ShoppingTour

Einkaufsoptimierung
informatiCup 2012 Aufgabe 1

Kai Fabian
Hasso-Plattner-Institut, IT-Systems Engineering, 3. Semester
kai.fabian@student.hpi.uni-potsdam.de

Dominik Moritz
Hasso-Plattner-Institut, IT-Systems Engineering, 3. Semester
dominik.moritz@student.hpi.uni-potsdam.de

Matthias Springer
Hasso-Plattner-Institut, IT-Systems Engineering, 3. Semester
matthias.springer@student.hpi.uni-potsdam.de

Malte Swart
Hasso-Plattner-Institut, IT-Systems Engineering, 3. Semester
malte.swart@student.hpi.uni-potsdam.de

15. Januar 2012

Inhaltsverzeichnis

1 Einleitung

ShoppingTour ist ein in Python geschriebenes Programm zum Optimieren von Einkaufsfahrten, welches im Rahmen des InformatiCup 2012 erstellt wurde. Das Programm kann sowohl mit der in Qt geschrieben Oberfläche, als auch vollständig über die Kommandozeile benutzt werden. Bei Der Programmierung haben wir auch Wiederverwendbarkeit von Programmteilen geachtet und eine Architektur entwickelt, die Erweiterungen sehr einfach macht. Diese können zum Beispiel die Grafische Darstellung der Qualität des Suchergebnisses oder einfache Navigation durch die gefundenen Lösungen sein. Es werden zwei ausgereifte Verfahren zur Optimierung verwendet, wobei eines sogar Optimalität garantieren kann!

Die Architektur unseres Programms ist so aufgebaut, dass Programmlogik, Daten und die Gui getrennt wurden. Die Datenstrukturen liegen im Verzeichnis `data`. Die Programmlogik findet sich im Verzeichnis `program`. Die gesamte UI-Implementierung findet sich im Verzeichnis `gui`. Da die Ui teilweise generiert wird, finden sich im Ordner `gen` die generierten Ui-Dateien. Das verwendete Programm clingo findet sich in einem eigenen Verzeichnis `dist/clingo`, da es in C++ geschrieben ist und somit verschiedene Binaries für verschiedene Betriebssysteme ausgeliefert werden müssen.

Wir wünschen dem Leser an dieser Stelle viel Freude beim Studium dieser Dokumentation und des Quellcodes sowie der Benutzung des Programms.

2 Installation

Windows-Nutzer können weitgehende Installationsschritte durch die Verwendung unserer *Windows One-Click Distribution* in der beigelegten Datei `shoppingtour-w32.zip` vermeiden. Hierzu wird diese Datei in einen beliebigen Ordner entpackt und die enthaltene `shoppingtour.exe`-Datei ausgeführt.

Da dieses Projekt ausschließlich in der plattformunabhängigen Skriptsprache *Python* geschrieben wurde, ist eine Distribution auf fast alle Betriebssysteme möglich. Notwendige Voraussetzung hierfür ist jedoch, dass die im Projekt verwendeten notwendigen Bibliotheken für das Zielsystem verfügbar sind. Da dies meist nur für verbreitete Betriebssysteme der Fall ist, können wir eine Lauffähigkeit zum aktuellen Zeitpunkt nur für *Microsoft Windows* (2000, XP, Vista, 7, 8 Developer Preview), *Apple Mac OS X* (>= 10.4) und *Linux* (Kernel 2.6/3.0, insbesondere Distributionen "Debian", "Ubuntu" und "Gentoo") garantieren.

Im Folgenden werden die notwendigen Softwarepakete aufgeführt, die für die Verwendung von *ShoppingTour* erforderlich sind.

2.1 ShoppingTour

`http://www.myhpi.de/~kai.fabian/shoppingtour/`

ShoppingTour benötigt keine Installation und kann nach dem Entpacken in ein beliebiges Verzeichnis, welches das Ausführen von Anwendungen erlaubt, verwendet werden.

2.2 Python 2.7

`http://www.python.org/`

Als Interpreter für die verwendete Programmiersprache empfehlen wir die Referenzimplementierung *CPython* in der *Version 2.7*, wobei auch andere kompatible Python-Implementierungen verwendbar sein sollten, solange die weiteren Abhängigkeiten Kompatibilität zu diesen aufweisen.

Bezüglich der Python-Prozessorarchitekturen konnten die Intel 32-bit-Architektur (x86) sowie die AMD 64-bit-Architektur (x64) erfolgreich erprobt werden.

2.3 PyQt 4.9

`http://www.riverbankcomputing.co.uk/software/pyqt/download`

Das Benutzer-Interface verwendet Nokias Qt-Bibliothek. Die hierfür notwendigen Python-Bindungen werden durch Riverbank Computing Limited bereitgestellt. Für Windows existieren vorkompilierte Pakete, während Linux- und Mac-Nutzer die Bibliothek selbst kompilieren müssen, sofern nicht bereits vorgefertigte Pakete für das Betriebssystem

existieren (bspw. das Paket `py27-pyqt4` für Mac OS X unter Verwendung von MacPorts).

Anzumerken ist hierbei, dass zur Verwendung (und eventuellen manuellen Übersetzung) der PyQt-Bibliothek Nokias Qt-Bibliothek neben weiteren Abhängigkeiten (hierzu sei an die Seiten des PyQt-Distributors verwiesen) auf dem System vorhanden sein müssen.

