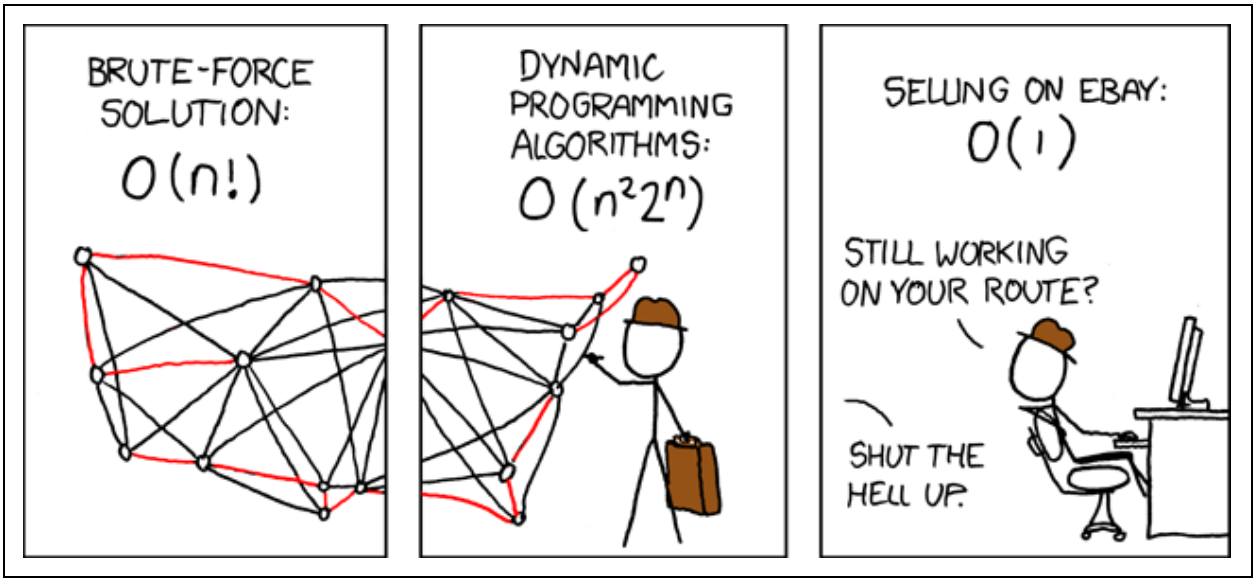
TWO TSP Project

Modul: Software Projekt 2

By Jérémie Blaser, Martin Eigenmann und Nicolas Roos

Zürcher Hochschule für Angewandte Wissenschaften, Zürich, Mai 2013



Inhaltsverzeichnis

Einleitung

Traveling Salesmen Problem

Das Traveling Salesmen Problem ist ein Gebiet der theoretischen Informatik. Die Herausforderung des Problems ist, den möglichst kürzesten Pfad durch eine endliche Anzahl von Städten zu finden. Das Traveling Salesmen Problem gehört zur Klasse der NP-äquivalenten Problemen. Das bedeutet, die Zeit, die für die Lösung des Problems benötigt wird, kann schon bei einer geringen Anzahl von Städten unpraktikabel lang werden.
1930 wurde das Traveling Salesmen Problem erstmals in der Mathematik erwähnt und viele Forscher beschäftigten sich seitdem damit. Heute stehen eine Vielzahl von heuristischen und exakten Methoden zur Verfügung, mit welchen auch komplizierte Fälle mit einigen tausend Städten optimal gelöst werden können.[1]

Eines der grössten Traveling Salesmen Problem ist das “World TSP”. Dabei muss man einen Pfad durch 1’904’711 Städte über den ganzen Globus verteilt finden. Dabei sind auch ein paar Forschungs Stationen in der Antarktis. Die momentane beste Route mit einer Länge von 7’515’778’188 wurde am 25.10.2011 von Keld Helsgaun mit einer Lin-Kernighan Heuristik gefunden.[2]

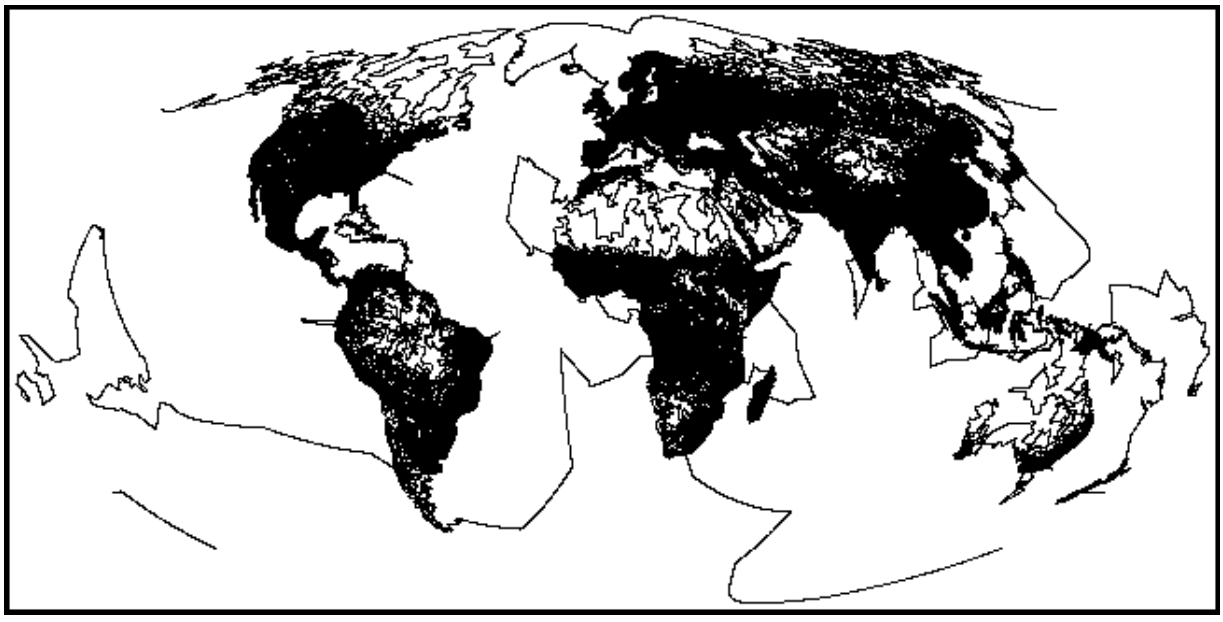


Abb 2: Pfad durch die fast 2 Millionen Städte des “World TSP”

