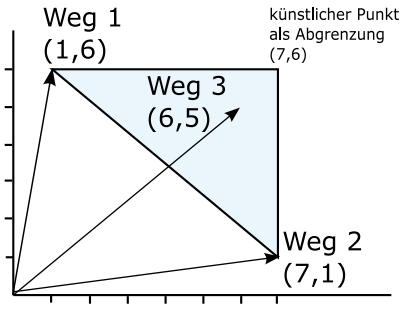

Abbildung 3.5: konvexe Hülle

Was bleibt in Phase 5 des Algorithmus noch zu tun?

Zuerst bilde ich die konvexe Hülle aus den Bewertungsvektoren und dem "sehr schlechten" Vektor. Danach sind alle Vektoren, die nicht Teil dieser Hülle sind, als non-supported erkannt. Da sich diese Vektoren innerhalb der Hülle befinden und die Hülle eine konvexe Kombination von anderen Bewertungsvektoren ist, muss es eine konvexe Kombination geben, die diese Vektoren dominiert.

Mit welchem Algorithmus diese Hülle berechnet wird bzw. welche Software diese Arbeit im Programm übernimmt, wird in Kapitel 4 beschrieben.

3.6 Tabu-Kriterien

In Kapitel 3.3 wird beschrieben, wie die Nachbarschaft eines bereits gefundenen pareto-optimalen Weges durchsucht wird. Bei dieser Suche sollte möglichst verhindert werden, dass bereits früher analysierte Wege im Graphen erneut Ziel einer Untersuchung sind. Dass dieser Mehraufwand nicht gänzlich verhindert werden kann, liegt an der enormen Anzahl möglicher Wege. Es müssten letztlich alle bereits besuchten Wege gespeichert werden. Das ist nicht möglich und deshalb beschränke ich mich in diesem Algorithmus auf folgendes Tabu-Kriterium.

Nachdem ein neuer Weg aufgrund der Nachbarschaft zu einem, bereits als pareto-optimal Erkannten, generiert wurde, wird der erste Knoten, der sich auf dem alten Weg befindet und nicht mehr benutzt wird, als Tabu-Knoten gespeichert. Im Fall der Abbildung 3.1 heißt das: Der alte Weg lautet: A B C D E, der neue A F C D E und für diesen neuen Weg wird der Knoten B als Tabu-Knoten gespeichert. Dieser Knoten darf bei der Suche nach einem weiteren neuen Weg bei diesem nicht enthalten sein. Wird nun ausgehend von diesem neuen Weg ein weiterer neuer Weg gefunden, dann erbt dieser den Tabu-Knoten und darf ihn ebenfalls nicht verwenden. Bei jedem Weitervererben wird ein anfangs initialisierter Zähler dekrementiert. Bei Erreichen von 0 wird dieser Knoten nicht mehr als Tabu-Knoten weitervererbt. Dahinter steckt die Idee, dass sich neue Wege anfangs durch die Tabu-Knoten nicht in Richtung alter Wege *ausdehnen* können, wenn sie aber einen genügend großen Umweg geschafft haben, sollen sie wieder die Möglichkeit haben, den Tabu-Knoten zu verwenden. Dann allerdings in einer anderen Konstellation als bei einem der alten Wege.

Diese sehr eingeschränkte Verwendung von Tabu-Kriterien resultiert auch aus der Beobachtung, dass die Hauptintention bei der Suche nach pareto-optimalen Wegen nicht das Finden einer Superlösung ist. Wichtiger ist das Finden einer möglichst großen Anzahl pareto-optimaler Wege. Beide Hauptmerkmale für die Tabu-Suche, die Definition der Nachbarschaft und der Tabu-Kriterien ist auf das schnelle Auffinden neuer Wege ausgerichtet.

In vielversprechenden Umgebungen von bereits existierenden pareto-optimalen Wegen wird intensiv gesucht. Der Algorithmus arbeitet in die Breite, wenn diese Umgebung durchsucht ist.

3.7 Zeit und Speicheraufwand

Der Zeitaufwand für den Algorithmus hängt in erster Linie von der Struktur des Graphen ab. Haben die Knoten im Durchschnitt eine hohe Anzahl von Kanten, dann sind wesentlich mehr Wege möglich. Im folgenden Abschnitt wird der Speicheraufwand für einen Weg ermittelt. In den darauf folgenden Abschnitten der Zeitaufwand der einzelnen Phasen, wobei folgende Abkürzungen verwendet werden:

V Anzahl der Knoten im Graph (Vertex)

E Anzahl der Kanten im Graph (Edge)

W (auch W_{XY}) Anzahl möglicher Wege von Knoten X zu Knoten Y

P Durchschnittliche Größe der Pareto-Menge

k durchschnittliche Anzahl von Kanten eines Knotens (E/V)

l durchschnittliche Anzahl von Kanten in einem Weg (Weglänge)

b Anzahl der Bewertungen (Kriterien) pro Kante

t Suchtiefe für Nachbarschaftswege

3.7.1 Speicheraufwand

Der Speicheraufwand für einen einzelnen Weg wird durch folgende Komponenten bestimmt.
Auszug aus dem file pareto.hpp (A.4.1):

```
private:
 evalVector evalVec;
 list <edge> edgePath;
 EvalMap distance2Target;
 map <int,int> tabuNodeMap;
 int nSearchUsed;
 INSTYPE iType;
```

Erläuterung zu den einzelnen Objekten:

evalVector evalVec ist die Klasse zum Verwalten der Bewertungskriterien. Die Größe dieser Klasse ist gegeben durch die Anzahl der Bewertungskriterien und des zugrundeliegenden Typs. Im Allgemeinen ist das eine Gleitkommazahl, wobei das Programm aber auch mit dem Bewertungstyp Integer generiert werden kann.

list $<$ *edge* $>$ **edgePath** ist die Liste von Kanten, die den Weg bilden. Die Größe eines Eintrags wird in [6] beschrieben. Sie bewegt sich in der Größenordnung von 8 Bytes (4 Byte pro Knoten).

EvalMap distance2Target ist die map (assoziatives Array) zum Verwalten der Distanzen von einem Knoten zum Zielknoten. Diese map wird beim Suchen neuer Wege über Nachbarn gebraucht. Immer wenn eruiert werden soll, ob der neue Knoten Bestandteil des alten Weges ist, dann wird der Eintrag des entsprechenden Knotens in der map gesucht. Ist er vorhanden, dann liegt er auf dem alten Weg, sonst nicht. Zur eigentlichen Größe der map kommen noch $l + 1$ Knoten als Einträge in diese map.

map $< int, int >$ **tabuNodeMap** ist die map zum Verwalten der Tabu-Knoten.

Die Größe von map und list sind in diversen Beschreibungen zur STL (Standard Template Library) nachzulesen. Zusammen ergibt sich pro Weg ein Speicheraufkommen von jeweils $O(l)$ für die Liste der Kanten des Weges, die Knoten der distance map und für die Tabu-Knoten.

Die Größe der restlichen 2 Parameter (*nSearchUsed* und *iType*) ist im sizeof(int) Bereich und somit konstant. Sie beeinflussen nur marginal die Gesamtspeichergröße eines Weges.

3.7.2 Zeitaufwand Phase 1: Komplette Suche

Diese Phase kann relativ einfach beurteilt werden. Jeder Weg bzw. jede Abzweigung wird verfolgt. Ist der Zielknoten erreicht, werden die bisher aufsummierten Bewertungen der einzelnen Kanten an die Pareto-Mengenverwaltung übergeben, die dann feststellt, ob die Bewertung des neuen Weges paretooptimal ist.

Wenn im Laufe des Zusammensetzens eines neuen Weges dessen Bewertung eine bestimmte Schranke überschreitet, bricht der Algorithmus ab. Da es keine Kanten mit negativen Bewertungen gibt, hat ein Weiterverfolgen dieses Weges keinen Sinn.

Ausserdem wird festgestellt, ob durch Aufnahme der nächsten Kante in den Weg ein Kreis gebildet wird. In diesem Fall wird ebenfalls abgebrochen, weil der Weg ohne Kreis sicher eine dominante Bewertung zum Weg mit Kreis hat. Der Gesamtaufwand setzt sich also folgendermaßen zusammen:

$$\underbrace{O(W * l)}_{Kreistest} + \underbrace{O(W * P * b)}_{Pareto-Test} = O(W * (l + P * b))$$

3.7.3 Zeitaufwand Phase 2: Dijkstra-Wege

Der Gesamtaufwand für die Bestimmung der kürzesten Wege für alle Kriterien beträgt: $b * O(E + V * \log(V))$ (siehe [8])

Wenn man zusätzlich den Hinweis aus Kapitel 3.2 letzter Absatz berücksichtigt, dann erhöht sich der Aufwand um den Faktor l (durchschnittliche Weglänge). Zur Erinnerung: jede Kante im Weg wird sukzessive durch eine längere ersetzt und der Dijkstra-Algorithmus wird jedesmal neu durchlaufen.

