Traveling Salesman Problem

1 Problemstellung

Wer kennt es nicht? Ob beim Zeitungen austragen, Einkaufen gehen oder Farmen von irgendwelchen Items in dem Lieblingsspiel. Häufig müssen wir uns überlegen, wie wir am schnellsten von A nach B kommen - und dabei noch über C, D und E gehen. Viele Wege führen nach Rom, so heißt es, aber von Gießen nach Italien erst über München und dann über Berlin zu fahren klingt für uns sehr unsinnig. Mit etwas Überlegung finden wir relativ schnell einen guten Weg bzw. einen guten *Rundkurs*, welchen wir ohne zu viel Sprit und Zeit zu verbrauchen nehmen können.

Dieses Problem wurde schon früh im 19. Jahrhundert beschrieben und trägt daher auch seinen Namen. Damalige Händler wollten ihre Waren in möglichst vielen Städten zum Verkauf anbieten, ohne dabei zu viel Zeit auf der Reise zu verschwenden. Ca. 100 Jahre später konnte diese Aufgabe auch mathematisch als *Graphenproblem* beschrieben werden. Man kann sich hierzu alle Zielorte als *Knoten* vorstellen und die Wege dazwischen als *Kanten*.

Das dieses Problem nicht trivial ist, zeigt sich bei näherer Betrachtung: Um drei Punkte $\{A, B, C\}$ als Kreis zu verbinden gibt es zwei Möglichkeiten $[A, B, C]$ und $[A, C, B]$. Alle weiteren Kombinationsmöglichkeiten wie z.B. $[C, B, A]$ sind (im Kreis) identisch mit einer der beiden Lösungen. Bei vier Punkten $\{A, B, C, D\}$ sind es bereits sechs Möglichkeiten $[A, B, C, D]$, $[A, B, D, C]$, $[A, C, B, D]$, $[A, C, D, B]$, $[A, D, B, C]$ und $[A, D, C, B]$. Bei fünf Punkten werden es entsprechend 24 Kombinationen und bei sechs schon 120.

1.1 Beschreibung

Eine Lösung für das Problem des Handlungsreisenden (TSP) ist also ein Weg, welcher alle Punkte miteinander verbindet. Dieser Pfad muss zyklisch sein, also dort aufhören, wo er begonnen hat. Zudem darf kein Ort zweimal besucht werden. Unter der Annahme, dass wir von jedem Ort direkt zu jedem anderen Ort kommen können (also alle denkbaren Straßen vorhanden sind), kann die Anzahl der Lösungen mit (n-1)! angegeben werden.

Die *Fakultät* hat nur die problematische Eigenschaft, dass sie enorm schnell wächst. Während 5! noch „nur“ 120 sind, ist 8! schon größer als 40.000 und 12! etwa eine halbe Million. 20! ist 2.432.902.008.176.640.000 und somit schon viel zu groß um einfach mal ausprobiert werden zu können. Denn unter der Annahme, dass ein moderner Prozessor mit 3 GHz und 8 Kernen somit 24.000.000.000 Rechenoperationen pro Sekunde durchführen kann und (sehr stark vereinfacht) eine Kombination nur einen einzige Operation bräuchte, müsste man trotzdem über drei Jahre auf eine Lösung warten. Bei 22 Orten also bereits 66 Jahre usw.

1.2 Anwendbarkeit

Unter teilweise anderen Bedingungen, wie z.B. asymmetrischen Wegen ist das TSP weitverbreitet. Ob in der Elektrotechnik zur optimalen Anordnung von Bauteilen auf einer Leiterplatte oder bei Online-Händlern zur schnellstmöglichen Kommission von allen Produkten einer Bestellung. Daher ist eine gute Lösung schwer zu finden und enorm wichtig um in Unternehmen wirtschaftlich arbeiten zu können.

2 Lösungsansätze

Um bei mittelschweren Problemen nicht bis ans Ende seiner Tage auf eine Lösung warten zu müssen, hat man natürlich auch versucht schneller zum Ziel zu kommen. Diese Verfahren nennt man *Heuristiken*. Das Ziel eines heuristischen Verfahrens ist es, in deutlich geringerer Zeit zu einer annähernd „guten“ Lösung zu kommen. Aufgrund der Komplexität des Problems kann man eben nicht nachweisen, dass es sich dabei um die beste Lösung handelt. Daher gilt das Motto: Lieber eine gute Lösung, als gar keine.