[1] Quelle: Wikipedia.de - http://de.wikipedia.org/wiki/Traveling_Salesman_Problem
[2] World TSP Project - http://www.tsp.gatech.edu/world/

Traveling Santa Problem

Beim Traveling Santa Problem geht es darum zwei disjunkte Pfade durch die selben Städte zu finden. Die Idee kommt von einer Algorithmen-Wettbewerb-Website (www.kaggle.com) welche die Herausforderung für St. Nikolaus gestellt hat. Darin geht es um Nikolaus der nicht jedes Jahr um Weihnachten die selbe Route benutzen will, um seine Geschenke zu verteilen. Darum sollen zwei komplett unterschiedliche Pfade, d.h. ohne gemeinsame Kanten, berechnet werden.[3]
Obwohl die Herausforderung von einer imaginären Person stammt, kann dieses Problem auch in der Realität zur Anwendung kommen. Bei der Herstellung einer Computer Platine sollen für die Stromversorgung zwei disjunkte Pfade gefunden werden. Oder für eine riskante Situation kann es sehr nützlich sein, zwei disjunkte Routen zu haben.

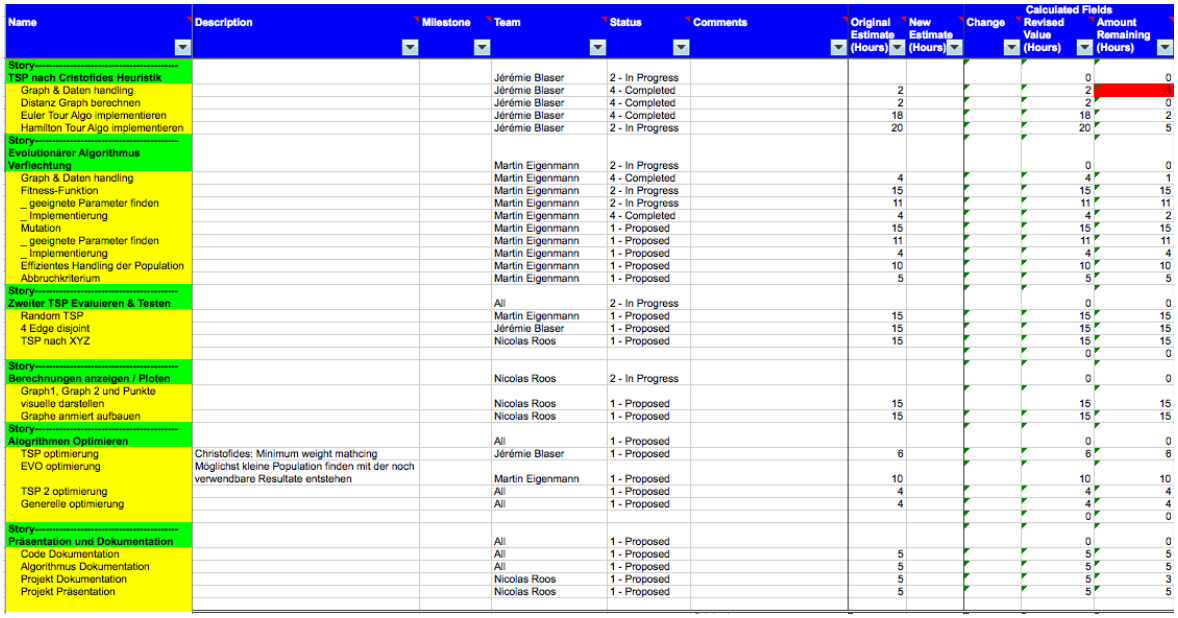
Herausforderung

Wir betrachten diese Problemstellung als ein sehr interessantes Thema und haben uns der Herausforderung gestellt ein Programm zu schreiben, welches durch eine gegebene Anzahl von Städten zwei möglichst optimale Pfade findet und danach, falls notwendig, alle doppelt benutzen Kanten auflöst.

[3] Quelle: kaggle.com - http://www.kaggle.com/c/traveling-santa-problem

Planung

Zu Beginn des Projekts wurde die Arbeit in Tasks aufgeteilt und auf die Team-Mitglieder verteilt. Die einzelnen Tasks wurden noch in etwa 10 Subtasks aufgeteilt und in ein Excel Backlog Sheet eingetragen.



Name	Description	Milestone	Team	Status	Comments	Original Estimate (Hours)	New Estimate (Hours)	Change	Calculated Fields: Revised Value (Hours)	Calculated Fields: Amount Remaining (Hours)
Story----------										
TSP nach Cristofides Heuristik			Jérémie Blaser	2 - In Progress					0	0
Graph & Daten handling			Jérémie Blaser	4 - Completed		2			2	0
Distanz Graph berechnen			Jérémie Blaser	4 - Completed		2			2	0
Euler Tour Algo implementieren			Jérémie Blaser	4 - Completed		18			18	2
Hamilton Tour Algo implementieren			Jérémie Blaser	2 - In Progress		20			20	5
Story----------										
Evolutionärer Algorithmus Verflechtung			Martin Eigenmann	2 - In Progress					0	0
Graph & Daten handling			Martin Eigenmann	4 - Completed		4			4	1
Fitness-Funktion			Martin Eigenmann	2 - In Progress		15			15	15
_ geeignete Parameter finden			Martin Eigenmann	2 - In Progress		11			11	11
_ Implementierung			Martin Eigenmann	4 - Completed		4			4	2
Mutation			Martin Eigenmann	1 - Proposed		15			15	15
_ geeignete Parameter finden			Martin Eigenmann	1 - Proposed		11			11	11
_ Implementierung			Martin Eigenmann	1 - Proposed		4			4	4
Effizientes Handling der Population			Martin Eigenmann	1 - Proposed		10			10	10
Abbruchkriterium			Martin Eigenmann	1 - Proposed		5			5	5
Story----------										
Zweiter TSP Evalulieren & Testen			All	2 - In Progress					0	0
Random TSP			Martin Eigenmann	1 - Proposed		15			15	15
4 Edge disjoint			Jérémie Blaser	1 - Proposed		15			15	15
TSP nach XYZ			Nicolas Roos	1 - Proposed		15			15	15
									0	0
Story----------										
Berechnungen anzeigen / Ploten			Nicolas Roos	2 - In Progress					0	0
Graph1, Graph 2 und Punkte visuelle darstellen			Nicolas Roos	1 - Proposed		15			15	15
Graphe anmiert aufbauen			Nicolas Roos	1 - Proposed		15			15	15
Story----------										
Alogrithmen Optimieren			All	1 - Proposed					0	0
TSP optimierung	Christofides: Minimum weight mathcing		Jérémie Blaser	1 - Proposed		6			6	6
EVO optimierung	Möglichst kleine Population finden mit der noch verwendbare Resultate entstehen		Martin Eigenmann	1 - Proposed		10			10	10
TSP 2 optimierung			All	1 - Proposed		4			4	4
Generelle optimierung			All	1 - Proposed		4			4	4
									0	0
Story----------										
Präsentation und Dokumentation			All	1 - Proposed					0	0
Code Dokumentation			All	1 - Proposed		5			5	5
Algorithmus Dokumentation			All	1 - Proposed		5			5	5
Projekt Dokumentation			Nicolas Roos	1 - Proposed		5			5	3
Projekt Präsentation			Nicolas Roos	1 - Proposed		5			5	5