Partiell kann man diesen Mehraufwand reduzieren, indem man z.B. direkt den Dijkstra-Algorithmus modifiziert und sich merkt, welche Kanten eine

Nichteindeutigkeit verursacht haben. Nur bei diesen Kanten ist es sinnvoll in erwähnter Weise einzugreifen.

Eine weitere Möglichkeit besteht darin, sich die Tatsache zu merken, dass solche Kanten aufgetreten sind und in der Folge den kompletten Mechanismus anzuwenden. Ist diese spezielle Situation nicht aufgetreten, dann kann auf die Anwendung verzichtet werden.

3.7.4 Zeitaufwand Phase 3: Nachbarwege

Zur Bestimmung eines Nachbarweges wird der gesamte alte Weg durchlaufen, ergibt $O(l)$. Bei jedem Knoten werden alle adjazenten Kanten betrachtet und zwar bis zur Tiefe t. Ergibt somit $O(k^t)$. Für jeden dieser Versuche muss festgestellt werden, ob sich ein inzidenter Knoten im alten Weg befindet, also $O(\log(l))$. Wenn ein neuer Weg gefunden wurde, dann muss festgestellt werden, ob er pareto-optimal ist, ergibt $O(P * b)$. Die Tatsache, dass bisher gefundene Wege sich im nachhinein als nicht pareto-optimal herausstellen können und entfernt werden müssen, ist dabei schon berücksichtigt. Bis hierher habe ich den Aufwand zur Berechnung der Nachbarwege für einen alten Weg erfasst. Dieser Aufwand ist aber letztendlich für jeden Weg in der Pareto-Menge erforderlich. Ergibt zusätzlich den Faktor $O(P)$.

Zusammengefasst ergibt sich: $O(l * k^t * log(l) * b * P^2)$. Für spezielle Graphen, wie in Abbildung 1.2 gezeigt, bedeutet der Faktor P^2 natürlich das Ende jeder Spekulation, ob der Algorithmus bei großen Graphen in vertretbarer Zeit endet. Wobei man beachten muss, dass sich ein "P" aus der Anzahl der Wege ergibt und das zweite "P" aus der Notwendigkeit, den neuen Weg in Bezug zur bisher gefundenen Menge an pareto-optimalen Wegen zu setzen. D.h., die Pareto-Menge muss durchlaufen und jeder Eintrag muss mit dem neuen Weg verglichen werden.

Diesen zweiten Faktor könnte man vermutlich reduzieren, indem *geschickt* in der vorhandenen Menge gesucht wird. Eine diesbezügliche Untersuchung könnte Bestandteil einer weiterführenden Analyse sein, die allerdings in dieser Arbeit nicht gemacht wurde.

3.7.5 Zeitaufwand Phase 4: Diversifizierung

Je mehr Wege über die bisherigen Phasen gefunden wurden, desto weniger unbenutzte Kanten gibt es im Allgemeinen. Das ist ein Hinweis, dass für eine gesamte Abschätzung des Algorithmus, eine amortisierte Analyse die Laufzeit in positiverem Licht erscheinen lassen würde. Da dies aber ebenfalls weit über den Rahmen dieser Arbeit hinausgeht, muss hier darauf verzichtet werden. Eine kurze Beschreibung zu "amortisierte Analyse" kann man auf [9] nachlesen. Ohne amortisierte Analyse muss ich davon ausgehen, dass $O(E)$ unbenutzte Kanten nach den ersten Phasen existieren. Für jede dieser Kanten muss der Dijkstra -Algorithmus zweimal aufgerufen werden, $= 2 * (O(E + V * \log(V)))$. Für jeden neuen Weg muss danach festgestellt werden, ob er pareto-optimal ist, ergibt $O(P * b)$. Alle Teilabschätzungen ergeben zusammen:

$$O((E + V * \log(V)) * O(E) * O(P * b)$$

Wiederum muss bemerkt werden, dass $O(P)$ in besonderen Konstellationen alle anderen Faktoren dominiert. Im Normalfall dominiert der Faktor $O(E^2)$. Im Programm ist als *Notbremse* eine Aufrufoption eingebaut (siehe A.2), die den Algorithmus zwingt, nach einer bestimmten Zeit abzubrechen.

3.7.6 Zeitaufwand Phase 5: Konvexe Hülle

Für die Abschätzung des Quickhull-Algorithmus verweise ich auf [1].

Kapitel 4

Externe Software

In diesem Kapitel werden zwei Softwareteile beschrieben, die aus externen Quellen stammen. Im ersten Teil Quickhull (qhull), eine Software, die aus n Vektoren des d-dimensionalen Raumes die umschließende Hülle berechnet und im zweiten Teil die Graph Template Library, die die Verwaltung des Graphen übernimmt.

4.1 qhull

Basis von qhull ist der Quickhull Algorithmus. Dieser wird in [1] beschrieben. Die hier verwendete Software kann von [7] bezogen werden.

Nachdem festgestellt wurde, dass bei höheren Dimensionen (ca. 5) die Laufzeit mit der Anzahl der Vektoren erheblich steigt, wurde die Verwendung von qhull optional in das Programm eingebaut. Eine Abschätzung für die Laufzeit kann man in [1] nachlesen. Diesbezügliche Dokumente sind aber auch auf [7] verfügbar.

Die Software qhull kann aber nicht nur zur Bestimmung einer konvexen Hülle verwendet werden, sondern deckt viele graphenorientierte Bereiche ab. Dazu direkt von qhull.org zitiert:

> Qhull computes the convex hull, Delaunay triangulation, Voronoi diagram, halfspace intersection about a point, furthest-site Delaunay triangulation, and furthest-site Voronoi diagram. The source code runs in 2-d, 3-d, 4-d, and higher dimensions. Qhull implements the Quickhull algorithm for computing the convex hull. It handles roundoff errors from floating point arithmetic. It computes volumes, surface areas, and approximations to the convex hull.

4.2 GTL

Für die Verwaltung des Graphen wird in dem realisierten C++Programm die "Graph Template Library" (GTL) verwendet. Eine genaue Beschreibung findet sich in [6], wo auch die Software bezogen werden kann. Die GTL wurde aus mehreren Gründen gewählt:

- Erprobte Software

Die GTL wird in mehreren Projekten verwendet. Dadurch ist sie einerseits gut getestet und andererseits sind viele Anforderungen erfüllt, die in typisch graphenorientierten Problemen auftauchen.

- Implementierte Algorithmen

  Meist sind in solchen Software-Paketen, wie auch in der GTL, typische Algorithmen wie Dijkstra bereits implementiert. Eigenversuche in diese Richtung enden oft mit viel Zeitaufwand bei einer schlechteren Performance. Letztendlich ist eine zukünftige Erweiterung der Software mit bereits vorhandenen Algorithmen ohne große Probleme möglich. Schon implementierte Algorithmen in der GTL sind neben anderen: Bellman/Ford, BreathFirstSearch, DepthFirstSearch, Biconnectivity, Max/Minflow, etc. Weitere Details können unter [6] nachgelesen werden.

- Standardisierte Eingabe

  Bei der GTL ist es möglich, die Beschreibung eines Graphen in GML-Format aus einer Datei zu lesen. Mithilfe dieses Formats ist es möglich, auch andere Softwarepakete auf den Graphen aufzusetzen, bzw. die Beschreibung eines Graphen von anderen Paketen zu übernehmen. Bei einem proprietären File Format ist das wesentlich schwieriger und benötigt i.A. einen Konverter.

- Kurze Einarbeitungszeit

  Die GTL ist *relativ* einfach aufgebaut. Dadurch muss kaum Zeit investiert werden, um in die Source/den Ablauf einzugreifen zu können. Bei meinem Program ist das z.B. beim Scannen und Parsen der GML-Eingabe Datei passiert. Der Scanner und Parser wurde so umgeschrieben, dass er auch die Bewertungen der Kanten analysiert und über Callbacks an das aufrufende Programm zurückgibt.