2.4 Clingo (part of Potassco)

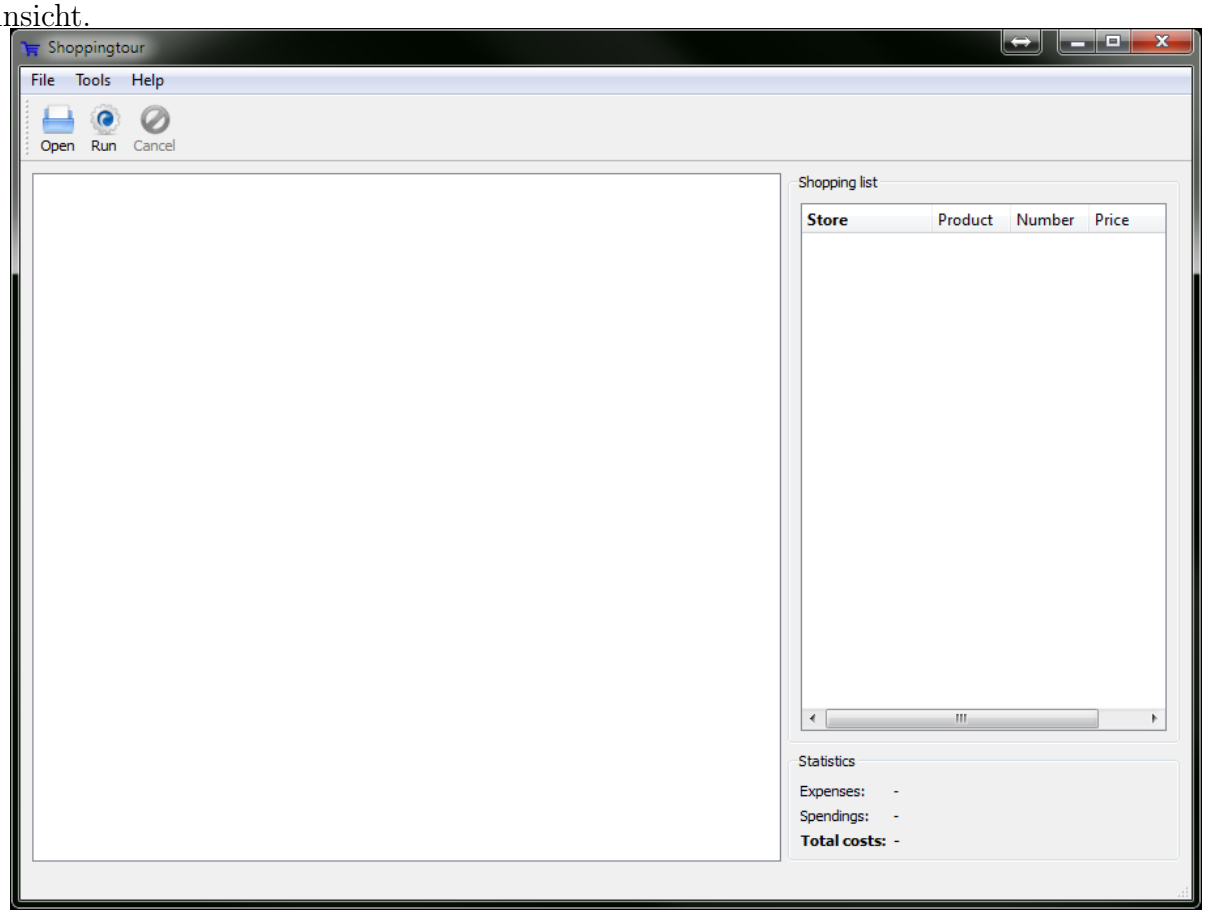
`http://potassco.sourceforge.net/`

Das von der Universität Potsdam gepflegte Potassco-Projekt stellt Softwarewerkzeuge für ASP-Programmierung (Answer Set Programmung) bereit. *ShoppingTour* verwendet *Clingo*, welches die Verbindung von clasp, einem Lösungsproblem für logische Probleme, und Gringo, einem Konvertierer zur Erzeugung von variablen-freien logischen Problembeschreibungen, darstellt. Vorkompilierte Binärdateien für Windows, Mac OS/Darwin und Linux sind von der Anbieterseite ebenso zu erhalten wie der Software-Quellcode.

3 Programmverwendung

3.1 Verwendung der graphischen Ansicht

Nach dem Programmstart befindet sich der Benutzer üblicherweise in der graphischen Ansicht.

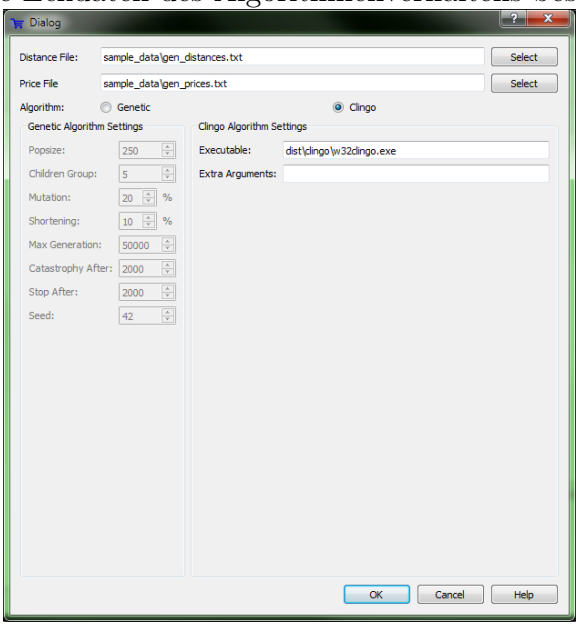


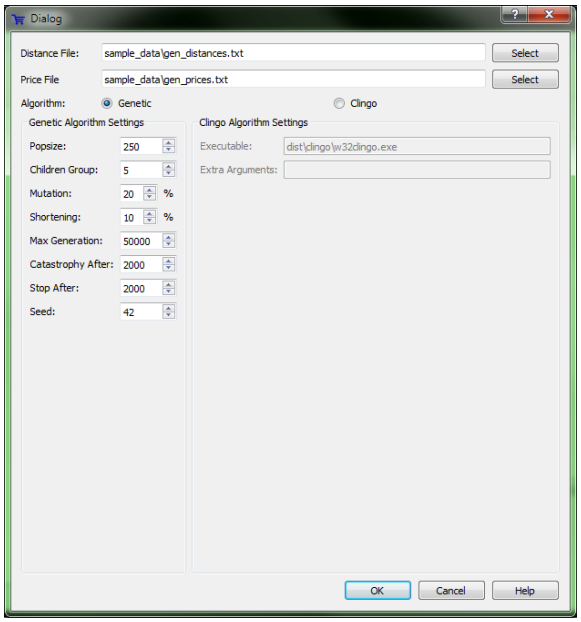
Der Benutzer öffnet nun eine Problemdefinition, in dem er den *Open*-Button in der Toolbar, oder wahlweise den gleichnamigen Menüpunkt im File-Menü, verwendet. Hierbei hat er die Wahl zwischen den beiden implementierten Algorithmen.

In der folgenden Ansicht wählt der Benutzer die Problembeschreibungsdateien über den *Select*-Button oder durch manuelle Eingabe in das Textfeld aus.

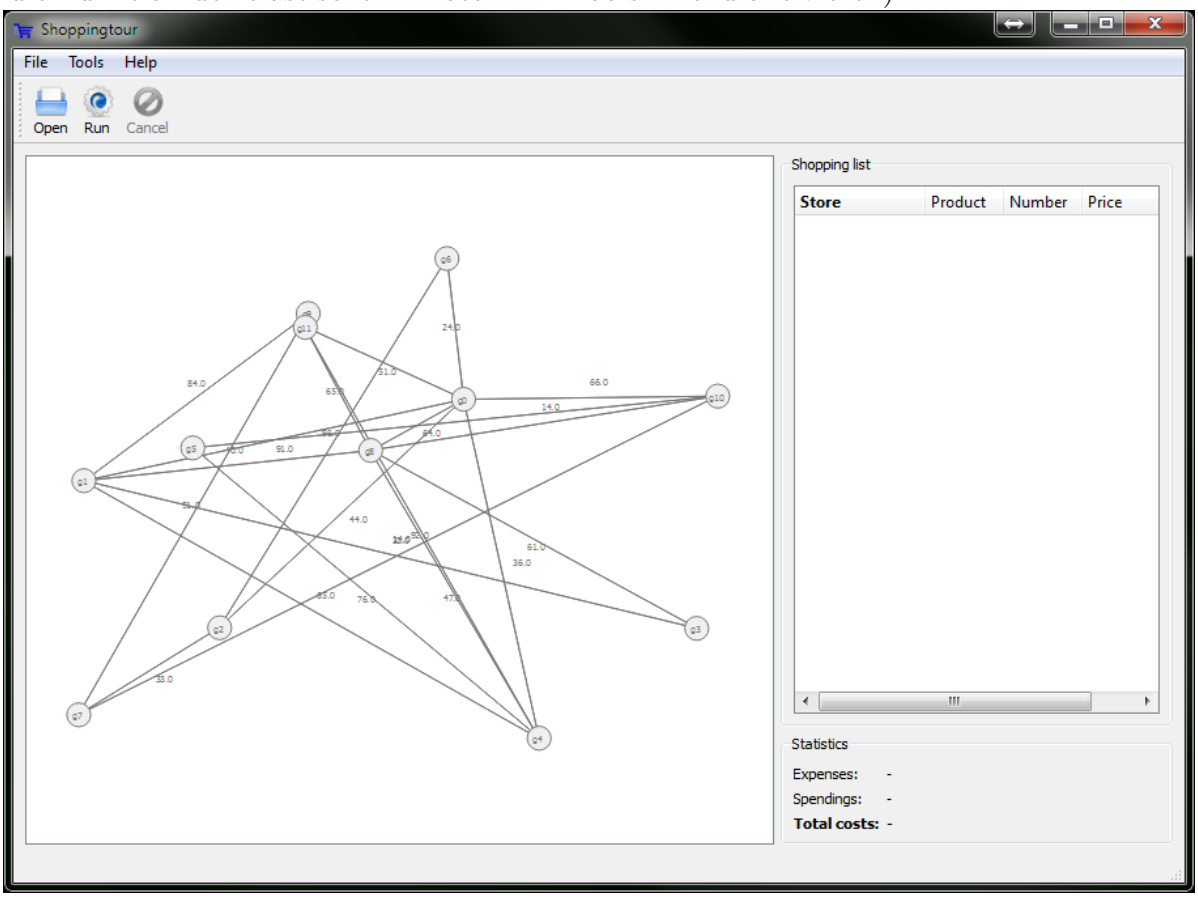
Für die Verwendung von Clingo zur Problemlösung muss der Pfad zur ausführbaren Programmdatei angegeben werden. In der *Windows One-Click Distribution* befinden sich diese im dist/clingo-Ordner; Windows-Benutzer müssen hier also den Pfad zur `w32clingo.exe`-Datei, also beispielsweise `dist/clingo/w32clingo.exe`, angeben. Der Pfad kann hierbei absolut oder relativ zum aktuellen Arbeitsverzeichnis angegeben werden.