Abb 3: Excel Backlog Sheet für das two TSP Projekt

Aufgrund unserer Teamgrösse und der Verteilbarkeit unserer Aufgaben konnten wir sehr unabhängig von einander Arbeiten. Wir definierten zu Beginn der Implementierung, woher wir die Punkte für die Städte nehmen und wie wir die einzelnen Pfade zurück oder übergeben. Danach arbeiteten wir komplett unabhängig von einander und informierten uns gegenseitig über grössere Probleme. Diese Unabhängige Arbeiten machte den Backlog Zeitplan für uns überflüssig. Nach dem erreichen der ersten Resultate ging es mehrheitlich nur noch ums optimieren und dieser Prozess lässt sich sehr schlecht planen. Verbleibende Aufgaben und Probleme wurden bei unserem Wöchentlichen Meetings besprochen und Erledigungs-Termine vereinbart.

Anforderungen

Bei der Planung und Entwicklung stand weniger ein absoluter perfekter Pfad als zwei möglichst gute und ähnlich lange Pfade im Vordergrund. Die Applikation sollte daher in kurzer Zeit zwei komplett disjunkte Pfade durch eine anschauliche Menge von Städten berechnen. Daher haben wir folgende Anforderungen vor der Planung und Implementierung definiert:

Städte: mindestens 1500

Pfade: 2 diskunkte Pfade

Zeit: unter 2 Minuten

Wo die jeweiligen Pfade starten spielt keine Rolle aber sie müssen alle Städte genau ein mal besuchen und wieder zum Start zurückkehren.

Tools

Wir haben uns sehr schnell auf die moderne Programmiersprache Python für dieses Projekt geeinigt. Python ist eine Programmiersprache welche von Anfang an konsequent objektorientiert entwickelt wurde[4] und sich bestens für die Lösung von mathematischen Problemen eignet. Die Sprache wurde zu beginn der 1990er Jahre von Guido van Rossum am Centrum Wiskunde & Informatica in Amsterdam entwickelt. Die Sprache wurde nach der berühmten englischen Komikertruppe *"Monthy Python"* benannt.[5] Alle in unserem Team haben schon mit der Programmiersprache Python Erfahrungen gemacht und Software damit entwickelt. Zur Sicherung und Versionskontrolle des Codes setzen wir Github ein.

Bibliotheken

Die Python Gemeinschaft hat eine unzählig grosse Sammlung von Open Source Bibliotheken entwickelt. Für die Lösung des Problems haben wir auch einige zusätzliche Bibliotheken benutzt.

inspyred: Wird eingesetzt im Rahmen des Evolutionären Algorithmus.

networkx: Hilft bei der Entwicklung, Manipulation und Erforschung von Graphen und komplexen Netzwerken.

matplotlib: Plottet 2D Graphen aller Art.

numpy: Bietet verschiedene Datenstrukturen und Hilfs-Funktionen für wissenschaftliche Berechnungen.

Datenstrukturen

Zur Abbildung von Pfäden verwendeten wir Listen aus Tupeln. Ein Tupel *(n1,n2)* stellt eine Kante von Knoten *n1* zu Knoten *n2* dar. Die Knoten bzw. Städte wurden als numerische Werte durchnummeriert.

Die Struktur zu Abbildung eines ungerichteten Graphen wurde von NetworkX vorgegeben. Das Framework verwendet intern Adjazenz-Listen, welche als Python Dictionaries implementiert sind.

[4] Quelle: Pythonmania.de - http://www.pythonmania.de/article/warum.html

[5] Quelle: Wikipedia.org - http://de.wikipedia.org/wiki/Python_(Programmiersprache)

Ansatz

Vor der Entwicklung der Applikation haben wir uns mit dem Traveling Salesmen Problem und den aktuellen Lösungsansätzen auseinandergesetzt. Es erschien sinnvoll zuerst einen unabhängigen Pfad mit einer Heuristik zu berechnen. In einem zweiten Schritt wird mit geringerem Aufwand ein zweiter Pfad berechnet. Diese beiden Pfade sollen dann in einem dritten Algorithmus von ihren doppelt benutzen Kanten befreit werden.
Wir entschieden uns für die Berechnung des ersten Pfad die Christofides-Heuristik einzusetzen, welche zwar eine Laufzeitkomplexität von $O(n^3)$ aufweist aber maximal nur um den Faktor 1.5 vom exakten Optimum abweicht.[6]
Zur Konstruktion des zweiten Pfades standen diverse vom ersten Pfad abhängige, teilweise abhängige und vollkommen unabhängige Algorithmen und Ideen zur Diskussion. Wir entschieden uns für einen teilweise abhängigen evolutionären Algorithmus. Bei diesem evolutionären Algorithmus verwendeten wir den Ameisen Kolonie optimierungs Algorithmus.
Obwohl dieser Ansatz für uns sehr vielversprechend aussah war uns klar dass noch weitere Ansätze geprüft werden müssen.