4.3 GML File Format

Das GML-Eingabeformat kann an folgender Stelle nachgelesen werden:
`http://infosun.fmi.uni-passau.de/Graphlet/GML/`
Hier ist sehr gut das Format beschrieben, welches als Grundlage für die Speicherung des Problemgraphen dient. Da ich die Eigenschaft der Mehrfachbewertung von Kanten in den Scanner und Parser hinzugefügt habe, möchte ich hier einen kompletten Beispielgraphen zeigen. Die Attribute *sourceNode*, *targetNode*, *nrOfEval* und *evals* mitsamt ihren Werten sind in der ursprünglichen GML-Definition nicht enthalten. Sie sind die Spezifika dieser Art multikriterieller Optimierungsaufgaben.
File example.gml:

```
graph [
  id 0
  sourceNode 0
  targetNode 9
```

```
  directed 1
  nrOfEval 2
  edge [ source 0 target 1 evals 3 14 ]
  edge [ source 0 target 2 evals 5 15 ]
  edge [ source 1 target 3 evals 6 18 ]
  edge [ source 1 target 4 evals 7 12 ]
  edge [ source 1 target 5 evals 8 12 ]
  edge [ source 1 target 6 evals 9 15 ]
  edge [ source 2 target 3 evals 1 19 ]
  edge [ source 2 target 4 evals 6 14 ]
  edge [ source 2 target 5 evals 3 12 ]
  edge [ source 2 target 6 evals 4 13 ]
  edge [ source 3 target 7 evals 7 11 ]
  edge [ source 3 target 8 evals 8 15 ]
  edge [ source 4 target 7 evals 9 17 ]
  edge [ source 4 target 8 evals 5 18 ]
  edge [ source 5 target 7 evals 6 13 ]
  edge [ source 5 target 8 evals 2 14 ]
  edge [ source 6 target 7 evals 1 17 ]
  edge [ source 6 target 8 evals 1 15 ]
  edge [ source 7 target 9 evals 3 12 ]
  edge [ source 8 target 9 evals 4 10 ]
]
```

Erwähnt werden muss, dass ich bei der Definition der Kantenbewertungen (evals) von der Basisphilosophie des GML-Formates abgewichen bin. Dort ist es üblich, pro Attribut (z.B. *nrOfEval*) genau einen Wert zuzulassen, unabhängig davon, ob es für das jeweilige Attribut sinnvoll ist. Ich lasse beim Attribut *evals* beliebig viele Werte zu. Die Anzahl muss im gesamten Graphen übereinstimmen und dem Headereintrag *nrOfEval* entsprechen.

Anmerkung: Dieser Graph wird später verwendet, um den Ablauf des Programmes zu dokumentieren (A.8) und ist deshalb größer ausgefallen als zur Darstellung des Formats nötig wäre.

Kapitel 5

Ergebnisse und Vergleich

Ziel dieser Arbeit ist es festzustellen, ob Tabu-Suche ein geeignetes Mittel ist, um die pareto-optimale Menge von Wegen in einem Graphen mit multikriteriellen Kanten zu bestimmen. In den ersten beiden Abschnitten dieses Kapitels werden die Testläufe des Programms analysiert und beurteilt. Im dritten Abschnitt wird ein Vergleich mit einem Algorithmus gezogen, der in einer früheren Arbeit zu dieser Problematik bereits implementiert worden ist [2]. In dieser Arbeit wurde ein Testgraph verwendet, den ich als Grundlage des Vergleiches herangezogen habe. Die Daten zu diesem Testgraph sind in [2] angeführt.

5.1 Ergebnisse

Tabu-Suche ist ein ausgezeichnetes Mittel, um die pareto-optimalen Wege in einem Graphen festzustellen. Diese Aussage gilt für Graphen, bei denen die optimalen Wege große Bereiche bilden, in denen viele pareto-optimale Wege in einer Nachbarschaftsrelation zueinander sind. Zwei Wege sind im Sinne des Algorithmus in einer Nachbarschaftsrelation, wenn ausgehend von einem Weg, der andere über Nachbarschaftssuche erreichbar ist. Ob ein Weg von einem anderen über Nachbarschaftssuche erreichbar ist, hängt selbstverständlich davon ab, wie groß die maximale Tiefe der Suche gewählt wird. Je größer die Tiefe gewählt werden muss, um zwei Wege in eine Nachbarschaftsrelation zu bringen, desto weiter entfernt voneinander liegen sie. Diese Relation gilt nur in eine Richtung. Wenn ein Weg X_1 von einem Weg X_2 erreichbar ist, dann heißt das nicht automatisch, daß der Weg X_2 vom Weg X_1 aus erreichbar ist.
Zusammenfassend kann man es so sagen:

Der Algorithmus arbeitet bei allen Graphen gut, bei denen die vorgegebene Tiefe nicht groß gewählt werden muss, um große Gebiete in der Nachbarschaft eines Weges zu erreichen.

Wird ein Vertreter dieser Äquivalenzklasse (z.B. durch Diversifizierung 3.4) gefunden, dann findet der Algorithmus alle Wege, die in seiner Nachbarschaftsrelation liegen.

Ist der Graph in viele kleine Bereiche geteilt, dann werden zwar einige Wege über Diversifizierung gefunden, für den Algorithmus insgesamt sind das aber keine guten Voraussetzungen.

Ich werde drei verschiedene Graphen zur Beurteilung des Algorithmus heranziehen. Der erste hat bereits im Kapitel über die Komplexität (1.3) der Pareto-Menge traurige Berühmtheit erlangt. Er zeigt, dass es nicht immer sinnvoll ist Tabu-Suche einzusetzen, ja manchmal sogar die komplette Suche, das komplette Traversieren des Graphen (3.1) besser ist.

Der zweite Graph ist nicht ein einzelner Graph sondern eine Klasse von Graphen, die durch ein eigens dafür geschriebenes Programm generiert wurden. Mit diesem Programm habe ich 100 Graphen generiert und sie vom Algorithmus bearbeiten lassen. Diese Vorgangsweise habe ich deshalb gewählt, weil es zu aufwendig war, einen Graphen manuell zu erstellen, der möglichst kleine zusammenhängende Bereiche von pareto-optimalen Wegen enthält.

Der dritte Graph ist der Testgraph aus [2]. Er zeigt, dass Tabu-Suche in praktischen Problemen ausgezeichnet funktioniert.

5.1.1 Exponentielle Anzahl von Wegen

Basis dieses Tests ist der Graph aus Abbildung 1.2. Nochmal zur Erinnerung: Für je zwei Knoten (4 Kanten), die am Ende des Graphen angehängt werden, verdoppelt sich die Anzahl der pareto-optimalen Wege. Ich habe mich beim Test auf die Länge 10 beschränkt. Lässt man den Algorithmus mit der Option -c2 durchlaufen, also "Komplette Suche", dann wird er in $\leq$ 1 Sekunde fertig und gibt alle 1024 pareto-optimalen Wege aus. Lässt man den Algorithmus mit der Option -l2 laufen, also Nachbarschaftssuche bis level 2, dann braucht er für den Durchlauf ca. 9 Sekunden, also wesentlich länger als mit kompletter Suche. Die Daten sind in Tabelle 5.1 zu sehen.

N-Tiefe	Zeitaufwand	#gleiche Wege	#Pareto Vergleiche
kompl.Suche	$\leq$ 1	0	523.776
2	$\leq$ 9	5487	ca. 4.000.000
3	$\leq$ 11	6897	ca. 5.300.000

Tabelle 5.1: Aufwandsvergleich bei exponentieller Anzahl

#gleiche Wege sagt aus, wie oft ein Weg gefunden wurde, der bereits in der Pareto-Menge enthalten war.
#Pareto Vergleiche ist die Gesamtanzahl von Vergleichen von potentiell neuen Wegen mit Wegen aus der Pareto-Menge.

Eine große Anzahl "gleicher Wege" ist ein Indiz, dass die Tabu-Kriterien nicht wirksam werden. Das darf hier allerdings nicht verwundern. Jeder der beiden ursprünglich gefundenen Dijkstra-Wege ist alleine in der Lage alle pareto-optimalen Wege zu finden. Nachdem die Tabu-Kriterien in diesem Algorithmus nur lokal eingesetzt werden, greifen sie natürlich nicht, wenn ein neuer Weg von zwei verschiedenen Startpunkten aus gefunden wird.

Die Ablaufzeit für die Weitervererbung eines Tabu-Knotens ist begrenzt. Wenn also, wie in diesem Testgraph, die Wege sehr dicht liegen, dann passiert es, dass man sich immer noch im Bereich des Ausgangsweges befindet, obwohl ein *gefährlicher* Knoten schon nicht mehr tabu ist. Dadurch werden immer

wieder gleiche Wege gefunden. Sie sind zwar pareto-optimal, aber bereits in der Pareto Menge vorhanden.