Alternativ kann der Benutzer den genetischen Algorithmus wählen. Hierbei kann er die Eckdaten des Algorithmenverhaltens bestimmen.

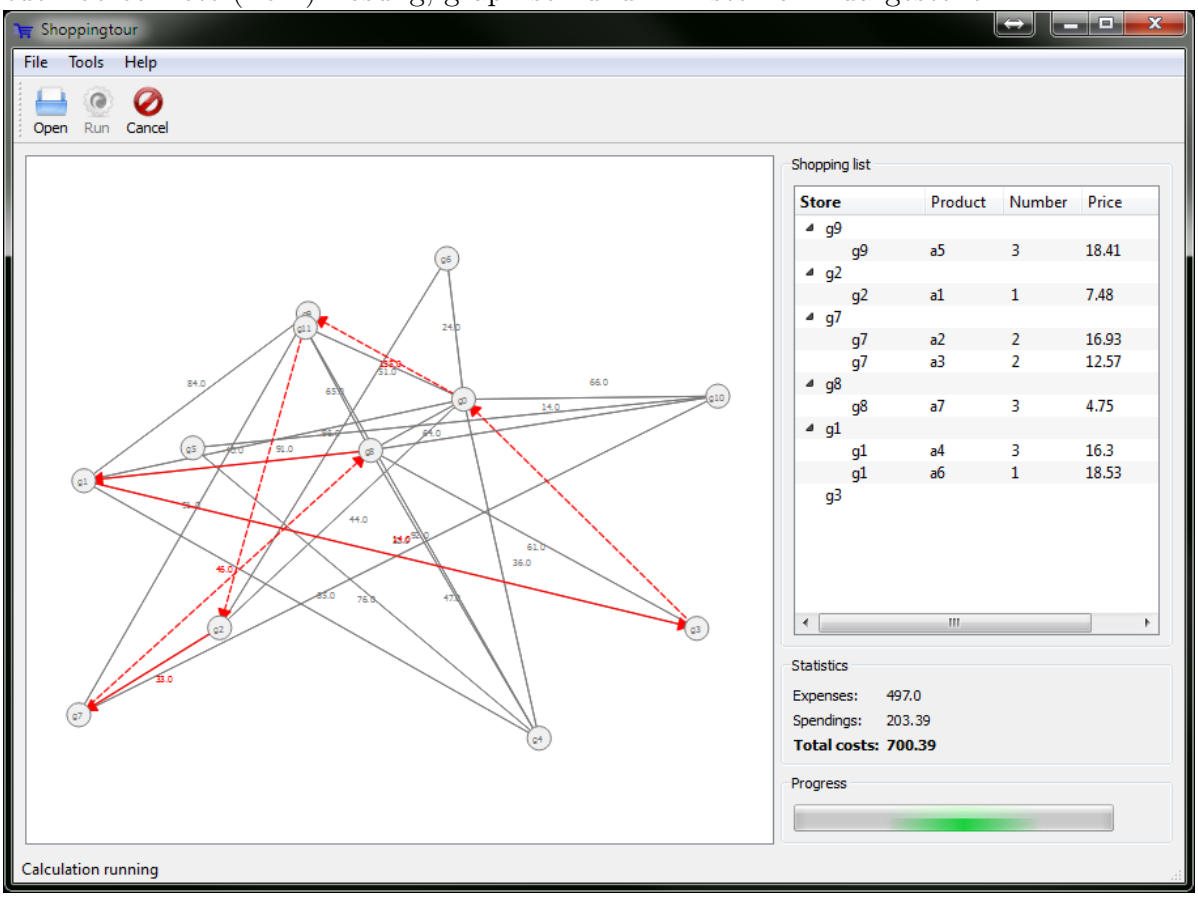




Der Benutzer sieht nun eine graphische Darstellung des Problembereiches. Er kann per Klicken und Ziehen die einzelnen Knoten des Graphen verschieben (optional kann er die Funktion der elastischen Knoten im "Tools"-Menü aktivieren).

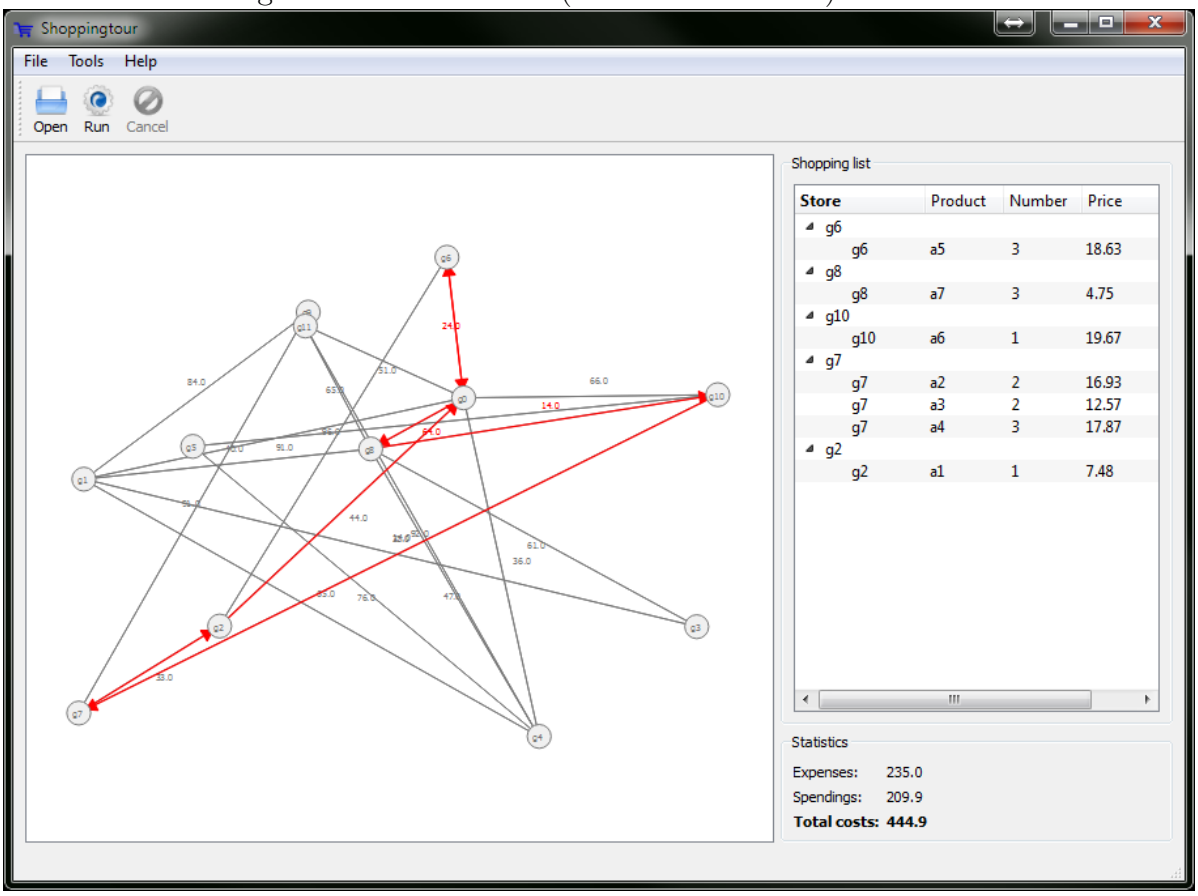


Um die Berechnung zu starten, klickt der Benutzer auf den *Run*-Button in der Toolbar. Während des Algorithmen-Durchlaufs wird ihm der aktuelle Programmzustand, also die aktuell berechnete (Teil-) Lösung, graphisch und in Listenform dargestellt.