Weitere Ansätze zur Berechnung des zweiten Pfades:

- Anderer evolutionärer Algorithmus
- Andere Heuristik
- 3 OPT Algorithmus

Da es nicht genügt nur einen zweiten kurzen Pfad zu finden, sondern dieser zweite Pfad auch disjunkt sein musste, haben wir auch zur Lösung dieses Problems zwei mögliche Vorgehensweise in Betracht gezogen.

Vorgehensweise zur Lösung von doppelt benutzten Kanten:

- Evolutionärer Ansatz
- Iterativer Ansatz

[6] Quelle: Wikipedia.org - http://de.wikipedia.org/wiki/Christofides-Heuristik

Algorithmen

In diesem Kapitel werden die einzelnen Algorithmen und deren Funktionsweise beschrieben.

Christofides Heuristik

Die 1976 publizierte und nach seinem Erfinder, Nicos Christofides, benante Heuristik ist ein Algorithmus zur Approximation des TSP-Problems.

Bezeichnung: che

Funktionsweise:

1. Erzeuge einen MST[7] *B* für den zugrunde liegenden Graphen.
2. Suche ein (bezüglich Kantengewicht) minimales perfektes Matching von den Knoten aus *B*, welche ungeraden Grad haben.
3. Füge diese Kanten zu *B* hinzu. Der entstehende Graph *G* ist dann Eulersch.
4. Konstruiere eine Eulertour in *G*.
5. Konstruiere einen Hamiltonkreis in *G*. Wähle dazu einen beliebigen Startknoten und gehe die Eulertour ab. Ersetze dabei die bereits besuchten Knoten durch direkte Verbindungen (bzw. Abkürzungen) zum nächsten noch nicht besuchten Knoten.

Laufzeit: $O(n^3)$

Hauptverantwortlich für die hohe Laufzeit ist der 2. Schritt im Algorithmus.

Qualität: Die entstandene Tour ist höchstens 1.5 mal so Lang wie die optimale Lösung.

Nacharbeiten: Da der CHE-Algorithmus unseren Referenz-Pfad generiert wird dieser nur bei Duplikaten des Nachfolge-Algorithmus mit dem ME-Algorithmus von Duplikaten befreit.

[7] Quelle: Wikipedia.org - http://de.wikipedia.org/wiki/Minimal_spannender_Baum

Evolutionärer Algorithmus

Random Evolution Optimierung

Beim REO handelt es sich um eine performante jedoch unzuverlässigere Variante eines Evolutionären Algorithmus.

Bezeichnung: evo

Funktionsweise: Es werden für jede Evolutions-Stufe (E_k), bestehend aus M Einheiten, die Besten N Einheiten der vorhergehenden Evolutions-Stufe ausgewählt. Dabei gilt M > N und für E_0 werden alle Einheiten neu generiert. Die noch zu generierenden Einheiten werden mittels eines Zufalls-Algorithmus generiert. Nach einer fixen Evolutions-Stufe E_{max} wird die beste Einheit als Lösung definiert. Jede Einheit besteht aus F Kanten.

Variabilität: Eingabegrösse ist F. Weitere Grössen sind E_{max} , M, N

Laufzeit: $O(E_{max} * (M-N)*F+N*F) = O(n)$

Stabilität: Der Algorithmus terminiert zuverlässig nach vorhersehbar Zeit.

Qualität: Die Resultate sind weder vorhersehbar noch reproduzierbar.

Nacharbeit: Da kein Duplikate-freier Weg garantiert werden kann wird der EVO-Algorithmus in einem Nachgang mit dem ME-Algorithmus von Duplikaten befreit.

Konkrete Implementierung:

```
# Generation eines einzelnen Kandidaten
def generator(self, random, args):
    """Return a candidate solution for an evolutionary computation."""
    locations = [i for i in range(len(self.weights))]
    random.shuffle(locations)
    return locations
```

Abb 4: Code Beispiel des Random Evolution Optimierungs Algorithmus

Ameisen-Kolonie Optimierung

Beim ACO handelt es sich um eine zuverlässige und einer der besten Approximationen eines Optimierungsproblems, jedoch auch um eine sehr unperformante Variante eines Evolutionären Algorithmus. Der Algorithmus beruht auf der Idee der Schwarmintelligenz und gehört deshalb in die Gruppe der heuristischen Algorithmen[8].

Bezeichnung: aco

Funktionsweise: Es werden für jede Evolutions-Stufe (E_k), bestehend aus M Einheiten, die Besten N Einheiten der vorhergehenden Evolutions-Stufe ausgewählt. Dabei gilt M > N und für E_0 werden alle Einheiten neu generiert. Die noch zu generierenden Einheiten werden mittels eines Heuristik-Algorithmus (A) generiert. Nach einer fixen Evolutions-Stufe E_{max} wird die beste Einheit als Lösung definiert. Jede Einheit besteht aus F Kanten.

Variabilität: Eingabegrösse ist F. Weitere Grössen sind E_{max} , M, N

Laufzeit: $O(E_{max} * (M-N)*A(F)^4 + N*A(F)^4) = O(F^4)$

Stabilität: Der Algorithmus terminiert zuverlässig nach vorhersehbar Zeit.

Qualität: Die Resultate sind sehr gut jedoch weder vorhersehbar noch reproduzierbar.

Nacharbeit: Da kein Duplikate-freier Weg garantiert werden kann wird der ACO-Algorithmus in einem Nachgang mit dem ME-Algorithmus von Duplikaten befreit.