2.1 Permutation - Die beste Lösung

Für kleine Problemfälle lässt sich natürlich weiterhin „die beste“ Lösung finden. Erfahrungsgemäß bis zu 12 Punkte lassen sich *permutativ* zu ca. 40.000 Lösungen umwandeln um daraus den kürzesten Weg zu finden. Diese Holzhammermethode nennt sich aufgrund dieser Eigenschaft auch *Brute-Force*.

1. Bestimme einen beliebigen Startpunkt
2. Bestimme alle weiteren Nachfolger per Rekursion
3. Rekursion:
   3.1. Wenn kein Punkt mehr verbleibt: Eine Lösung gefunden
   3.2. Ansonsten iteriere über alle verbleibenden Punkte
      3.2.1. Nutze den Punkt als Nächsten
      3.2.2. Bestimme alle weiteren Nachfolger per Rekursion

Vorteil: Das ist die einzige Variante um sicher das Optimum zu bestimmen.
Nachteil: Schon bei 10 Punkten steigt die Laufzeit deutlich an, mehr als 12 ist nicht machbar.
Schwierigkeitsgrad: Leicht

2.2 Heuristik - Eine gute Lösung

Deutlich schneller ist das, was jede/-r von uns intuitiv auch hinbekommt: Gehe von einem Punkt zu dem nächstgelegenen, der noch nicht besucht wurde. Diese *Nearest-Neighbour-Heuristik* geht relativ fix und kommt zu einer guten Lösung. Einzelne Ausreißer werden jedoch häufig ungünstig mit dem Rest verbunden.

Praktisch funktioniert diese Umsetzung genauso leicht wie beschrieben. Von einem beliebigen Startpunkt wird in der Menge der verbleibenden Punkte der nächstgelegene gesucht. Dieser dient im nächsten Schritt ebenfalls als Startpunkt. Da es hier in jedem Schritt nur einen nächsten Punkt (und nicht viele wie bei der Permutation) gibt, ist das Verfahren nicht rekursiv, sondern *iterativ*.

Dieses Verfahren kann auch optimiert werden. Da das Ergebnis von der Wahl des Startpunkts abhängt, kann man nacheinander bei jedem Punkt starten und sich den kürzesten Weg merken.

Vorteil: Sehr schnelle Lösung.
Nachteil: Einzelne Verbindungen sind relativ lang.
Schwierigkeitsgrad: Sehr Leicht

2.3 Selbst organisierende Karten - Eine intelligente Lösung

Selbst organisierende Karten sind ein Produkt der KI-Forschung des Finnen Teuvo *Kohonen*. Vorlage der Idee sind die Abläufe im Hirn, dessen Neuronen sich auch beständig auf neue Umstände einstellen müssen. Bildlich kann man sich diesen Algorithmus wie einen Gummiring vorstellen, der langsam auseinander gezogen wird. Auf einem Brett mit Nägeln an den Punkten wird dieser Ring über alle Nägel gelegt und bildet so eine *Hülle* um alle Punkte herum.

Während die eigentlichen Städte auf der Karte unveränderliche Fixpunkte sind, braucht man für diesen Ansatz auch bewegliche *Wegpunkte*. Zunächst wird ein Wegpunkt auf der Kartenmitte platziert, das anschließende Verfahren wiederholt sich dann bis zum Erreichen eines Abbruchkriteriums.

1. Iteriere über alle Städte
   1.1. Suche den Wegpunkt, der der Stadt am nächsten liegt
   1.2. Bewege diesen Wegpunkt in Richtung der Stadt
   1.3. Bewege die Nachbarn des Wegpunktes (etwas weniger stark) in Richtung der Stadt
2. Iteriere über alle Wegpunkte
   2.1. Prüfe, von wie vielen Städten ein Wegpunkt direkt angezogen wurde
   2.2. Falls niemand ihn direkt anzog, ziehe dem Wegpunkt ein Leben ab
   2.3. Falls ihn eine Stadt direkt anzog, setze die Leben auf Maximum
   2.4. Falls ihn mehrere Städte direkt anzogen, setze die Leben auf Maximum und klone ihn an gleicher Stelle in den Ring
   2.5. Falls ein Wegpunkt kein Leben mehr hat, entferne ihn aus dem Ring

Zum Schluss entsprechen die Wegpunkte annähernd den Städten. Für eine valide Antwort müssen dann die korrespondierenden Städte zu den Wegpunkte gefunden und ausgegeben werden.