Der Benutzer kann den Berechnungsvorgang selbstverständlich auch abbrechen. Hierzu wählt er entweder in der Toolbar die *Cancel*-Option oder alternativ die gleichnamige im Tools-Menü.

In der Endansicht wird der Benutzer über die beste gefundene Lösung informiert. Dies geschieht auch hier einerseits in graphischer Form und andererseits in Listenform. Die Listenform wird als *Shopping list* bezeichnet und beschreibt, in welcher Reihenfolge welche Geschäfte besucht werden sollen (erste Baumebene) und welche Gegenstände in welcher Anzahl dort gekauft werden sollen (zweite Baumebene).



4 Ein NP-schweres Problem

Bei dem vorliegenden Problem handelt es sich um NP-schweres Problem. Das ist deshalb interessant, weil es somit nahezu ausgeschlossen ist, einen Algorithmus zu finden, der die optimale Lösung des Problems effizient (in polynomieller Zeit) berechnet. Durch polynomielle Reduktion des *Travelling Salesman Problems* (TSP) auf das vorliegende Problem kann die NP-Schwierigkeit gezeigt werden.

Beim TSP geht man von einem ungerichteten, gewichteten Graphen aus, wobei die Knoten des Graphen Städte und die Kanten des Graphen Verkehrswege zwischen den Städten repräsentieren. Das Gewicht einer Kante gibt Aufschluss über die Länge des Weges zwischen zwei Städten. Nun soll eine Rundreise, also ein Pfad mit dem gleichen Start- und Endknoten, mit dem kleinstmöglichen Reiseweg gesucht werden, wobei alle Städte einmal besucht werden müssen.

Das TSP lässt sich auf das Problem der Einkaufsplanung reduzieren, indem man den Graphen ohne Veränderung übernimmt[1]. Wenn $S = \{s_1, s_2, \ldots\}$ die Menge der Städte ist, erzeugen wir für jede Stadt s_i einen Artikel a_i, den es nur in dieser Stadt zu kaufen gibt. Für die letzte Stadt s_n gibt es keinen Artikel, denn das soll unser Start- und Endpunkt sein, an dem die Reise beginnt. Der Preis der Artikel spielt keine Rolle, solange er endlich ist, also setzen wir ihn einfach auf 0.

Jeder Artikel soll nun einmal eingekauft werden. Durch die Lösung des Einkaufsplanungsproblems erhalten wir eine optimale Lösung des TSP, denn es werden die Gesamtkosten minimiert, die in diesem Fall nur aus den Reisekosten bestehen. Die Reisekosten aber ist die Länge der Rundreise. Um jeden Artikel einzukaufen, muss des weiteren jede Stadt einmal besucht werden. Als Ergebnis erhalten wir u.a. die Reihenfolge, in der die Städte besucht werden sollen, was der Lösung des TSP entspricht. Die Transformation des Problems ist trivialerweise in polynomieller Zeit möglich, denn es muss lediglich für jede Stadt ein Artikel erstellt werden (linearer Zeitaufwand).

Um zu zeigen, dass es sich außerdem um ein NP-vollständiges Problem handelt, muss nachgewiesen werden, dass das Problem in der Menge NP enthalten ist. Es muss also gezeigt werden, dass eine Lösung in polynomieller Zeit verifiziert werden kann. An dieser Stelle soll nur das Entscheidungsproblem betrachtet werden: Gibt es eine Lösung des des Einkaufsplanungsproblems, deren Gesamtkosten unter dem Betrag n liegen? Eine mögliche Lösung kann durch Aufsummieren der Reisekosten zwischen den Einkaufsläden[2] und dem Addieren der Artikelpreise bei den entsprechenden Einkaufsläden in polynomieller Zeit[3] verifiziert werden. Somit handelt es sich um ein NP-schweres Problem, das selbst in NP liegt, also folglich um ein NP-vollständiges Problem.

[1] Einkaufsgeschäfte sind Städte.
[2] Die Reisekosten ergeben sich aus den Kantengewichten.
[3] Wie man sich leicht überlegen kann, macht es keinen Sinn, eine Kante mehr als zweimal zu benutzen.

5 Pre- und Postprocessing

Um die Implementierung unserer Algorithmen zu verbessern und den Zustandsraum zu verkleinern, wird vor der Ausführung ein Preprocessing-Schritt und nach der Ausführung ein Postprocessing-Schritt durchgeführt.

5.1 Preprocessing

Als Eingabe erhält das Programm u.a. eine Adjazenzmatrix eines gewichteten, ungerichteten Graphen, in dem die Geschäfte auf Knoten und die Wege zwischen den Geschäften auf Kanten abgebildet werden. Das Gewicht einer Kante ist der Abstand zwischen zwei Geschäften bzw. die Kosten, die bei der Fahrt entstehen. Es ist auch möglich, dass zwischen Geschäften gar keine Verbindung existiert, was entweder bedeutet, dass das Geschäft nur über ein anderes Geschäft erreichbar ist, oder dass der Graph nicht zusammenhängend ist.

Falls der Graph nicht zusammenhängend ist, können die Knoten, die nicht vom Start- und Endknoten aus erreichbar sind, einfach verworfen werden. Diese sind dann nämlich nicht relevant für die Berechnung, weil sie in keiner validen Lösung auftauchen können. Sie würden den Suchraum nur unnötig vergrößern. Nicht erreichbare Knoten können mit einer Tiefensuche in linearer Zeit[4] gefunden und gelöscht werden.

Die eigentlichen Algorithmen entscheiden nur, ob und in welcher Reihenfolge bestimmte Knoten angesteuert werden sollen. Es wird also nicht direkt betrachtet, welchen Preis die Einkaufsartikel haben. Die Lösung, die ein Algorithmus generiert, ist nur eine Liste mit den anzusteuernden Knoten. Welche Artikel bei welchem Geschäft gekauft werden, muss nicht gespeichert werden. Dazu geht man einfach alle Artikel durch ein wählt für jeden Artikel das Geschäft unter den besuchten Geschäften aus, das den Artikel zum niedrigsten Preis anbietet.

Eine solche Rundreise kann aber sehr groß sein, da es passieren kann, dass ein Geschäft mehrmals besucht werden muss[5]. Durch eine Optimierung im Preprocessing kann der Suchraum aber stark verkleinert werden.