Konkrete Implementation:

```
# Generation eines einzelnen Kandidaten
def constructor(self, random, args):
    self._use_ants = True
    """Return a candidate solution for an ant colony optimization."""
    candidate = []
    while len(candidate) < len(self.weights) - 1:
        # Find feasible components
        feasible_components = []
        if len(candidate) == 0:
            feasible_components = self.components
        elif len(candidate) == len(self.weights) - 1:
            first = candidate[0]
            last = candidate[-1]
            feasible_components = [c for c in self.components
                                   if c.element[0] == last.element[1] and c.element[1] == first.element[0]]
        else:
            last = candidate[-1]
            already_visited = [c.element[0] for c in candidate]
            already_visited.extend([c.element[1] for c in candidate])
            already_visited = set(already_visited)
            feasible_components = [c for c in self.components
                                   if c.element[0] == last.element[1] and c.element[1] not in already_visited]
        if len(feasible_components) == 0:
            candidate = []
        else:
            # Choose a feasible component
            if random.random() <= self.bias:
                next_component = max(feasible_components)
            else:
                next_component = selectors.fitness_proportionate_selection(random,
                                                           feasible_components, {'num_selected': 1})[0]
            candidate.append(next_component)
    return candidate
```

Abb 5: Code Beispiel des Ameisen-Kolonie Optimierung Algorithmus

[8] Quelle: Wikipedia.org - http://de.wikipedia.org/wiki/Ameisenalgorithmus

Mathematischer Algorithmus

ME Algorithmus

Beim ME Algorithmus handelt es sich um einen geometrischen Algorithmus zur Auflösung von doppelt benutzen Kanten. Dieser Algorithmus wurde im Verlauf des Projektes entwickelt. Es sind uns keine vergleichbaren Lösungsansätze bekannt.

Bezeichnung: me

Funktionsweise: Für die beiden Pfade G_1 und G_2 werden alle doppelt benutzen Kanten gesucht. Es wird für jedes Duplikat, bestehend aus der Verbindung von P_1 zu P_2, eine Ausweich-Rute, über den geometrisch nahesten Punkt P_3, zu P_1 gelegt. Somit führt der Weg von P_1 über P_3 zu P_2. Der Punkt P_3 muss im späteren Verlauf des Pfades nicht mehr überquert werden und wird entfernt. Dadurch können neue Duplikate entstehen und der Algorithmus startet mit dem ersten Schritt.

Variabilität: Eingabegrösse ist G_1 und G_2.

Laufzeit: O(n) falls alle Duplikate in einem Schritt gelöst werden.

Stabilität: Der Algorithmus terminiert nicht zuverlässig für Duplikate > 15 und Punkte < 50.

Qualität: Für das Lösen einzelner Duplikate liefert der Algorithmus gute und reproduzierbar Lösungen.

Nacharbeit: keine

Konkrete Implementation:

```
# Alle Duplikate finden und lösen
# self.hashmap beinhaltet referenzen zu allen Duplikate => Zugriff und Erkennund der Duplikate in O(n)
while(len(self.hashmap)> 0 and self.hashmapOld != self.hashmap and self.hashmapOldOld != self.hashmapOld):
    # self.nearestpoints beinhaltet eine Liste der geometrisch nahesten Punkte
    self.nearespoints = dict()
    # Falls route0 kürzer als route1 ist, werden alle Duplikate in route0 gelöst
    if self.calc_path_lenght(self.route0) < self.calc_path_lenght(self.route1):
        for mapentry in self.hashmap:
            print('solving {0}'.format(mapentry))
            self.route0 = self.solve_duplicate(self.route0,self.hashmap.get(mapentry)[0])
    else:
        for mapentry in self.hashmap:
            print('solving {0}'.format(mapentry))
            self.route1 = self.solve_duplicate(self.route1,self.hashmap.get(mapentry)[1])
    # Erkennung von Endlos-Schleifen
    self.hashmapOldOld = self.hashmapOld
    self.hashmapOld = self.hashmap
    # ev. neu entstandene Duplikate finden
    self.build_hasmap()
```

Abb 6: Code Beispiel des ME Algorithmus

NJM Algorithmus

Beim NJM Algorithmus handelt es sich um einen Geometrischen Algorithmus zur Generierung eines Duplikate-freien zweiten Pfades in Abhängigkeit eines gegebenen Pfades. Dieser Algorithmus wurde im Verlauf des Projektes entwickelt. Es sind uns keine vergleichbaren Lösungsansätze bekannt.

Bezeichnung: njm

Funktionsweise: Der Vorgabepfad G_1 wird in Partitionen O_k^1 zu je fünf Kanten aufgeteilt. Für jede dieser Teilpfade O_k^1 wird nun anhand der Punkte $P_0,P_1,...,P_5$ ein neuer Teilpfad O_k^2 folgender Form erstellt: $O_k^2 = P_0,P_3,P_1,P_4,P_2,P_5$. Der Ausgabepfad ergibt sich aus der Konkatenation aller Teilpfade O_k^2.

Variabilität: Eingabegrösse ist G_1.

Laufzeit: O(n)

Stabilität: Der Algorithmus terminiert zuverlässig nach vorhersehbar Zeit.

Qualität: Der Algorithmus liefert für beliebige Vorgabepfade G_1 einen zweiten Douplikate-freien Pfad G_2 mit $3*G_1 => G_2$.

Nacharbeit: keine

Theorie

Für eine Partition mit dem Pfad G_k^1 wird ein alternativer Pfad G_k^2 gefunden der maximal um dem Faktor 3 grösser als G_k^1 ist und alle Punkte P_k in der Partition O_k^1 erreicht. Der Maximale Faktor kann anhand des ungünstigsten Fall nachgewiesen werden.

Annahme: Die Punkte $P_0,P_1,...,P_5$ befinden sich in selbem Abstand zueinander auf einer Gerade und der Vorgabepfad erreicht die Punkte in aufsteigender Reihenfolge und erreicht somit die Länge 5.

Beweis: Da die Annahme vom ungünstigsten Fall ausgeht ist der hier gezeigte Weg für G_k^2 maximal. Wenn nun auf diesen Teilpfad der NJM-Algorithmus angewendet wird ergibt sich ein Pfad mit der Länge 15.

Das Verhältnis zwischen G_k^1 und G_k^2 ist in diesem Fall 5:15.

