

# 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

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	ShoppingTour . . . . .	3
1.2	Python 2.7 . . . . .	3
1.3	PyQt 4.9 . . . . .	3
1.4	Clingo (part of Potassco) . . . . .	4
<b>2</b>	<b>Ein NP-schweres Problem</b>	<b>4</b>
<b>3</b>	<b>Pre- und Postprocessing</b>	<b>5</b>
3.1	Preprocessing . . . . .	5
3.2	Implementierung des Preprocessing . . . . .	6
3.3	Postprocessing . . . . .	6
<b>4</b>	<b>Optimierungsalgorithmen</b>	<b>6</b>
4.1	ASP solver clingo . . . . .	8
4.2	Genetischer Algorithmus . . . . .	8

# 1 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* ( $\geq 10.4$ ) 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.

## 1.1 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.

## 1.2 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.

## 1.3 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 Mac-Ports).

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.

## 1.4 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 Problem-beschreibungen, darstellt. Vorkompilierte Binärdateien für Windows, Mac OS/Darwin und Linux sind von der Anbieterseite ebenso zu erhalten wie der Software-Quellcode.

## 2 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<sup>1</sup>. Wenn  $S = \{s_1, s_2, \dots\}$  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

---

<sup>1</sup>Einkaufsgeschäfte sind Städte.

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 Einkaufsplanungsproblems, deren Gesamtkosten unter dem Betrag  $n$  liegen? Eine mögliche Lösung kann durch Aufsummieren der Reisekosten zwischen den Einkaufsläden<sup>2</sup> und dem Addieren der Artikelpreise bei den entsprechenden Einkaufsläden in polynomieller Zeit<sup>3</sup> verifiziert werden. Somit handelt es sich um ein NP-schweres Problem, das selbst in NP liegt, also folglich um ein NP-vollständiges Problem.

## 3 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.

### 3.1 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<sup>4</sup> 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 anzusteuern Knoten. Welche Artikel bei welchem Geschäft gekauft werden, muss nicht gespeichert werden. Dazu geht man einfach alle Artikel durch und 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<sup>5</sup>. Durch eine Optimierung im Preprocessing kann der

---

<sup>2</sup>Die Reisekosten ergeben sich aus den Kantengewichten.

<sup>3</sup>Wie man sich leicht überlegen kann, macht es keinen Sinn, eine Kante mehr als zweimal zu benutzen.

<sup>4</sup> $\mathcal{O}(|V| + |E|)$ , wobei  $|V|$  die Anzahl der Knoten und  $|E|$  die Anzahl der Kanten ist.

<sup>5</sup>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.

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.

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

### 3.2 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<sup>7</sup>. Es wird sowohl die Länge als auch der eigentliche Weg gespeichert. Die Adjazenzmatrix des Graphen kann nun mit den berechneten Längen vervollständigt werden.

### 3.3 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.

## 4 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 Analy-

---

<sup>6</sup>Eine Rundreise ist natürlich auch ein Zyklus, aber dieser Zyklus zählt hier nicht.

<sup>7</sup>Laufzeit  $\mathcal{O}(|V|^3)$  bei  $|V|$  Geschäften.

sen verschiedene Algorithmen evaluiert. Um die Komplexität nicht unnötig zu steigern wollten wir uns auf maximal zwei Algorithmen beschränken. Um trotzdem ausreichend 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 ebenso viele Produkte) aufweisen und wir den Suchraum durch unser Pre-processing schon weit einschränken konnten, haben wir uns dazu entschieden auch die Möglichkeit zur Berechnung von optimalen Lösungen zu implementieren.

Nachdem wir verschiedene Metaheuristiken wie *Simulierte Abkühlung*, *Greedy-Suche* und *Tabusuche* untersucht haben, haben wir uns dazu entschieden einen *genetischen 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 Metaheuristiken 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<sup>8</sup>. 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 Straßennetz 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.

---

<sup>8</sup>In unseren Tests funktionierte der genetische Algorithmus aber recht gut.

## 4.1 ASP solver clingo

Clingo ist Programm, das ähnlich wie Prolog Logische Probleme löst. Zum 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.

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 booleschen Constraint-Lösens. Dabei werden besonders die sehr effizienten konfliktgetriebenen Nachbarschaftsuchen 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.

graph.lp

## 4.2 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 über-



führt 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 Individuum 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`.