Dazu wird der Graph in einen vollständigen Graphen umgewandelt. Jeder Knoten ist dann von jedem anderen Knoten aus direkt erreichbar. Als Kantengewicht einer hinzugefügten Kante wählt man den minimalen Abstand zwischen den beiden betroffenen Knoten. In einem solchen Graphen muss kein Knoten mehr als einmal besucht werden. Um einen Artikel in einem bestimmten Geschäft zu kaufen, reicht es aus, den Knoten ein einziges Mal zu besuchen. Von diesem Knoten aus ist aber jeder andere Knoten direkt erreichbar. Man steuert nun also nur Geschäfte an, in denen man auch etwas kaufen will. Fälle, in denen man einen Knoten nur ansteuert, um einen anderen Knoten zu erreichen, gibt es nicht mehr.

[4] $\mathcal{O}(|V| + |E|)$, wobei $|V|$ die Anzahl der Knoten und $|E|$ die Anzahl der Kanten ist.

[5] Man denke z.B. an einen Fall, wo ein Geschäft A nur über ein einziges anderes Geschäft B erreichbar ist. Dann muss man dieses Geschäft B mindestens zweimal besuchen, einmal um zu A zu gelangen und einmal um A wieder zu verlassen. Analog lassen sich Fälle konstruieren, bei denen ein Geschäft beliebig oft besucht werden muss.

Der Suchraum kann durch diese Optimierung verkleinert werden, da nur noch zyklenfreie Rundreisen[6] betrachtet werden. Ein Algorithmus muss nur noch entscheiden, welche Geschäfte er überhaupt besuchen will und dann eine optimale TSP-Rundreise finden.

5.2 Implementierung des Preprocessing

Um den Graphen in einen vollständigen Graphen zu überführen, wird zunächst mit dem Floyd-Warshall Algorithmus der kürzeste Weg zwischen allen Paaren von Knoten berechnet[7]. Es wird sowohl die Länge als auch der eigentliche Weg gespeichert. Die Adjazenzmatrix des Graphen kann nun mit den berechneten Längen vervollständigt werden.

5.3 Postprocessing

Die generierte Lösung eines Algorithmus kann Kanten benutzen, die im vollständigen Graphen zwar enthalten sind, nicht aber im ursprünglichen Graphen. Solche Kanten müssen im Postprocessing identifiziert werden und durch den kürzesten Pfad zwischen den beiden betroffenen Knoten ersetzt werden. Das ist mit wenig Aufwand möglich, weil während der Berechnung der minimalen Pfade im Floyd-Warshall Algorithmus auch die Knoten, aus denen ein Pfad besteht, gespeichert wurden.

Die grafische Oberfläche zeigt Zwischenlösungen der Algorithmen an. Kanten, die nur im vollständigen Graphen existieren, nicht aber im originalen Graphen, sind gestrichelt eingezeichnet. Die endgültige Lösung enthält keine solchen Kanten mehr. Stattdessen kann es sein, dass Knoten dort mehrmals besucht werden.

6 Optimierungsalgorithmen

Nach dem Preprocessing wird auf die Daten ein Optimierungsalgorithmus angewendet, welcher iterativ bessere Lösungen findet und anschließend die beste gefundene präsentiert. Wir haben aus den Erfahrungen des letzten Informaticups und erneuten Analysen verschiedene Algorithmen evaluiert. Um die Komplexität nicht unnötig zu steigern wollten wir uns auf maximal zwei Algorithmen beschränken. Um trotzdem ausrechend Flexibilität zu behalten sollen die Parameter der Algorithmen aber so weit wie möglich anpassbar sein. Da die Probleminstanzen im Allgemeinen eine geringe Größe (ca. 10 Läden und ebensoviele Produkte) aufweisen und wir den Suchraum durch unser Preprocessing schon weit einschränken konnten, haben wir uns dazu entschieden auch die Möglichkeit zur Berechnung von optimalen Lösungen zu implementieren.

Nachdem wir verschiedene Metaheurstiken wie *Simulierte Abkühlung*, *Greedy-Suche* und *Tabusuche* untersucht haben, haben wir uns dazu entschieden einen *genetischen*

[6]Eine Rundreise ist natürlich auch ein Zyklus, aber dieser Zyklus zählt hier nicht.
[7]Laufzeit $\mathcal{O}(|V|^3)$ bei $|V|$ Geschäften.

Algorithmus für die Lösung des Problems zu nutzen. Greedy-Suche hätte leicht in einem lokalen Minimum landen können, sodass dieser Algorithmus für uns ausschied. Alle genannten Metaheuristiken, einschließlich dem genetischen Algorithmus, hätten das Problem der lokalen Minima gelöst, aber die Performance von genetischen Algorithmen ist im Allgemeinen höher.

Da Meteheuristiken zwar oft sehr gute und manchmal auch optimale Lösungen finden können, aber diese nicht garantieren oder auch nur nachweisen können, haben wir einen weiteren Algorithmus implementiert. Das Problem der Optimalität liegt darin, dass Metaheuristiken mit Zufall arbeiten und den Suchraum nicht vollständig explorieren. Da das Problem aus der Aufgabenstellung NP-vollständig ist und selbst der Nachweis der Optimalität einer Lösung NP-vollständig ist, kann es passieren, dass eine Metaheuristik nur eine unzureichend gute Lösung findet[8]. Die Nutzung von Clingo ermöglicht es uns, in relativ kurzer Zeit sehr gute Lösungen zu finden und bei der letzten Lösung auch Optimalität zu garantieren! Damit ist unser Programm in diesem Punkt allen Implementierungen die nur auf Metaheuristiken aufbauen (wie auch unser genetischer Algorithmus), überlegen.

An dieser Stelle möchten wir noch anmerken, dass auch die Metaheuristik *Simulierte Abkühlung* implementiert wurde. Jedoch nicht zur Berechnung einer Lösung des Problems sondern zur Verteilung der Knoten des Graphen auf der grafischen Ansicht. Der Algorithmus befindet sich in der Klasse `PositionCities` in der gleichnamigen Datei. Da die Funktionsweise des Algorithmus allgemein bekannt ist und der Algorithmus zudem nur zur grafischen Darstellung dient, möchten wir an dieser Stelle nur beschreiben, wie die Bewertung einer Lösung im Groben berechnet wird. Zu jeder Lösung wird für jedes Paar von Knoten der Abstand der Knoten auf der grafischen Ansicht mit dem Abstand der Knoten in der Adjazenzmatrix verglichen. Diese Abstände von grafischer Ansicht und Adjazenzmatrix verwenden subtrahiert, gewichtet und dann aufsummiert. Diesen Wert gilt es zu minimieren. Die Verteilung der Knoten mit diesem Algorithmus ist dann interessant, wenn es sich um echte, sinnvolle Abstände, d.h. um einen metrischen Graphen handelt. Bildet man beispielsweise ein existierendes Straßennnetz als Graphen ab und berechnet dann die grafische Darstellung mit diesem Algorithmus, erhält man ein *sinnvolles* Abbild, wo sich z.B. Straßen nicht überschneiden, wenn an dieser Stelle keine Kreuzung ist. Und dazu muss die Position der Knoten nicht gespeichert werden, es wird lediglich die Graphstruktur benötigt.

Bei der Implementierung haben wir darauf geachtet, dass die Algorithmen als Generatoren/ Iteratoren implementiert sind. Das heißt, es kann ganz einfach von einer Lösung zu einer weiteren gewechselt werden. Des Weiteren kann die Fortsetzung leicht verzögert oder sogar abgebrochen werden.

Wir haben des Weiteren bei unserer Implementierung nicht auch Randfälle, wie widersprüchliche Daten, fehlerhafte Eingabedateien oder nicht zusammenpassende Eingabedateien geachtet, um den Aufwand nicht zu sehr in die Höhe zu treiben.

[8] In unseren Tests funktionierte der genetische Algorithmus aber recht gut.