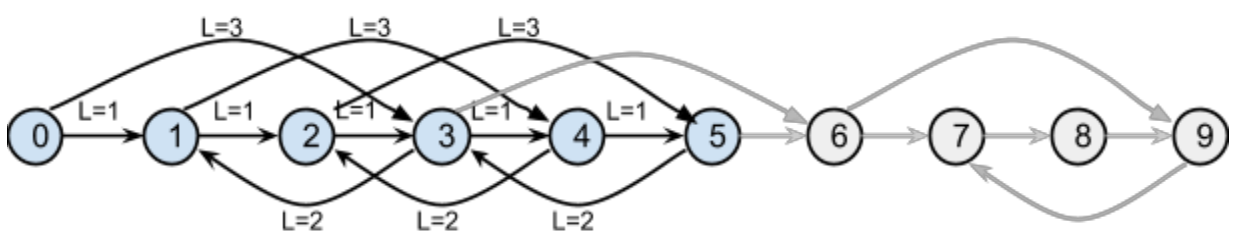


Abb 7: Grafische Darstellung zum Beweis der maximale NJM Pfadlänge

Weiteres: Da jeder TSP-Pfad eine geschlossene Tour bilden muss wird ein Verhältnis von um die 1:2 in einem realen Fall zustande kommen.

Konkrete Implementation:

```
# Einen alternativen Pfad zu route0 finden
# Partitionierung mit genau 5 Kanten
while inx < len(self.route0)-3 and inx != len(self.route0)-5:
    # 3 Kanten nach vorne
    inx += 3
    pt1 = self.route0[inx][0]
    self.route1.append( (pt0,pt1) )
    pt0 = pt1

    # Neustart der Partitionierung
    if counter < 2:
        counter += 1
        # 2 Kanten zurück
        inx -= 2
        pt1 = self.route0[inx][0]
        self.route1.append( (pt0,pt1) )
        pt0 = pt1
    else:
        counter = 0

# Lösen der letzen verbleibenden Punkte
```

Abb 8: Code Beispiel des NJM Algorithmus

Implementierung

Zuerst wurde die Christofides Heuristik für den fast optimalen Pfad implementiert. Gleichzeitig wurde der evolutionäre Algorithmus für den zweiten Pfad entwickelt. Die Christofides Heuristik zeigte von Beginn an gute Ergebnisse. Unsere Implementierung der Heuristik generierte zu Beginn bei mehr als 1000 Städte Fehler, die wir aber schnell korrigieren konnten. Bei der Implementierung von Schritt 4 der Christofides Heuristik (Eulertour generieren), haben wir zuerst auf eine Methode des NetworkX-Frameworks zurückgegriffen. Diese lief jedoch äusserst langsam, da diese, wie sich beim inspizierent des Quellcodes herausstellte, den Algorithmus von Fleury verwendete, welcher eine Laufzeit von $O(|E|^2)$ aufweist. Als Optimierung haben wir dann den Algorithmus von Hierholzer[9] implementiert, der das selbe Problem in linearer Zeit löst. Somit blieb nur noch der 2. Schritt als Flaschenhals.

Der evolutionäre Algorithmus bereitet erheblich mehr Probleme aber lieferte schlussendlich schneller als die Implementation der Christofides Heuristik einen zweiten Pfad.

Nachdem wir in der Lage waren, zwei gute Pfade zu finden, mussten wir die noch vorhandenen Duplikate ausmerzen. Dazu entwickelten wir einen Deduplizierungs Algorithmus (ME-Algorithmus) welcher beide Pfade als Eingabe nimmt und beide Pfade iterativ solange bearbeitet bis alle Duplikate eliminiert wurden.

Nach ausgiebigen Tests eines linearen Ansatzes machten wir zwei Feststellungen. 1. Durch das Lösen eines Duplikats können weitere Duplikate erstellt werden. 2. Um dies zu verhindern muss jede mögliche Lösung eines Duplikates duchprobiert werden. Daraus folgte dass wir den Evolutionären Ansatz nicht mehr weiter verfolgen.

Nach der Fertigstellung der Grundkomponenten begann die Optimierungs-Phase. Sehr viel Zeit wurde in die Optimierung des evolutionären Algorithmus für den zweiten Pfad gesteckt, welcher aber nie eine Lösung, die besser als Faktor 10 gegenüber der Länge des Pfades der Heuristik war, fand. Nach unzähligen Anpassungen am evolutionären Algorithmus ohne wirklichen Erfolg, haben wir uns zusammengesetzt und versucht eine bessere Lösung zu finden. Die eleganteste und mathematisch sicherste Lösung entwickelten und implementierten wir in Form des NJM-Algorithmus. Die Vorteile des NJM-Algorithmus liegen in der linearen Laufzeitkomplexität und in der Duplikate-freien Lösung.

[9] Quelle: Wikipedia.org - http://de.wikipedia.org/wiki/Algorithmus_von_Hierholzer

Resultate

Unser Ziel einen Algorithmus zu implementieren, der zwei disjunkte möglichs kurze Rundreise-Pfade in absehbarer Zeit über 1500 Städte findet, haben wir bei weitem übertroffen. Nachfolgend die beiden Pfade über 15'000 und 1500 Städte, welche in unter 76 Minuten bzw. in unter einer Minute generiert wurden.

Zu beachten gilt dass beide Pfade maximal um den Faktor 1.5 bzw. 2.97 vom maximal kürzesten Pfad entfernt sind und somit auch das noch verbleibende Optimierungs-Potenzial vergleichsweise mager ausfällt.