Resumee: Für sehr dichte Graphen müssen komplexere Tabu-Kriterien entwickelt werden.

5.1.2 Künstliche Testgraphen

Für diese Art Test wird ein Graph generiert, der folgende Eigenschaften besitzt:

1. Der Graph soll so klein sein, dass er in akzeptabler Zeit komplett durchsucht werden kann. Eine Beurteilung der Ergebnisse für einen normalen Durchlauf ist sonst fast unmöglich.

2. Der Graph soll so groß sein, dass der Aufwand für einen normalen Durchlauf klein ist, im Verhältnis zur kompletten Suche.

3. Es soll möglichst wenig zusammenhängende Bereiche geben. Deshalb ist die Option -e5 (Anzahl von Kanten/Knoten) in Relation zur Anzahl der Knoten pro Schicht (10) relativ klein gewählt.

4. Mit der Anzahl der Testgraphen (100) soll erreicht werden, dass nicht zufällig ein Graph mit großen zusammenhängenden Bereichen ausgewählt wird.

Um den geforderten Eigenschaften möglichst nahe zu kommen, wird ein Graph generiert mit einem Anfangs- und Endknoten und mehreren Schichten von Knoten dazwischen. Die Anzahl der Schichten, die Anzahl der Knoten, die durchschnittliche Anzahl von Kanten pro Knoten und die Anzahl der Bewertungen pro Kante können beim Aufruf des Generators gewählt werden. Die Knoten werden einfach von 0 (Anfangsknoten) bis *#Knoten+1* (Endknoten) durchnummeriert. Danach werden die Kanten generiert. Bewertung, Zielknoten und Level des Zielknotens werden gleichverteilt nach folgendem Schema:

Zielknotenschicht	Bewertung Min	Bewertung Max	Wahrscheinlichkeit
0	1	10	10
1	**5**	**50**	**50**
2	40	100	30
3	90	200	10

Tabelle 5.2: Kantenbewertung und Zielknotenverteilung

Nehmen wir als Beispiel Zeile 2. Sie sorgt dafür, dass ausgehend vom momentan bearbeiteten Knoten, 50 Prozent aller ausgehenden Kanten in der nächsten Schicht (1) landen und dass dessen Bewertung gleichverteilt zwischen 5 und 50 gewählt wird. Die Anzahl der Kanten, die von einem Knoten ausgehen, werden gleichverteilt zwischen $e-2$ und $e+2$, wobei e die Anzahl der durchschnittlich geforderten Kanten laut Option ist.

Für den eigentlichen Test wurden 100 Graphen generiert mit folgenden Optionen:

gengraph -n70 -e5 -l7 -w5

Das ergibt 70 Knoten in 7 Schichten zu je 10 Knoten, wobei jeder Knoten im Schnitt 5 Kanten hat und jede dieser Kanten 5 Bewertungen.

Von diesen Graphen habe ich jeweils 3 mit den schlechtesten Ergebnissen genommen. Das Ergebnis wurde mit dem Output der kompletten Suche verglichen und ist in Tabelle 5.3 zu sehen. Anhand der ersten Zeile ist in der Spalte Zeit gut zu erkennen, wie unterschiedlich die Struktur der erhaltenen Graphen ist. Die komplette Suche hat zwischen 2 und 324 Sekunden gedauert.

Jede weitere Zeile der Tabelle ist das Ergebnis des Algorithmus in Relation zum jeweiligen exakten Ergebnis dieses bestimmten Graphen.

N-Tiefe	Zeit	#pareto	#dominierte	% von allen
kompl.Suche	2-324	alle	0	100
2	2	64	39	13
2	1	92	33	27
2	1	63	23	28
3	2	75	3	68
3	14	73	0	72
3	3	45	1	76
4	3	92	3	84
4	9	99	2	88
4	2	123	1	89

Tabelle 5.3: Ergebnis künstliche Graphen

"#pareto" Anzahl der gefundenen pareto-optimalen Wege.
"#dominierte" Anzahl vermeintlich pareto-optimaler Wege.
"% von allen" daraus resultierender prozentueller Anteil in Relation zu allen, im jeweiligen Graphen enthaltenen pareto-optimalen Wegen.

Negativ: Nachbarschaftssuche mit Tiefe 2 reicht bei manchen Graphen nicht aus, um gute Ergebnisse zu erhalten. Bei ca. 20% aller Graphen wurde auf diesem level nur die Hälfte der pareto-optimalen Wege gefunden. Nur ein einziges Beispiel war dabei, wo der Algorithmus mehr als 90% der Wege gefunden hat.

Bei level 4 sind die Ergebnisse zwar zufriedenstellend, allerdings beginnen bei level 4 i.A. die ersten Laufzeitprobleme. In den hier verwendeten Graphen gibt es noch kein Problem, dafür sind die Graphen zu klein. Um zu sehen, wie schnell die Laufzeit steigen kann, habe ich einen ähnlichen Graphen mit Option -e10 generiert. Dadurch steigt die Laufzeit bei level 2 im Schnitt auf 151 Sekunden, bei level 4 auf ca. 1000! Ich habe keine komplette Testreihe unter diesen Bedingungen gemacht, weil eine komplette Suche unter den gegebenen Systemvoraussetzungen nicht mehr möglich war. Ergebnisse sind ohne Wissen über die exakte Anzahl und Qualität der pareto-optimalen Wege kaum zu beurteilen.

Positiv: Bei level 3 sind die 3 schlechtesten Resultate mit ca. 70% der pareto-optimalen Wege immer noch nicht akzeptabel, aber es ist z.B. kein einziger Graph dabei, bei dem weniger als die Hälfte der Wege gefunden wurde. Bei immerhin 75% wurden mehr als 90% der Wege gefunden.

Das Ergebnis zeigt, dass es einen gewissen Prozentsatz von künstlich generierten Graphen gibt, bei denen der Algorithmus nicht zufriedenstellend funktioniert. Dieser Prozentsatz liegt bei ca. 20% (unter den Voraussetzungen aus 5.2). Bei diesen Graphen war die Erfolgsrate bei level 2 unter 50%. Es wurde also nur die Hälfte der vorhandenen pareto-optimalen Wege gefunden.

5.2 Vergleich zu Ergebnissen aus [2]

In früheren Diplomarbeiten am Institut für Wirtschaftsmathematik wurde das Problem der multikriteriellen Routenplanung bereits bearbeitet. In einer dieser Arbeiten ([2]) wurde mithilfe eines Evolutionären Algorithmus die Pareto-Menge bestimmt. Die Ergebnisse dieses Algorithmus und ein Vergleich mit dem hier vorgestellten ist in Tabelle 5.4 dargestellt.

Algorithmus	Zeit(Sek.)	#Wege	Qhull	dominierte Wege
kompl.Suche	4800	188	33	0
Evol. Algo	10	83	3	2
L2	1	129	13	1
L3	2	184	32	0
L4	3	188	33	0

Tabelle 5.4: Ergebnisvergleich Evolutionärer Algorithmus - Tabu-Suche

"Qhull" ist die Anzahl der Wege, die bei der Bildung der konvexen Hülle nicht berücksichtigt wurden und damit aus der Menge der guten pareto-optimalen Wege ausgeschieden werden. Die relativ kleine Anzahl von non-supported Wegen im Evolutionären Algorithmus ist darauf zurückzuführen, dass bei der praktischen Umsetzung des Algorithmus schon ein Teil der, von hybriden Kombinationen dominierten Wege, herausgefiltert wurde.

Beim Vergleich der beiden Algorithmen fällt auf, dass in der Tabu-Suche schon mit Nachbarschaftssuche der Tiefe 2 wesentlich mehr pareto-optimale Wege gefunden werden als mit dem Evolutionären Algorithmus. Weiters ist erstaunlich, dass schon mit Tiefe 3 fast alle Wege gefunden werden. Wie wir im letzten Abschnitt (5.1.2) gesehen haben, muss das nicht unbedingt bei jeder Art von Graph der Fall sein.

5.3 Zusammenfassung

Der Algorithmus Tabu-Suche ist zum Auffinden möglichst vieler pareto-optimaler Wege in einem Graphen mit mehrfach bewerteten Kanten gut geeignet. Bei Graphen, deren pareto-optimale Wege im Sinne des Algorithmus benachbart sind, ist die Nachbarschaftssuche (3.3) erfolgreich. Graphen, bei denen diese

Wege *zu weit* auseinander liegen, sind oft durch die Diversifizierung aus Kapitel 3.4 zu analysieren. Und es gibt Graphen (5.1.2), bei denen diese Form von Tabu-Suche nicht geeignet ist: "No free lunch"

Letztendlich bleibt einem Anwender nicht erspart, die jeweilige Struktur seines modellierten Problems diesbezüglich zu analysieren und sich dementsprechend für/gegen den Algorithmus zu entscheiden.