6.1 ASP solver clingo

Clingo ist Programm, das ähnlich wie Prolog Logische Probleme löst. Zum vollständigen Verständnis dieses Abschnitts sind Grundlagen aus der Logikprogrammierung und KI erforderlich. Unabhängig vom Verständnis des Programms ist festzustellen, dass mit dieser Lösung Optimalität garantiert werden kann und gleichzeitig relativ kurz gerechnet wird.

Die Stelle im Programm, an der die Implementierung zu finden ist, lautet `program/clingo.py`.

Das Eingabeformat von Clingo (bzw in diesem Fall Gringo) ähnelt dem von Prolog, wobei die Syntax erweitert wurde und auch Minimierungskriterien ermöglicht. Da die Programmbeschreibung von Clingo keine induktive wie Python, C++ oder Java ist, müssen auch andere Paradigmen angewendet werden.

Clingo steht für *clasp on Gringo* und kombiniert dadurch die beiden Systeme zu einem einfach zu verwendenden Programm. *clasp* ist ein Problemlöser für erweiterte Logikprobleme. Es kombiniert abstrakte Modellierungsmöglichkeiten des *answer set programming (ASP)* mit aktuellen lösungsalgorithmen aus dem Bereich des boolschen Constraint-Lösens. Dabei werden besonders die sehr effizienten konfliktgetriebenen Nachbarschaftssuchen verwendet. Dies ist eine Technik die sich in der Vergangenheit als sehr effizient herausgestellt hat. Clasp hängt dabei nicht von bestehenden Programmen ab sondern wurde von Grund auf neu entwickelt. Da clasp variablenfrei arbeitet, wird ein grounder benötigt, der eine bestehende Problemrepräsentation in eine gegroundete Version umwandelt. Ein solcher grounder ist Gringo.

Für uns von Interesse sind allerdings weniger die technischen Details, als die tatsächliche Nutzbarkeit von Clingo für unser Problem. Da Clingo einen besonderen Teil unseres Programms darstellt, haben wir die leicht verständliche Dokumentation mit angefügt (`clingo_guide.pdf`).

Unsere Problembeschreibung besteht aus vier Teilen. Der erste Teil ist die Definition des Graphs, wobei diese in Abhängigkeit vom Problem automatisch generiert wird. Ein Beispiel kann so aussehen.

```
1 % Nodes
2 node(0..3).

4 % (Directed) Edges
5 edge(X,Y) :- node(X),node(Y),X!=Y.

7 % Products
8 product(1..6).
```

Der zweite Teil ist die Definition der Kosten ,die ebenfalls automatisch generiert wird. Dabei werden die Pfadkosten E für die Reise von X nach Y als `expenses(X,Y,E)` und die Produktkosten C des Produkts P im Shop X als `cost(P,X,C)` dargestellt. `expenses(X,Y,E) :- expenses(Y,X,E).` definiert die Rückrichtung der Kanten. Kosten werden allgemein in Teilen von 100 Repräsentiert um Fließkommaoperationen zu verhindern.

```
% Edge Costs/ Traveling Expenses
expenses(0,1,300). expenses(0,2,400). expenses(0,3,600).
expenses(1,2,200). expenses(1,3,500).
expenses(2,3,300).

expenses(X,Y,E) :- expenses(Y,X,E).

% Product Costs
cost(1,1,1495). cost(1,2,1699). cost(1,3,1150).
cost(2,1,499).  cost(2,2,459). cost(2,3,499).
cost(3,2,1290). cost(3,3,1199).
cost(4,1,599). cost(4,2,455). cost(4,3,495).
cost(5,1,1990). cost(5,2,1995).
cost(6,2,819). cost(6,3,799).
```

Der Dritte Teil ist die Definition der Lösung. `selected(X).` beschreibt, welche Nodes betrachtet werden, also in der Lösung auftauchen sollen. `cycle(X,Y)` beschreibt, dass es eine Verbindung von X nach Y gibt. `canReach(X,Y)` definiert Erreichbarkeit im Graphen. `minimum_cost(P,CC)` definiert die geringsten Kosten CC für das Produkt P. Hier tritt die Optimierung zum Vorschein, dass wie nicht betrachten, wo ein Produkt gekauft wird, sondern nur, zu welchem Preis (Erklärung im Abschnitt Preprocessing). `first_sorted_cost`, `sorted_costs` und `not_minimum_cost` sind Optimierungen für die Ermittlung der minimalen Kosten. Dabei wird vor dem eigentlichen Grounden der Lösung eine Sortierung der Preise erstellt, sodass die Suche anschließend um ein Vielfaches schneller wird. An sich hätte auch eine einfache Definition gerecht, die Beschreibt, dass es kein günstigeres gibt. Diese Version wäre allerdings um einiges langsamer. Am Ende wird mit der Definition von `bought` sichergestellt, dass auch wirklich jedes Produkt gekauft wird.

```
% Generate
selected(0).
{ selected(X) } :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- selected(X). % outgoing
1 { cycle(X,Y) : edge(X,Y) } 1 :- selected(Y). % incoming

% Define
:- cycle(X,Y), cycle(Z,Y), Z!=X. % just a single outgoing
:- cycle(X,Y), cycle(X,Z), Y!=Z. % just a single incoming

canReach(X,Y) :- cycle(X,Y).
canReach(X,Y) :- cycle(X,Z), canReach(Z,Y).
:- not canReach(0,0). % there has to be a cycle from 0 to 0
% every selected node has to be reachable
:- not canReach(0,X), selected(X).
```

```
17 cost(P,C) :- cost(P,Y,C). %simplification
18 selected_costs(P,C) :- selected(Y), cost(P,Y,C).

20 sorted_costs(P,C1,C2) :- cost(P,C1), cost(P,C2), C1<C2,
21         not cost(P,C3):cost(P,C3):C1<C3:C3<C2.
22 first_sorted_cost(P,C) :- product(P),
23         C = #min [ cost(P,CC) = CC ], C != #supremum.
24 not_minimum_cost(P,C) :- first_sorted_cost(P,C),
25         not selected_costs(P,C).
26 not_minimum_cost(P,C) :- not_minimum_cost(P,C2),
27         sorted_costs(P,C2,C), not selected_costs(P,C).

29 minimum_cost(P,C) :- first_sorted_cost(P,C),
30         not not_minimum_cost(P,C).
31 minimum_cost(P,CC) :- not_minimum_cost(P,C),
32         sorted_costs(P,C,CC), not not_minimum_cost(P,CC).

34 % buy every product
35 bought(P) :- cost(P,Y,C), selected(Y).
36 :- not bought(P), product(P).
```

Als letztes wird noch definiert, was minimiert werden soll.

```
1 % Optimize
2 #minimize [ cycle(X,Y) : expenses(X,Y,E) = E, minimum_cost(P,C) = C ].
```

Anschließend muss Clingo nur noch mit den Beschreibungen als Eingabe ausgeführt werden und die Ausgabe entsprechend geparsed und Nachbearbeitet (Postprocessing) werden.

6.2 Genetischer Algorithmus

Beim vorliegenden Problem handelt es sich um eine leicht modifizierte Variante des TSP. Genetische Algorithmen lassen sich leicht auf das TSP-Problem anwenden. Genetische Algorithmen sind recht bekannt, deshalb wollen wir an dieser Stelle nur auf die Besonderheiten unserer Implementierung eingehen.

Eine dieser Besonderheiten ist die Darstellung der Individuen. Dabei wird nicht einfach eine Liste von besuchten Shops benutzt, da diese Repräsentation nach einem Crossover nicht notwendigerweise eine gültige Lösung generiert. So würde bei den naiven Liste `[0,1,3,2]` und `[0,3,1,2]` mit einem Crossover in der Mitte ein ungültiges Individuum `[[0,1,1,2]]` entstehen. Es ist ungültig, da die 1 zweimal besucht würde.