Vorteil: Schnelle Lösung ohne grobe Ausreißer.
Nachteil: Keiner
Schwierigkeitsgrad: Mittel

2.4 Genetische Algorithmen - Eine evolutionäre Lösung

Aus der Biologie schaut sich die Informatik sehr viel ab. So auch Lösungen für (fast) unlösbare Probleme. Die Evolution passt sich durch raffinierte Methoden immer wieder aufs Neue an veränderte Umweltbedingungen an. Schon Darwin hat über das *Survival of the Fittest* in der Natur geschrieben. Diese Vorgänge lassen sich auch auf das TSP übertragen. Rundenbasiert werden dabei in *Generationen* verschiedene *Individuen* einer *Population* ausgewählt oder verworfen. Am Ende genügender Generationen kann so ein Individuum ausgewählt werden, die sich am besten auf das Problem eingestellt hat.

Ein Individuum ist in diesem Fall ein valider Rundweg über alle Punkte. Eine Population besteht aus vielen Individuen und wird rundenbasiert (in Generationen) optimiert. Die Startpopulation wird völlig zufällig generiert. Das anschließende Verfahren wiederholt sich dann beliebig häufig.

1. Auswahl der Eltern
   Wähle einen Teil der Population aus, die sich vermehren dürfen
2. Erzeugung neuer Kinder
   Erzeuge solange aus zwei Elementen der Elternmenge ein neues Kind, bis die Population wieder ihre originale Größe erreicht hat
3. Mutation
   Tausche ggf. in einem Individuum zwei Punkte aus

Vorteil: Der Ablauf ist universell auf viele Probleme anwendbar.
Nachteil: Deutlich mehr Code als bei anderen Lösungen.
Schwierigkeitsgrad: Schwer

3 Vorgaben

Um eine Vergleichbarkeit unter den Gruppen zu schaffen und die Resultate leicht visualisieren zu können, haben wir einige Vorgaben zur Verfügung gestellt. Diese Klassen und Interfaces befinden sich in dem Package *tsp.api*. Implementieren Sie bitte Ihre Klassen in einem eigenen Package wie z.B. *tsp.mgroh*.

3.1 City

Die Stadt ist ein Fixpunkt auf der Karte, die besucht werden soll. Jede Stadt hat neben einer x und y Koordinate als Float auch einen (zufälligen) Namen. Auf die Koordinaten kann per Methode zugegriffen werden, zudem sind die Hilfsmethoden *toString*, *clone* und equals implementiert.

3.2 Solvable

Das Interface dient als gemeinsamer Nenner für alle Ihre Lösungen. Vorgeschrieben ist nur, dass die Methoden *setInput*, *hasNext* und *next* implementiert werden müssen. SetInput übergibt Ihrem Programm ein Array von City-Objekten, mit denen Sie arbeiten dürfen. Die Methoden hasNext und next sind Teil eines Iterators. Mit hasNext wird gefragt, ob eine weitere (Zwischen-) Lösung verfügbar ist. Falls ja, kann diese per next erfragt werden. Der letzte Wert, der von next zurückgegeben wird muss die finale (also die beste) Lösung sein. Next gibt einen Rundkurs als City-Array zurück.

3.3 Tsp

Diese Klasse dient der Ausführung und Visualisierung Ihres Programms. Von Interesse ist nur die Methode *run*, welche neben Ihrem Programm (als Implementierung von Solvable) auch eine Konfiguration annimmt.

3.4 Config

Die Konfiguration kann per Default-Constructor erzeugt werden. Standardmäßig wird so eine zufällige Karte mit acht Städten erzeugt und visuell dargestellt. Mit den jeweiligen Set-Methoden können die Eigenschaften *displayMode* (Visuell, Textuell, Ohne), *executionMode* (Zufällige Welt, konkrete Welt, Tournament), *cityCount*, *world* (für die konkreten Welten) und *seed* (für die zufälligen Welten) verändert werden. Wenn Sie einen speziellen Seed nutzen, können Sie mehrfach die gleiche zufällige Karte erzeugen. Der Tournament-Modus dient am Ende als Wettbewerb zwischen Ihnen und Ihren Mitstreitern/-innen. Die Ergebnisse werden von uns gesammelt und geranked.