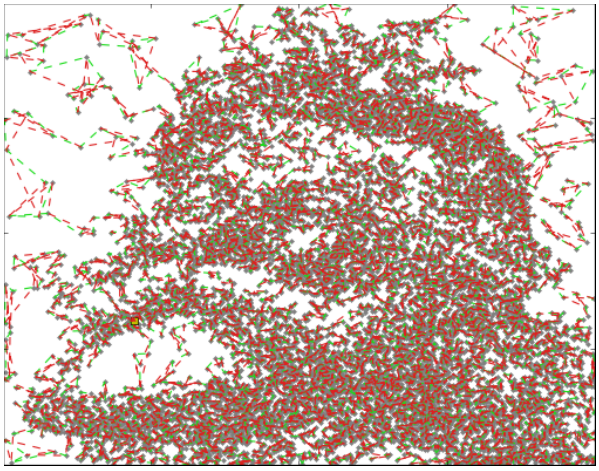


Abb 9: Plot bei 15'000 Städte

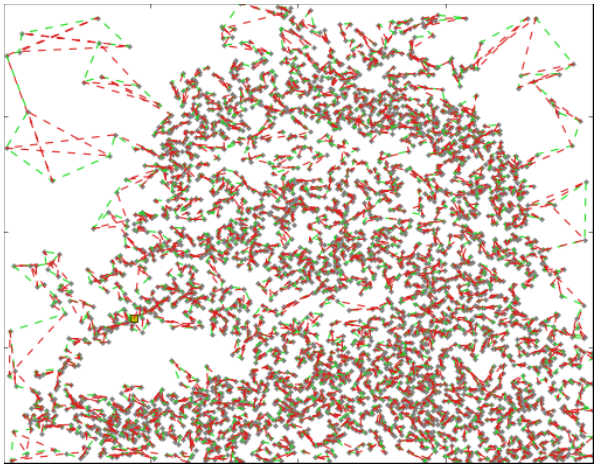


Abb 10: Plot bei 1500 Städte

Performance

Nach der Fertigstellung aller Algorithmen und der Überprüfung auf deren Korrektheit waren wir in der Lage einen aussagekräftigen Vergleich anzufertigen.
In der nachstehenden Grafik werden die Anzahl Städte und die Pfadlänge in Relation gesetzt. Der CHE- und der NJM-Algorithmus (in Abhängigkeit vom CHE-Algorithmus) besitzen eine lineare Abhängigkeit von der Eingabegrösse. Der Evo-Algorithmus besitzt einen überquadratische Abhängigkeit der Eingabegrösse und über die beiden verbleibenden Lösungsansätze kann keine derartige Aussage getätigt werden.

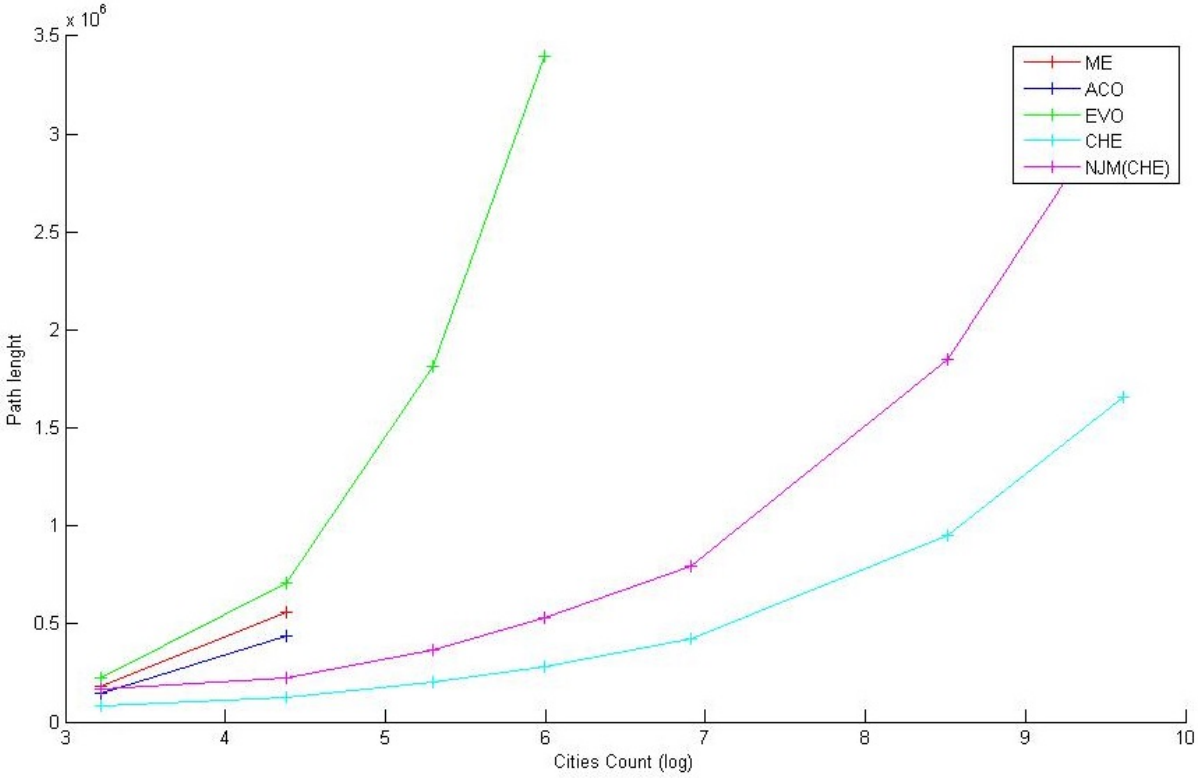


Abb 11: Vergleich Resultate der verschiedenen Algorithmen
Hinweis: Die X-Aches ist logarythmisiert.

Detail Daten Performance-Test

- vs che: Faktor gegenüber dem veränderten CHE-Pfad
- Loading...che: Faktor der Änderung des CHE-Pfad wegen der Nacharbeit

Städte	Algorithmus	Pfadlänge	vs che	Loading che	Laufzeit	Duplikate
25	me	183835	1.47	1.58	<1s	25
	evo	221190	2.59	1.08	~1s	~5
	aco	142784	0.93	1.95	~2s	~6
	njm	169182	2.14	1.00	<1s	0
	che	78884.86	-	-	<1s	-
80	me	558096	2.82	4.37	3s	80
	evo	708230	5.54	1.02	~4s	~10
	acs	435391	1.00	3.40	~45s	~30
	njm	227130	1.78	1.00	<1s	0
	che	127798	-	-	<1s	-
400	evo	3395832	11.51	1.05	32s	~20
	njm	531793	1.90	1.00	2s	0
	che	279769	-	-	4s	-
1000	njm	793894	1.87	1.00	5s	0
	che	423456	-	-	20s	-
5000	njm	1847309	1.94	1.00	25s	0
	che	952955	-	-	10m 31s	-
15’000	njm	3207936	1.94	1.00	1m 15s	0
	che	3207936	-	-	73m 58s	-

Grafische Darstellung

Nachfolgend finden sind Plots der verschiedenen Lösungsansätze mit 80 Städten.