5.4 Ausblick und Weiterentwicklung

In diesem Kapitel möchte ich ein paar Anregungen für eine Weiterentwicklung der Software machen.

- Die Berechnung der konvexen Hülle der Bewertungsvektoren mit quickhull stützt sich auf einen inkrementellen Algorithmus. Es wäre daher sinnvoll, permanent die Berechnung der Hülle mitlaufen zu lassen und so non-supported Lösungen von vornherein nicht in die Pareto-Menge aufzunehmen. Damit wäre diese Menge kleiner und der Vergleich neuer Lösungen mit der Menge schneller durchzuführen.

- Derzeit wird die Berechnung der konvexen Hülle der Bewertungsvektoren ausschließlich dazu verwendet um non-supported Lösungen zu eliminieren. Diese Lösungen werden von Algorithmen, die von vornherein mit einem Gewichtsvektor arbeiten, oft gar nicht erkannt. Sie stellen aber unter Umständen eine interessante Alternative zur mathematisch besten Lösung dar. Sie sollten immer dann separat ausgewiesen werden, wenn sie mit ihrer gewichteten Zielfunktion nicht allzusehr von der besten Lösung abweichen.

- Nachdem ein neuer Weg gefunden wurde, wird die Pareto-Menge durchlaufen, um festzustellen, ob der neue Weg pareto-optimal ist. Es könnte sich als sinnvoll erweisen, die Elemente der Pareto-Menge nach ihrer Erfolgsrate beim Erkennen von nicht pareto-optimalen Wegen zu ordnen. Mit anderen Worten, nicht alle pareto-optimalen Wege sind *gleich gut*. Als Beispiel mögen die beiden folgenden Vektoren dienen:

$$\begin{pmatrix} 99 \\ 100 \\ 100 \end{pmatrix} <> \begin{pmatrix} 100 \\ 9 \\ 9 \end{pmatrix}$$

  Beide sind pareto-optimal aber wie leicht zu sehen ist, traut man dem zweiten Vektor wesentlich eher zu, andere Vektoren als nicht pareto-optimal zu erkennen. Sind diese guten Vektoren vorgereiht, dann spart man sich den Vergleich eines neuen, dominierten Vektors mit den nicht so guten. Vielleicht reicht es auch, wenn man den Ausgangsweg an den Anfang der Liste der pareto-optimalen Wege stellt. Da es doch sehr wahrscheinlich ist, dass Nachbarwege vor allem mit ihrem Ausgangsweg um Optimalität konkurrieren, könnte diese kleine Umstellung schon Erfolge zeitigen.

Ein weitere Möglichkeit den Aufwand zum Erkennen der Pareto-Optimalität zu senken, ist der Einsatz von "quad-trees". Da ich erst kurz vor Abschluss dieser Arbeit von der Existenz dieser binary search tree ähnlichen Datenstruktur erfahren habe, konnte ich sie weder in der Software noch in dieser Arbeit berücksichtigen.

- Für den praktischen Einsatz vorteilhaft, wäre die dynamische Anpassung der Suchtiefe in der Nachbarschaftssuche. Angenommen, der Algorithmus hat eine Vorgabe von Suchtiefe 2 und einer Maximalzeit von 10 Sekunden. Wenn er nach einer Sekunde fertig ist, dann hört er mit der weiteren Suche auf. Sinnvoll wäre in so einem Fall, die Tiefe dynamisch zu erhöhen. In diesem Fall könnte man auch berücksichtigen, dass bis zur Tiefe 2 bereits gesucht wurde und neue Wege werden erst ab Tiefe 3 zu neuen Ergebnissen führen können.

Literaturverzeichnis

[1] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[2] Totz Bettina. Multikriterielle Entscheidungsfindung bei der Routenplanung. Diplomarbeit am Institut für Wirtschaftsmathematik, 2004.

[3] Gendreau Michel. *An Introduction To Tabu Search*, chapter 2, pages 37–54. Kluwer Academic Publishers, 2003.

[4] Gendreau Michel. Trajectory-based search methods. CP-AI-OR'05, Master Class, Prag, May 2005.

[5] Raidl Günther. Heuristische Optimierungsverfahren. Folien zur Vorlesung. www.ads.tuwien.ac.at, 2004.

[6] M. Raitner und C. Bachmaier M. Forster, A. Pick. Graph Template Library. http://infosun.fmi.uni-passau.de/GTL/index.html.

[7] Quickhull homepage. http://qhull.org/.

[8] Wikipedia. Algorithmus von Dijkstra. [Inhalt vom 07.03.2006]. http://de.wikipedia.org/wiki/Dijkstra-Algorithmus.

[9] Wikipedia. Amortisierte Laufzeitanalyse. [Inhalt vom 07.03.2006]. http://de.wikipedia.org/wiki/Amortisierte_Laufzeitanalyse.

[10] Wikipedia. Graphentheorie. [Inhalt vom 07.03.2006]. http://de.wikipedia.org/wiki/Graphentheorie.

[11] Wikipedia. No Free Lunch. [Inhalt vom 11.03.2006]. http://de.wikipedia.org/wiki/NFL-Theorem.

[12] Wikipedia. Np-schwere. [Inhalt vom 27.03.2006]. http://de.wikipedia.org/wiki/NP-hart.

[13] Wikipedia. Pareto Optimierung. [Inhalt vom 07.03.2006]. http://de.wikipedia.org/wiki/Pareto-Optimierung.

Anhang A

Systemumgebung, Optionen und Programmbeschreibung

In diesem Kapitel wird beschrieben, auf welchem System das Programm entwickelt wurde und wie es verwendet werden kann. Weiters wird eine Beschreibung der wichtigsten Klassen und deren Methoden gemacht. Bei Inline Funktionen wie getVal habe ich den Source Code entfernt und Operatoren wie Gleich (==) oder Ungleich (!=) habe ich nicht extra beschrieben. Den entsprechenden Sourcecode habe ich durch ... angedeutet, bzw. überhaupt weggelassen.

A.1 Systemumgebung

Das Programm wurde entwickelt auf einem PC mit folgenden Kenndaten:
Betriebssystem: Microsoft Windows XP Home Edition ©
Prozessor: x86 mit 3 Gigahertz Taktfrequenz und 512 Megabyte Hauptspeicher
Entwicklungsumgebung: Microsoft .NET Framework 1.1 Version 1.1.4322
Microsoft Visual C++.NET 69536-270-0000007-18565

A.2 Programmoptionen

Das Programm kann mit folgenden Optionen aufgerufen werden:

```
programname -h
 -d file.mdb database-file with problem description or
 -g file.gml gml-file with problem description
 -i[0-9]  ignore i'th evaluation, can be multiple used
 -f[0-9] in seconds until finish
 -l[0-9] level, how far should be searched
 -s with or without statistics
 -c[0-9]* complete search, interrupt after x seconds
 -w string weightstring example: "0.3 0.3 0.3 0.1"
 -wi ignore weights e.g. from database
 -z allow zero in edge evaluation
 -q call q-hull algorithm
```

```
-h show this message
```

Option g/d Option g bzw d sagt, in welcher Form die Probleminstanz vorliegt. Das kann einerseits im GML-Format sein (siehe 4.2) oder als Tabellen in einer Datenbank ([2]).

Option i Option i für ignore wird verwendet, um die Dimension des Bewertungsvektors zu verringern. Durch die Verringerung wird i.A. die Anzahl der möglichen Lösungen erheblich verkleinert. Ausserdem ist es dadurch möglich, bei Durchläufen mit unterschiedlich vielen Bewertungen, die gleiche Probleminstanz zu verwenden.

Option f Option f zwingt das Programm, den Algorithmus nach einer bestimmten Zeit abzubrechen. Das ist insbesondere dann notwendig, wenn zu viele Lösungen vorliegen (1.3) und der Algorithmus immer neue Wege in die Pareto-Menge aufnimmt.

Option l Option l gibt die maximale Tiefe der Nachbarschaftssuche vor.

Option s Option s wird verwendet, um Statistiken zu sammeln. Das geschieht in erster Linie, um den Algorithmus weiterzuentwickeln, bzw. ihn beurteilen zu können (3.7).

Option c Option c gibt vor, wie lange in Phase 1 (komplette Suche) gesucht werden soll. Auch diese Option wird in erster Linie zur Abschätzung des Algorithmus verwendet.