Aus diesem Grund repräsentieren wir die Individuen als Liste, in der die einzelnen Werte für Indizes stehen. Beim Auflösen werden diese nacheinander an auf eine sortierte Liste angewendet und das jeweilige Element entfernt. Nehmen wir beispielsweise die interne Darstellung `[2,0,0]`. Nun soll diese in die die normale Repräsentation überführt werden. Dazu nehmen wir die sortierte Liste `[1,2,3]` und entfernen zuerst das

dritte (Beginn bei 0, also Index 2). Nun haben wir eine Liste mit einem Element `[3]` und den Rest der sortierten Liste `[2,3]`. So verfahren wir weiter und erhalten die Liste `[3,1,2]`. Vorteil dieser Darstellung ist, dass sie beim Crossover immer eine gültige Lösung generiert und der Algorithmus wesentlich effizienter arbeiten kann.

Mutation tritt auf einem bestimmten Individuum mit einer bestimmten Wahrscheinlichkeit auf. Zuerst wird zufällig ein Gen, also ein Element der Liste ausgewählt. Dieses Element wird dann zufällig gesetzt. Die Anzahl der Gene, die pro Indiviuduum verändert werden, kann variiert werden.

Eine Besonderheit in unserem genetischen Algorithmus ist das Auftreten von *Katastrophen*. Bei einer Katastrophe wird eine große Anzahl an Individuen durch Mutation verändert, während die *normale* Mutation nur sehr wenige Individuen betrifft. Das führt zwar dazu, dass unmittelbar nach der Mutation die Individuen insgesamt schlechter werden. Allerdings konnten wir feststellen, dass die endgültige Lösung dadurch etwas besser wird.

Die Abbruchbedingung unseres genetischen Algorithmus ist das Durchlaufen einer großen Anzahl von Iterationen (Generationen), ohne dass ein besseres Individuum gefunden wurde. Der Vorteil gegenüber einer festen Anzahl an Iterationen liegt darin, dass der Algorithmus auch bei größeren Probleminstanzen gut funktioniert.

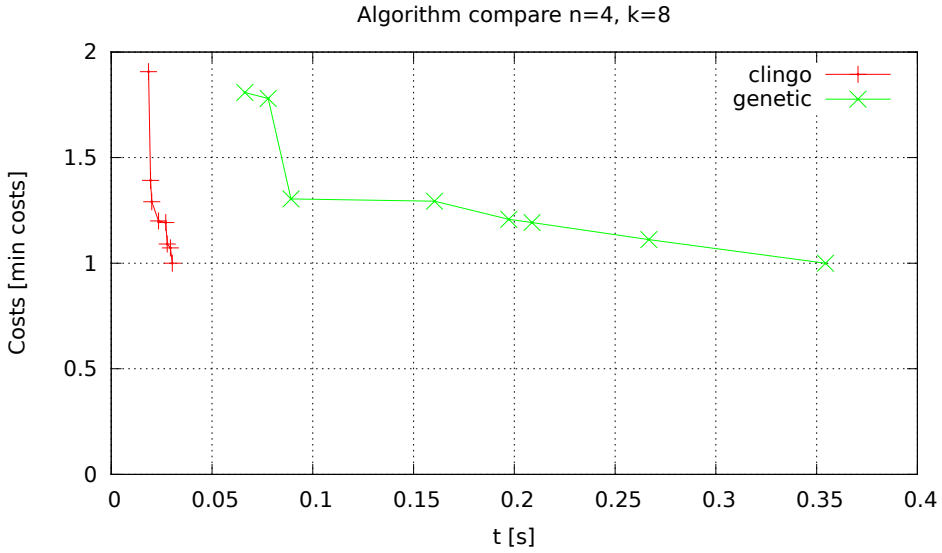
Der Code zum genetischen Algorithmus befindet sich in der Klasse `Genetic` in der Datei `program_genetic.py`.

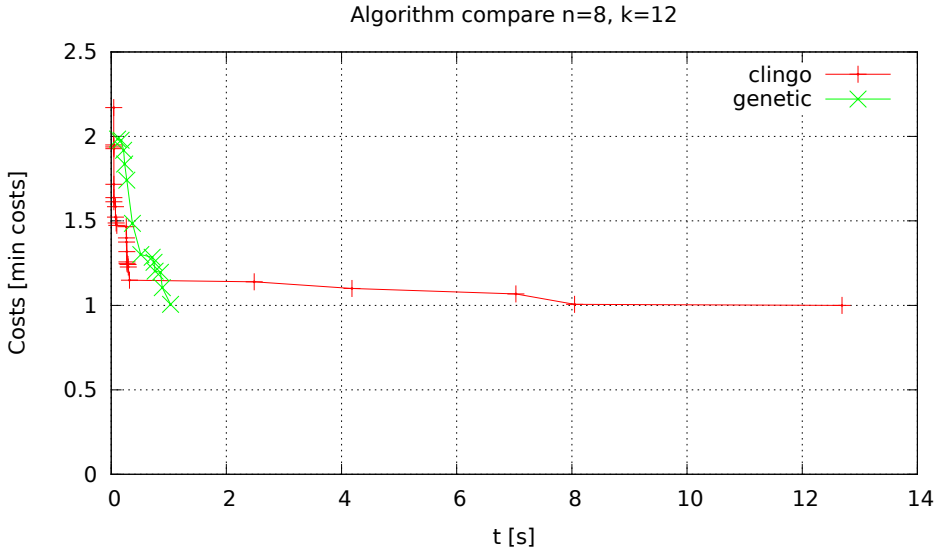
7 Pre- und Postprocessing

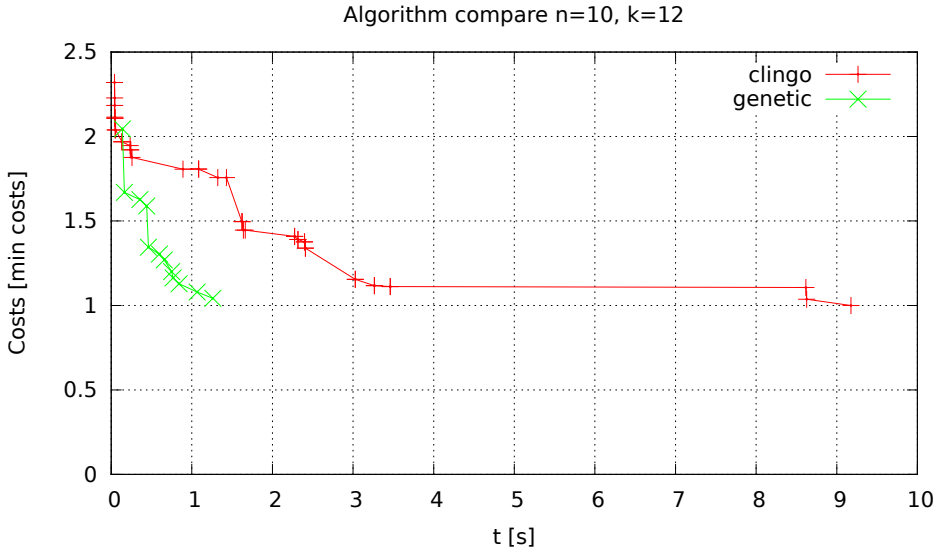
Es wurden ein paar sehr grundlegende Untersuchungen bezüglich der Abhängigkeiten des Programs von n und k durchgeführt.

7.1 Einwicklung der Lösungsgüte

Für verschiedene n und k Paare wurde zufällig ein Problem generiert und anschließend unterschied wie lange die Algorithm benötigen um Lösungen für dieses Problem zu finden. Die Güte der Lösungen wird dabei im Verhältnis zur optimalen Lösung angegeben (Die mittels clingo gefunden werden kann).