3.5 World

Mit den statischen Methoden *germany*, *donut*, *ring*, *angle* und *square* können Sie einige vorimplementierte Karten auswählen. Die letzten beiden sind von fester Größe (drei bzw. vier Städte), die Deutschlandkarte hat max. 50 Städte.

3.6 GeneticScheme

Diese abstrakte Klasse dient nur als grobe Vorlage (und Gedankenstütze) für die Implementierung des genetischen Algorithmus. Die Methoden *generatePopulation*, *selection*, *recombination* und *mutation* müssen dabei implementiert werden und geben eine Liste von City-Arrays (also eine Liste von Individuen also eine Population) zurück. Die vorimplementierte Methode *doStep* führt einen Zyklus (eine Generation) durch.

4 Hilfestellungen und Ergänzungen

Für die Bearbeitung sind hier noch ein paar Tipps:

4.1 Generell

Häufig werden zur Bearbeitung mathematische Mengen gebraucht. Speziell, wenn es darum geht eine Menge von Städten zu haben und daraus eine zu entfernen bzw. die Menge der verbleibenden Städte zu betrachten. Dazu lassen sich Listen wie die *LinkedList* oder die *ArrayList* sehr gut nutzen. Sie haben Methoden wie *remove*, *add* und *contains* die wir für die Mengenoperationen brauchen.

Versuchen Sie Ihren Code möglichst sauber in einzelne Aufgabenpakete zu teilen. Steckt in einer Methode zu viel Code verlieren Sie schnell den Überblick und fangen sich Fehler ein. Eine Einteilung in einzelne Teile hat auch den Vorteil, dass man sich Gedanken darüber machen muss, was denn in diese Methode genau reinkommt und rausgeht.

Noch ein ganz wichtiger Hinweis: Die größte Hürde bei dieser Problemstellung sind Referenzen. Achten Sie darauf, dass wenn Sie ein Objekt aus einer Datenstruktur (Liste, Array, ...) lesen und in eine neue schreiben, dass es sich dabei um ein neues, gleiches Objekt handelt nicht weiterhin um das selbe! Das bedeutet, dass wenn Sie das Objekt in der neuen Datenstruktur verändern, Sie nicht auch gleichzeitig das originale Objekt in der original Struktur mit verändern. Wenn Sie eine wirkliche Kopie Ihres Objekts benötigen, nutzen Sie die clone-Methode.

Egal für welche Variante Sie sich entscheiden, benötigen Sie zwei Hilfsfunktionen: Eine gibt Ihnen die Distanz zwischen zwei Städten zurück. Nutzen Sie hierzu beispielsweise die Euklidische Distanz $len(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$ oder die Manhattan Distanz $len(a,b) = |a_x - b_x| + |a_y - b_y|$. Zudem müssen Sie die Länge eines ganzen Rundkurses berechnen. Das ist die summierte Distanz von je zwei Elementen. Denken Sie hierbei auch an den Weg zurück im Array, d.h. wenn Sie immer $dist = dist + len(path[i-1],\ path[i])$ berechnen, müssen Sie auch an $dist = dist + len(path[0],\ path[n-1])$ denken.

4.2 Brute-Force-Permutation

Der einzige Knackpunkt bei dieser Strategie ist, dass Sie in der Schleife an die nächste freie Position je eine andere Stadt einsetzt. Beim rekursiven Aufruf darf dann diese Stadt nicht erneut benutzt werden. Wird aber im nächsten Schleifendurchlauf eine andere Stadt an diese Stelle gesetzt, so muss die nun ersetzte wieder im Spiel sein.

Eine Lösung für diese Aufgabe ist nicht ganz leicht, aber mit etwas Knobelaufwand sicher lösbar. Sie können jede gefundene Lösung in eine Liste von City-Arrays speichern. Das hat den Charme, dass Sie dann den Listeniterator (List.iterator()) für die Methoden hasNext und next nehmen können. Denken Sie nur daran unter den vielen Lösungen auch die beste zu suchen, um sie als letztes erneut zurückzugeben.

4.3 Nearest-Neighbour-Heuristik

Hier gibt gar nicht viel zu erklären. Bei der einfachen Variante (ein beliebiger Startpunkt und los) gibt es nur eine Lösung, weswegen hasNext nur einmalig *true* sagen könnte und next dementsprechend nur eine Antwort ausspuckt. Erst, wenn Sie den Algorithmus nacheinander von jedem möglichen Startpunkt loslaufen lassen, können Sie auch Zwischenlösungen ausgeben.