Option w Option w wird mit einem Gewichtsvektor verwendet. Dieser überschreibt einen in der Datenbank vorliegenden Gewichtsvektor. Diese Option kann allerdings auch in der Form -wi aufgerufen werden und sorgt dann dafür, dass der Gewichtsvektor ignoriert wird. In diesem Fall wird die gesamte Pareto-Menge als Ergebnis ausgegeben. Bei einem vorliegenden Gewichtsvektor wird nur der damit festgelegte beste Weg ausgegeben.

Option z Option z erlaubt es dem Programm, die Bewertung eines Kriteriums mit 0 als solche zu akzeptieren. Ohne Option z wird die Zahl 0 in Bewertungen als "sehr großer Wert" interpretiert.

Option q Option q sorgt dafür, dass am Ende des Durchlaufes, alle schlechten Lösungen, d.h. Lösungen, die sich innerhalb der konvexen Hülle aller Lösungen befinden, mittels der externen Software qhull eliminiert werden.

Option a Option a wird bei Aufruf des Programmes mit -h nicht ausgegeben, weil sie normalerweise von einem Anwender nicht verwendet wird. Sie kann mit den Ziffern 1-3 belegt werden und produziert stufenweise steigerbar einen trace des Programmablaufes.

A.3 Klasse EvalVector

Die Klasse EvalVector verwaltet die Bewertungskriterien.

```
class EvalVector {
public:
  ~EvalVector();
  EvalVector();
  EvalVector( int val );
  EvalVector (const EvalVector& w);
  inline void setVal( int pos, EVAL_TYPE value )
  ...
  inline EVAL_TYPE getVal( int pos ) const
  ...
  PARETO_RESULT relatedTo (const EvalVector* w) const;
  PARETO_RESULT relatedTo (const EvalVector* w, int pos) const;
  bool operator < (const EvalVector& w) const;
  bool operator != (const EvalVector& w) const;
  inline bool operator == (const EvalVector& w) const
  ...
  EvalVector& operator += (const EvalVector& w);
  EvalVector& operator -= (const EvalVector& w);
  EvalVector& operator = (const EvalVector& w);
  DEVAL_TYPE scalarProdukt(const EvalVector& v);
  bool noDiffFromZero();
  void adaptMax(const EvalVector* v);
  void adaptMin(const EvalVector* v);
  bool setIfBetter(const EvalVector* v, const EvalVector* goal,
                   const WeightVector& w);
private:
  EVAL_TYPE* val;
};
```

A.3.1 Daten EvalVector

EVAL_TYPE* val ist ein Zeiger auf das array der Bewertungskriterien. Nachdem erst nach dem Laden des Problemgraph bzw. nach dem Analysieren der Kommandozeile bekannt ist, wie groß der Bewertungsvektor ist, wird der benötigte Speicher erst zu diesem Zeitpunkt allokiert.

A.3.2 Methoden EvalVector

relatedTo Die Methode relatedTo vergleicht zwei Objekte vom Typ EvalVector und liefert als Ergebnis ein enum vom Typ PARETO_RESULT. Dieses enum zeigt alle möglichen Ergebnisse bei einem Vergleich zwischen zwei Bewertungsvektoren. Die möglichen Ergebnisse dieses Vergleiches sind hier aufgelistet:

```
typedef enum PARETO_RESULT { XincompY, XdomY, YdomX, XequalY };
```

XincompY ergibt sich, wenn beide Vektoren unvergleichbar, also pareto-optimal sind. *XdomY* bedeutet X dominiert Y, *YincompX* das Gegenteil. Mit *XequalY* ist die Aufzählung komplett.

noDiffFromZero Die Methode noDiffFromZero liefert true, falls sich ein Bewertungsvektor nur minimal vom Nullvektor unterscheidet. Damit werden Rundungsfehler kompensiert.

Vor einem praktischen Einsatz sollte die Problematik kummulierender Rundungsfehler intensiver untersucht werden.

adaptMax,adaptMin Die Methoden adaptMax und adaptMin sammeln die jeweils besten/schlechtesten Ergebnisse für jede einzelne Dimension. Das Ergebnis ist einerseits der Zielvektor für die Selektion des besten Weges (1.4) und andererseits der schlechteste Vektor, der beim Eliminieren dominierter Wege über die konvexe Hülle (3.5) eine Rolle spielt.

setIfBetter Die Methode setIfBetter ist eine Hilfsfunktion bei der Bestimmung des besten Weges. Sie stellt von zwei Wegen den besseren im Sinne des Gewichtsvektors fest.

A.4 Klasse Pareto

Die Klasse Pareto verwaltet einen pareto-optimalen Weg.

```
class Pareto {
public:
  ~Pareto();
  Pareto( MotsGraph& g, list<edge> edgeList, Pareto& oldP,
          EvalVector w, edge tabuEdge, int tabuNr );
  Pareto( MotsGraph& g, list<edge> edgeList, INSTYPE type );
  void insertNodes( dijkstra::shortest_path_edge_iterator it,
                    dijkstra::shortest_path_edge_iterator ite,
                    MotsGraph& g );
  PARETO_RESULT relatedTo( const Pareto* p ) const;
  PARETO_RESULT relatedTo( const EvalVector* w ) const;
  bool isEvalCorrect( MotsGraph &g ) const;
  EvalVector& correctEval( MotsGraph &g ) const;

  void printEval( MotsGraph &g ) const;
  inline EVAL_TYPE getEval( int i )
  ...
  inline const EvalVector* getEval()
  inline list<edge>::iterator getPathBegin()
  inline list<edge>::iterator getPathEnd()
  inline const EvalMap* getDistanceMap()
  inline void incNSearch( int count )
```

```
  inline int getNSearch()
  inline bool getEdgeTabuStatus ( edge e )
  inline bool operator ==( const Pareto& p )
  inline bool getDeleted() const
  inline void setDeleted()
  inline void incKilledP()
  inline bool isTabuNode( int id )
  inline void markHyperCheck()
  ...
  DistinguishedName dName;

private:
  EvalVector evalVec; //eval of the path, for each crit. one
  list<edge> edgePath;
  EvalMap distance2Target;
  map<int,int> tabuNodeMap; // first is the node id
  int nSearchUsed;
  INSTYPE iType;
  bool toBeDeleted;
  int iKilledOtherP;

  friend class Paretoset;
};
```

A.4.1 Daten Pareto

EvalVector evalVec ist die Summe der Bewertungen aller Kanten des Weges, also die Bewertung des Weges.

list$< edge >$ **edgePath** ist die Liste der Kanten, die den Weg bilden.

EvalMap distance2Target ist die Entfernungsmap der Knoten des Weges zum Ziel. Diese wird verwendet, um beim Bilden neuer Wege über Nachbarwege, sehr rasch die Bewertung eines potentiell neuen Weges festzustellen.

map$< int, int >$ **tabuNodeMap** speichert die momentanen Tabu-Knoten mit ihren Zählern.

int nSearchUsed gibt an, wie oft der Weg schon Ausgangspunkt für die Nachbarschaftssuche war. Derzeit wird jeder Weg nur einmal verwendet.

INSTYPE iType ist eine Kennung, auf welche Art der Weg entstanden ist (Dijkstra, Nachbarschaft).

bool toBeDeleted Kennung, ob es sich um einen dominierten Weg handelt und damit aus der Pareto-Menge gelöscht wird.

int iKilledOtherP Kennung, wie viele Wege durch ihn aus der Pareto-Menge entfernt wurden. Einerseits für die Statistik, andererseits könnte über diese Kennung eine intelligentere Reihung der Wege gemacht werden, um den Vergleich bei neuen Wegen abzukürzen.

A.4.2 Methoden Pareto

Es gibt zwei Konstruktoren für die Bildung eines Pareto Objektes. Der eine wird verwendet, wenn ein Weg komplett neu entsteht, also entweder ganz am Anfang bei der Bildung der Dijkstra Wege, oder später beim Bilden neuer Wege über unbenutzte Kanten (3.4). Hauptaufgabe ist die Berechnung der Distanzmap der einzelnen Knoten zum Zielknoten.

Der zweite Konstruktor wird verwendet, wenn ein Weg über die Nachbarschaft zu einem schon Bestehenden gefunden wurde. Im Unterschied zum ersten Konstruktor muss hier die Map der Tabu-Knoten aufgebaut bzw. weitergegeben werden.

insertNodes Die Methode insertNodes baut die Distanzmap auf.

relatedTo Die Methode relatedTo ist das Pendant zur gleichnamigen Methode in der Klasse EvalVector. Sie reicht die Anforderung an den EvalVector der eigenen Klasse weiter und retourniert das Ergebnis.