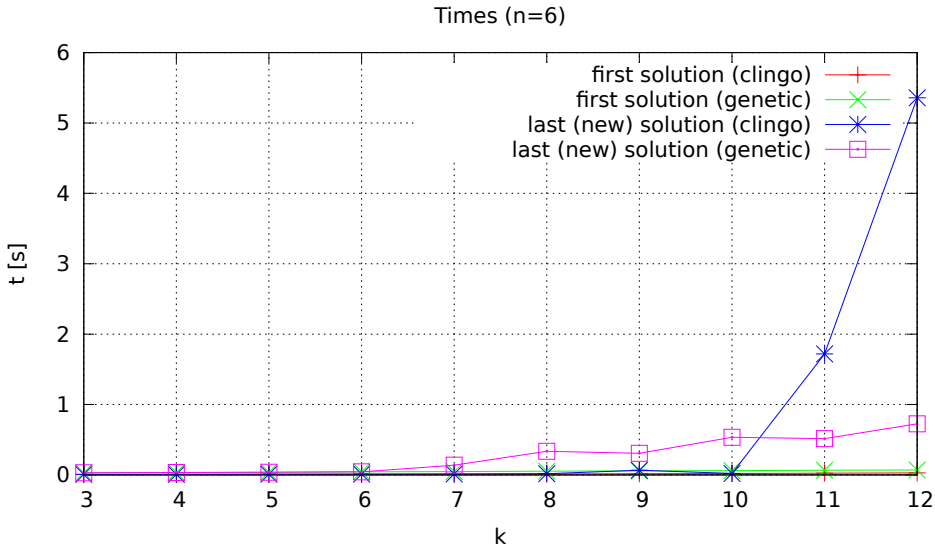

Algorithm compare n=4, k=8
Costs [min costs]
t [s]
clingo
genetic
0
0.5
1
1.5
2
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4

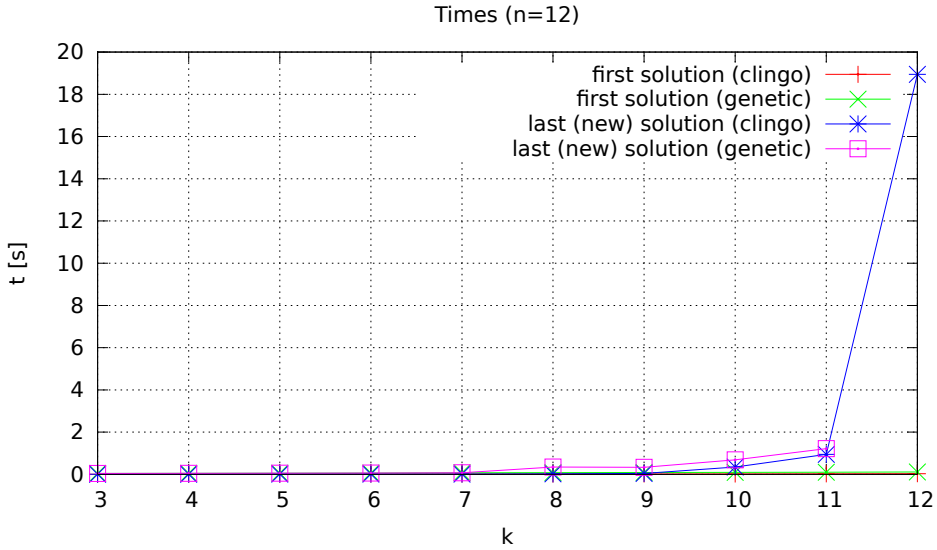

Algorithm compare n=8, k=12
Costs [min costs]
t [s]
clingo
genetic
0
0.5
1
1.5
2
2.5
0
2
4
6
8
10
12
14



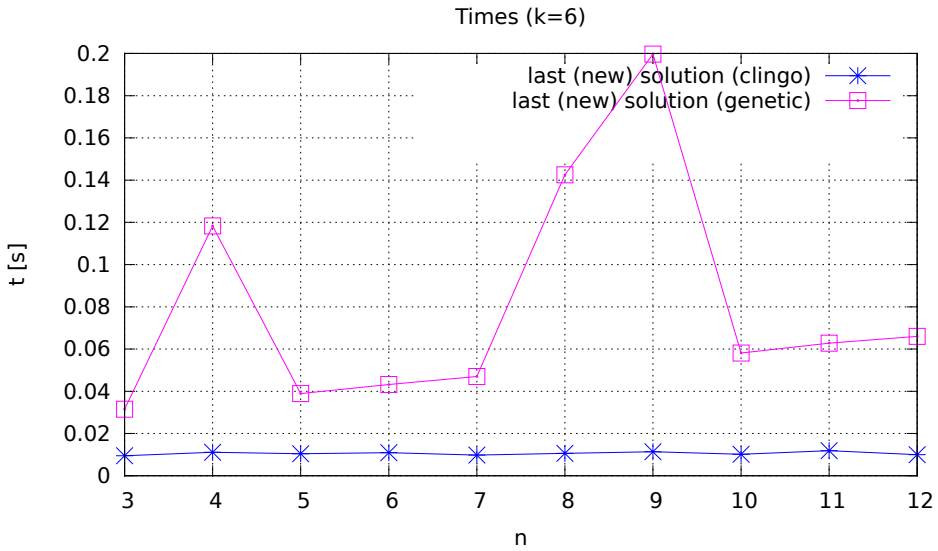
Es ist gut zu erkennen, dass die Algorithm mit einer relativ schlechten Lösung anfangen und sich dann sehr schnell der optimalen Lösung annähern. Die Laufzeit von Clingo hängt sehr stark von der Umfang des Problems ab, während der genetische Algorithmus deutlich konstanter in der Laufzeit ist.

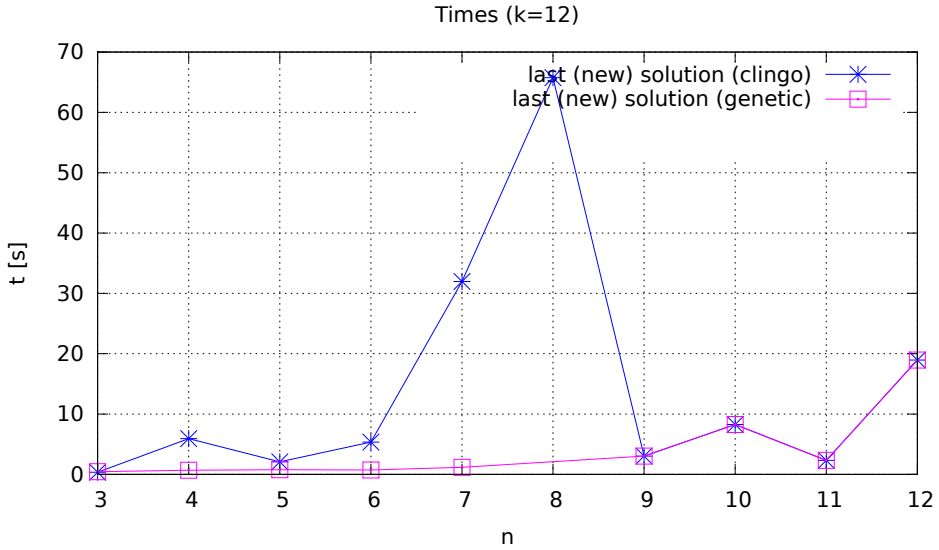
7.2 Abhängigkeit von k





7.3 Abhängigkeit von n




Times (k=12)
last (new) solution (clingo)
last (new) solution (genetic)
t [s]
n
0
10
20
30
40
50
60
70
3
4
5
6
7
8
9
10
11
12