4.4 Kohonen-Karten

Der generelle Ablauf des Algorithmus ist relativ simpel. Erst das ganze „Drumherum“ macht diese Lösung etwas anspruchsvoller.

Zunächst benötigen Sie eine Klasse für Ihre Wegpunkte. Diese erbt von City und ergänzt sie um die benötigten Attribute für den Ablauf. Da die Koordinaten von City unveränderlich (final) sind, benötigen Sie auch zwei Offset-Werte, um die Positionsänderung zu speichern. Überladen Sie dazu auch die get-Methoden, um die korrekten Werte beim Aufruf von x und y zu erhalten. Um einen Wegpunkt zu duplizieren, sollten Sie auch die Methoden clone und hashCode überladen.

Das Abbruchkriterium Ihrer Iteration besteht aus zwei Teilen. Zum einen muss die Anzahl der Wegpunkte identisch mit der Anzahl der Städte sein. Gestartet wird zunächst mit nur einem Wegpunkt, später können es aber auch mehr als Städte sein, ehe diese sich wieder reduzieren. Zum anderen darf sich die Länge des Rundkurses im Vergleich zum letzten Schritt nur minimal verändert haben (z.B. weniger als 0.1).

Die neue Position eines Wegpunktes berechnet sich aus einem komplexen Faktor. Ist dieser 1 (100%), so ist die neue Position des Wegpunktes die Position der Stadt. Ist er 0, so verändert sich die Position nicht. Der Faktor entspricht der Funktion $f(s,n) = 0.7 * e^{\frac{-n^2}{s^2}}$. S ist die Stärke, die beliebig z.B. 25 oder 50 sein kann, sich aber nach jeder Iteration über alle Städte um 5% verringert. N ist die Anzahl der Nachbarn zwischen dem der Stadt nächsten Wegpunkt und dem aktuellen. Also angenommen es gibt die Wegpunkte $[A, B, C, D, E]$ und B liegt der Stadt S am nächsten. Dann ist $n_B = 0$, $n_A = n_C = 1$ und $n_D = n_E = 2$.

4.5 Genetische Algorithmus

Der generelle Ablauf einer Generation ist in der *doStep* Methode des Schemas schon vorimplementiert. Sie müssen daher „nur“ die Methodenrümpfe ausfüllen.

4.5.1 generatePopulation

Überlegen Sie sich zuerst eine Zielgröße der Population (z.B. 50). Erzeugen Sie dann entsprechend so viele zufällige Rundwege und geben diese als Liste zurück.

4.5.2 selection

Überlegen Sie sih zuerst eine Zielgröße der Elternmenge (z.B. 1/3 der Gesamtmenge). Eine einfache Selektion wäre die Auswahl der x besten (kürzesten) Elemente. Alternativ, um auch einem blinden Huhn eine Chance zu lassen, können Sie auch die Längen der Rundwege normalisieren $lnorm_x = \frac{len_x}{\sum len} + lnorm_{x-1}$. So erhalten Sie für alle Wege einen kumulierten Wert zwischen null und eins. Jetzt können Sie den ersten Wert in der Liste auswählen (vom Besten zum Schlechtesten), der einen Zufallswert übersteigt.

4.5.3 recombination

Die Rekombination ist der wichtigste Schritt für einen guten GA als auch der laufzeitaufwändigste und kompliziertesten. Es gibt unendlich viele Varianten in wissenschaftlichen Papers für eine effiziente Lösung. Eine vergleichsweise einfache ist folgende:

- Wählen Sie einen zufälligen Wert (pivot) zwischen 0 und n (Anzahl Städte)
- Wählen Sie zwei zufällige Wege aus der Elternmenge aus
- Die Städte 0 bis pivot des neuen Kindes sind identisch mit denen des ersten Elternteils
- Die Städte pivot+1 bis n werden mit der jeweils ersten Stadt des zweiten Elternteils aufgefüllt, die bisher noch nicht Teil des Kindes ist

4.5.4 mutation

Überlegen Sie sich zuerst eine Mutationsrate (z.B. 0.1%). Eine einfache Mutation ist das zufällige Vertauschen zweier Städte. Dazu iterieren Sie über alle Städte aller Rundwege und vertauschen diese bei Bedarf mit einer anderen.