A.5 Klasse ParetoSet

Die Klasse ParetoSet verwaltet die Menge der pareto-optimalen Wege.

```
class ParetoSet {
public:
  // constructors
  ParetoSet();
  // this constructor uses the iterator to get the
  // edges and calculates the eval of the path
  PARETO_RESULT checkAndAdaptList( Pareto* p );
  PARETO_RESULT checkNewEval( const EvalVector* W );
  bool deleteDomP();
  ParetoIterator deleteDomP(ParetoIterator pIt);
  int insertP( Pareto* p );
  ParetoIterator getNextUnusedP(int levelSearch=0);
  // kick out solutions within the hyperplane of xtreme vectors
  int cleanWithConvexHull();
  void printBest( const WeightVector& e );
  void printName();
  bool checkAll();
  inline PARETO_RESULT ParetoSet::checkNewEval( Pareto* p )
  ...
  inline int getNrOfElems()
```

```
  inline ParetoIterator begin()
  inline ParetoIterator end()
  ...

private:
  // list of the collected pareto-optimal pathes
  list<Pareto*> pv;
};
```

A.5.1 Daten ParetoSet

list$<$ *Pareto*$*$ $>$ *pv* Liste der pareto-optimalen Wege.

A.5.2 Methoden ParetoSet

checkAndAdaptList Die Methode checkAndAdaptList läuft die Pareto--Menge ab und testet, ob der neue Weg optimal ist. Sie fügt den neuen Weg in die Liste ein. Dabei werden alle alten Wege, die vom neuen dominiert werden als "zu löschend" markiert.

checkNewEval Die Methode checkNewEval wird gerufen, um bei einem vorliegenden EvalVector festzustellen, ob er pareto-optimal ist. Diese Methode wird nur deshalb getrennt von *checkAndAdaptList* entwickelt, weil es mit Hilfe dieser Funktion möglich ist, den Vergleich durchzuführen, ohne ein Pareto Objekt anzulegen.

deleteDomP Die Methode deleteDomP gibt es in zweifacher Ausführung. In der einfachen Version werden alle zu löschende Objekte aus der Liste entfernt. In der zweiten Version wird zusätzlich ein Iterator an die Methode übergeben. Die Methode liefert ihrerseits einen Iterator zurück, der auf ein Element in der Liste zeigt, das noch nicht zur Suche verwendet wurde. Dieses Element muss in der Liste nach dem übergebenen Iterator stehen.

getNextUnusedP Methode getNextUnusedP liefert das nächste Element zurück, dass noch nicht für eine Nachbarschaftssuche verwendet wurde.

cleanWithConvexHull Methode cleanWithConvexHull bereitet die Daten für den Aufruf von qhull vor. Nach dem Aufruf werden alle Wege gelöscht, deren Bewertungsvektor sich innerhalb der konvexen Hülle befindet.

A.6 Klasse MotsGraph

Die Klasse MotsGraph verwaltet den dem Problem zugrundeliegenden Graphen. Sie ist abgeleitet von der Klasse graph, die in der GTL definiert wird.

```
class MotsGraph : public graph {
public:
```

```
  MotsGraph();
  GRAPH_RESULT load (const char* filename,
                     list<int>& iEval );
  GRAPH_RESULT loadViaDb (const char* dbName,
                          list<int>& iEval );
  void finishWeights();
  void setNrOfEval(int nr);
  void createEdge(int s, int t);
  // load the weights via command
  void createWeight(string weightString);
  void printEdgeEval();
  inline const WeightVector& getWeight()
  ...
  inline void setId(int id)
  inline void setSourceNodeId(int nodeId)
  inline void setTargetNodeId(int nodeId)
  inline int getTargetNodeId()
  inline const node& getTargetNode()
  inline const node& getSourceNode()
  inline const edge_map<double>& getAllEvals( int i )
  inline const edge_map<double>& getStdEval()
  inline void createEval(EVAL_TYPE e)
  inline void createWeight(double w)
  inline void resetInternalEvals()
  inline int getEdgeUsage( const edge& e )
  inline EvalVector* getEdgeEval( const edge& e )
  inline void incEdgeUsage( const edge& e, int i )
  inline const edge_map<int>& getAllUseCounter()
  inline const edge_map<double>& getwEval()
  inline void setSourceAndTarget()
  ...

private:
  int maxNodeIndex;
  list<int> ignoreEvalList;
  // data are set when graph is loaded
  node sourceNode;
  node targetNode;

  WeightVector weight; // weigths of the evaluations
  vector< edge_map <double> > allEvals;
  edge_map<double> stdEval;
  map<edge,EvalVector* > allEdgeEvals;
  edge_map<int> useCounter;
  map<int, node> id_2_node;
  evalMan* evalM;
  int sourceNodeId;
```

```
    int targetNodeId;
};
```

A.6.1 Daten MotsGraph

int maxNodeIndex ist eine Hilfsvariable beim Erzeugen der Kanten. Die Knoten des Graphen werden über die Kanten erzeugt. Wenn eine neue Kante eingefügt wird, dann werden Knoten, die durch diese Kante definiert sind, und noch nicht im Graphen existieren, erzeugt. Die Variable *maxNodeIndex* zeigt an, bis zu welchem Index die Knoten bereits erzeugt wurden.

list$<$ *int* $>$ **ignoreEvalList** ist eine Liste von Integerwerten die angibt, welche Bewertungen ignoriert werden sollen.

node sourceNode, targetNode sind Anfangsknoten und Zielknoten. Der Datentyp *node* wird in der GTL definiert.

WeightVector weight ist, falls vorhanden und nicht mittels -wi ausgeblendet, der Gewichtsvektor.

vector$<$ *edge_map* $<$ *double* $>>$ **allEvals** ist ein Vektor von Maps, wobei jeweils eine Map die Bewertung für alle Kanten einer Dimension enthält. Der Dijkstra- Algorithmus braucht eine Map mit einer Bewertung pro Kante.

edge_map$<$ *double* $>$ **stdEval** ist eine Bewertungsmap, die pro Kante den Wert 1 enthält. Diese Map wird bei der Wegbildung über bis dahin nicht benutzte Kanten (3.4) verwendet. edge_map ist eine map, die in der GTL definiert wird. Sie ist so aufgebaut, daß sie für jede Kante des Graphen die angegebenen Informationen enthält. In diesem Fall, zu jeder Kante ein double.

map$<$ *edge*, $EvalVector*$ $>$ **allEdgeEvals** enthält die gleiche Information wie *allEvals* in transponierter Form. Also pro Kante einen Vektor mit allen Bewertungen für diese Kante. In dieser Form ist sie im Algorithmus besser verwendbar.

edge_map$<$ *int* $>$ **useCounter** ist eine Map, die pro Kante zählt, wie oft sie verwendet wurde.

map$<$ *int*, *node* $>$ **id_2_node** ist eine Zuordnungsmap, die einem Integer den entsprechenden Knoten zuordnet.

evalMan* evalM verwaltet die Kantenbewertungen beim Einlesen des Graphen

int sourceNodeId, targetNodeId sind die Integer Pendants zu *sourceNode* und *targetNode*.

A.6.2 Methoden MotsGraph

load, loadViaDB Die Methoden *load* und *loadViaDB* lesen die Beschreibung des Graphen ein. *load* erwartet eine Beschreibung in dem erweiterten GML-Format *loadViaDB* erwartet eine Access Datenbank mit den entsprechenden Feldern ([2]). Die Methode *load* ruft einen Parser der wiederum Methoden des Objektes ruft. In *loadViaDB* werden die Daten direkt aus der Datenbank gelesen.

finishWeights, setNrOfEval, createEdge, createEval sind die callbacks, die vom Parser gerufen werden wenn er die entsprechenden Beschreibungen im Inputfile liest.

Alle restliche Methoden erschließen sich durch ihren Namen oder sind reine Hilfsfunktionen ohne Anspruch auf Originalität.