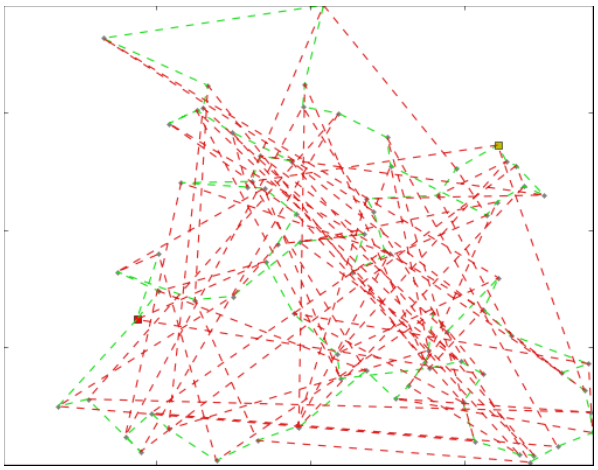


Abb 12: Plot bei 80 Städte mit EVO

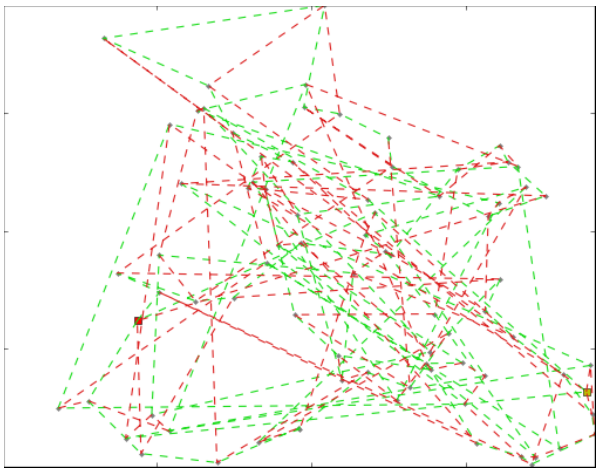


Abb 13: Plot bei 80 Städte ACO

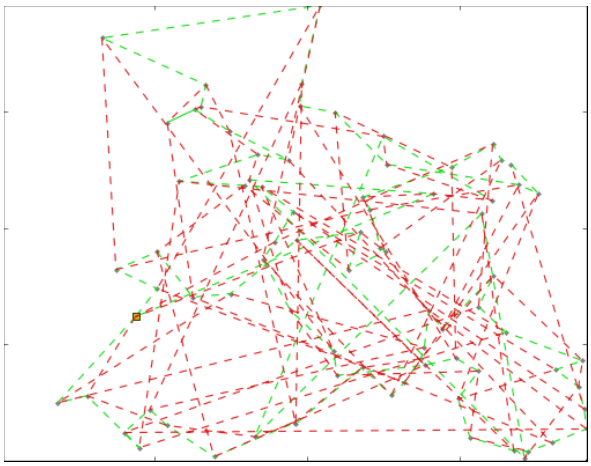


Abb 14: Plot bei 80 Städte ME

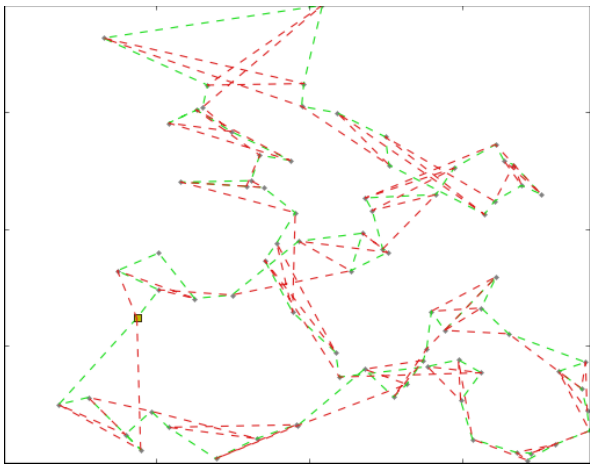


Abb 15: Plot bei 80 Städte NJM

Zukunft

Wir haben uns im Bestreben um eine noch bessere Lösung und zunehmende Divergenz der beiden Pfade gedanken zur Erreichung dieser Ziele gemacht. Um die beiden Pfade des NJM-Algorithmus und der Christofides Heuristik noch weiter zu verkürzen, bieten sich folgende zwei Lösungsansätze an:

Dynamische Partitionierung entlang des Pfades

Wenn die beiden Pfade so partitioniert werden dass pro Partition beide Pfade mit der gleichen Anzahl an Kanten die gleichen Punkte erreichen, können in diesem abgeschlossenen Teilstück beliebige Optimierungs-Algorithmen angewendet werden um die beiden Pfade noch weiter zu verkürzen.
Es wird immer genau dann ein abgeschlossenes Teilstück der beiden Pfade partitioniert, wenn ein Vielfaches von fünf Kanten enthalten ist.

Fixierte geometrische Partitionierung

Um die Pfade von dem vorgegebenen Weg wegführen zu können, wird eine geometrische Partitionierung des gesamten Lösungsraums erstellt und darin mittels eines beliebigen Optimierungs-Algorithmus eine kürzere Lösung gesucht. Zu beachten ist hier, dass der Pfad weiterhin durchgängig bleibt und nicht mehrere getrennte Pfäde entstehen.

Weitere Gedanken

Da wir nach Abschluss der Entwicklung nun zwei sehr ähnliche Pfade finden können, stossen wir mit mutativen Optimierungs-Algorithmen an Grenzen. Deshalb ist es sinnvoll an dieser Stelle auch die Implementierung einer weiteren Heuristik zur Suche eines unabhängigen Pfades zu evaluieren. Aber auch hier gilt es dann das Duplikate-Problem zu lösen.

Fazit

Mit der Christofides Heuristik und dem NJM Algorithmus können wir in ca. 75 Minuten für 15’000 Städte 2 Pfade berechnen welche nie dieselbe Kante benutzen. Das sind 10 mal mehr Städte als wir ursprünglich geplant haben und die beiden Pfade unterscheiden sich nur um den Faktor ~2 statt 10. Diese Resultate übertreffen unsere Vorstellungen, die wir zu Beginn hatten, bei weitem. Durch die Implementierung der einzelnen Aufgaben haben wir viel über die Umsetzung solcher Algorithmen gelernt. Das wichtigste was wir jedoch durch dieses Projekt gelernt haben, ist das manchmal ein ganz simpler und eleganter Ansatz wie der NJM Algorithmus viel bessere Resultate liefert als eine komplexe mathematische Berechnung.

Abbildungsverzeichnis