A.7 Klasse Mots

Die Klasse mots verwaltet den eigentlichen Ablauf des Algorithmus. Der Name leitet sich von dem Begriff multi-objective-tabu-search ab.

```
enum FINISH_TYPE { MOTS_FINISHED, MOTS_NORMAL, MOTS_COMPLETE };

class mots {
public:
  // Constructors
  Mots( string filename, string dbname,
        int neighbourhoodLevel, bool makeStat,
        int finish, int mots_exact, list<int>& ignoreEval,
        string weigthString, bool iHull, int tabu );
  int  run();
  // makes a dijkstra run and returns a list
  bool makeDijkstra( node s, node t,
             const edge_map<double>& w,
             list<edge>& newPath );
  // walk through Paretoset and analyze pathes
  // not being analysed before
  void searchNeighbour();
  // checks the neigbours of a path and return the numbers of
  // new Pareto elements
  int analysePath( Pareto* path );
  // from the start node check a new path to be light evaled,
  // this func is called recursive
  int followEdge( int level, edge e, Pareto* path,
          EvalVector accSum, list<edge>& newEdgeList );
  // traverses the whole graph, within spec. time
  FINISH_TYPE followEdgeComplete( node n, valVector
          oldAccSum, list<edge>& edgeList );
```

```
  // check if time ellapsed or Ctrl-C pressed
  // and make actions like print result
  bool checkOnFinish();
  void printStatistic();
  void printResult(FINISH_TYPE t);

private:
  string fileName;
  string dbName;
  MotsGraph g;
  ParetoSet ps;
  motsTime fini;
  motsTime motsExact;
  int nLevel;
  bool stat; // should statistics be gathered
  bool ignoreHull; // don't call qhull
  int tabuNr; // hoe many times is a node tabu
};
```

A.7.1 Daten Mots

string fileName, dbName speichern den Namen der Inputfiles, je nach Typus ist derzeit der jeweils andere Name leer.

MotsGraph g ist das Objekt, in dem der eigentliche Graph gespeichert ist. Die Basis wird in der GTL definiert, die Erweiterungen im Abschnitt A.6.

ParetoSet ps verwaltet die aktuelle Pareto-Menge.

motsTime fini steuert die Zeitverwaltung. Wenn ein Zeitfenster vorgegeben ist (Option -f), dann ist der Algorithmus angehalten innerhalb dieses Zeitfensters fertig zu werden.

motsTime motsExact ist die Zeitverwaltung für die erste Phase (Option -c, *komplette Suche*).

int nLevel speichert den vorgegebenen level für die Tiefensuche.

bool stat sagt aus, ob Statistikdaten gesammelt werden sollen.

bool ignoreHull steuert den Einsatz von qhull (Option -q).

int tabuNr sagt aus, wie oft ein Tabu-Knoten weitervererbt wird.

A.7.2 Methoden Mots

Mots Der Konstruktor Mots speichert die Optionen ab und ruft den Ladevorgang für den Graphen auf.

run Die Methode run ist die zentrale Methode des Programmes. In dieser Methode ist der gesamte Ablauf des Algorithmus enthalten. Alle Phasen bis auf die letzte können hier direkt im Code nachvollzogen werden. Der genaue Ablauf wird in Kapitel 3 beschrieben. Die Methode wird von den folgenden Hilfsfunktionen unterstützt.

makeDijkstra Die Methode makeDijkstra bekommt zwei Knoten und eine Bewertungsmap als Parameter und liefert den Dijkstraweg zwischen diesen Knoten bzgl. der Bewertung.

searchNeighbour Die Methode searchNeighbour durchforstet die bisher gesammelten Wege. Für alle, die bisher noch nicht zur Nachbarschaftssuche verwendet wurden, wird *analysePath* aufgerufen. Weiters wird in dieser Methode das Löschen von dominierten Wegen angestoßen.

analysePath Die Methode analysePath macht die ersten Schritte zur Nachbarschaftssuche für einen Weg. Der alte Weg wird sukzessive abgegangen und für jede abgehende Kante wird *followEdge* aufgerufen. Beim Traversieren des Weges wird die aufsummierte Bewertung der Kanten mitberechnet.

followEdge Die Methode followEdge stellt zuerst fest, ob der Zielknoten tabu ist. Falls nicht, wird festgestellt, ob der Zielknoten Teil des alten Weges ist. Wenn ja, wird die Gesamtbewertung berechnet und der Verwaltung der Pareto-Mengen übergeben. Diese stellt fest, ob der neue Weg pareto-optimal ist. Wenn der Knoten nicht Teil des alten Weges ist, dann wird bei dessen adjazenten Kanten mit einem rekursiven Aufruf von followEdge fortgesetzt. Abgebrochen wird, falls die maximale Tiefe der Suche erreicht ist.

checkOnFinish Die Methode checkOnFinish stellt fest, ob das Zeitlimit erreicht ist (bzw. je nach Aktivierung, wird auch ein Abbruch mittels Ctrl-C akzeptiert) und sorgt für die Ausgabe der Ergebnisse.

printStatistic, printResult Die Methoden printStatistic und printResult sorgen für die Ausgabe der Statistik und der eigentlichen Ergebnisse.

A.8 Trace eines Programmablaufes

Ich habe für das Beispiel aus Kapitel 4.3 das Programm mit der Option -a1 (interne Option zum tracen des Ablaufes) durchlaufen lassen. Ich habe monotone Stellen, z.B. mehrfache Hintereinander-Aufrufe von MotsGraph::createEdge

oder auch ParetoSet::checkNewEval durch ... ersetzt. In diesem trace sind nur die wichtigsten Aufrufe enthalten. Er soll einen groben Überblick geben.

```
ParetoSet::ParetoSet{
} exit
Mots::Mots{
  MotsGraph::load{
    MotsGraph::setNrOfEval{
    } exit
    MotsGraph::createEdge{
    } exit
    ...
    MotsGraph::createEdge{
    } exit
  } exit
  MotsGraph::createWeight{
  } exit
}exit
Mots::run{
  Mots::makeDijkstra{
  } exit
  Pareto::Pareto{
  } exit
  ParetoSet::checkAndAdaptList{
  } exit
  Mots::makeDijkstra{
  } exit
  Pareto::Pareto{
  } exit
  ParetoSet::checkAndAdaptList{
  } exit
  Mots::searchNeighbour{
    Mots::analysePath{
      Mots::followEdge{
        Mots::followEdge{
          ParetoSet::checkNewEval{
          } exit
        } exit
        Mots::followEdge{
        } exit
        Mots::followEdge{
        } exit
        Mots::followEdge{
        } exit
      } exit
      Mots::followEdge{
        Mots::followEdge{
```

```
      } exit
      Mots::followEdge{
        ParetoSet::checkNewEval{
        } exit
      } exit
    } exit
    Mots::followEdge{
      Mots::followEdge{
      } exit
      Mots::followEdge{
        ParetoSet::checkNewEval{
        } exit
      } exit
    } exit
    Mots::followEdge{
      Mots::followEdge{
      } exit
      Mots::followEdge{
        ParetoSet::checkNewEval{
        } exit
      } exit
    } exit
    Mots::followEdge{
      Mots::followEdge{
        ParetoSet::checkNewEval{
        } exit
        Pareto::Pareto1{
        } exit
        ParetoSet::checkAndAdaptList{
        } exit
      } exit
    } exit
  } exit
  ParetoSet::deleteDomP1{
  } exit
  Mots::analysePath{
  ...
  } exit
  Mots::analysePath{
  ...
  } exit
  ParetoSet::deleteDomP1{
  } exit
} exit
Mots::makeDijkstra{
} exit
Mots::makeDijkstra{
```

```
    } exit
    Pareto::Pareto{
    } exit
    ParetoSet::checkNewEval{
    } exit
    Mots::makeDijkstra{
    } exit
    Mots::makeDijkstra{
    } exit
    Pareto::Pareto{
    } exit
    ...
    ParetoSet::checkNewEval{
    } exit
    Mots::makeDijkstra{
    } exit
    Mots::makeDijkstra{
    } exit
    Pareto::Pareto{
    } exit
    Mots::searchNeighbour{
    } exit

#size 3

#eval 2

#PATH  1 3 7 8 10 evals 13 57

#PATH  1 2 6 9 10 evals 17 50

#PATH  1 3 6 9 10 evals 14 51

termination normal, convHull: 0 time 1
  } exit
```

Der Output des Programms gibt mit #size die Anzahl der pareto-optimalen Wege an, mit #eval die Anzahl der Bewertungen und darauf folgen die einzelnen Wege (#PATH) mit den Knoten und den dazugehörigen Bewertungen. Mithilfe des vorangestellten "#" ist es leichter möglich, den relevanten Output z.B. mittels *grep* oder *findstr* rauszufiltern. Bei einer Weiterverarbeitung der Ergebnisse kann das gute Dienste leisten. Anfang ({) und Ende (}) einer Methode sind immer mit entsprechenden Klammern gekennzeichnet. So ist es leichter mit einem dazu geeigneten Editor den Output zu verfolgen.