Evolutionäre Algorithmen: Lösung eines bestimmten Network-Flow-Problems

Das Single Source Uncapacitated Concave Minimum-Cost Network Flow Problem

Das Single Source Uncapacitated Concave Minimum-Cost Network Flow Problem (SSUC - MCNFP) ist ein Problem bei dem aus einer Quelle mehrere sog. Customer mit Waren versorgt werden müssen. Diese Versorgung geschieht über einen gerichten Graphen bestehend aus Kanten, die über eine mit verschiedenen Kosten versehen sind. Das Ziel ist es, möglichst geringe Gesamt-Kosten zu verursachen während weiterhin alle Customer mit ihren benötigten Waren (Demand) versorgt werden. Dementsprechend ist es ein Minimierungsproblem.

Mathematisch lässt sich das Problem folgendermaßen darstellen:

(1)
$$min : \sum_{(i,j)\in A} g_{ij}(x_{ij}, y_{ij}) = cost$$

(2) mit:
$$g(x_{ij}) = -a_{i,j} * x_{i,j}^2 + b_{i,j} * x_{i,j} + c_{i,j}$$

(3)
$$x_{i,j} \in M, \forall (i,j) \in A$$

(4)
$$x_{i,j} \geq 0, \forall (i,j) \in A$$

(5)
$$\forall j \in N \setminus t \sum_{i|(i,j)\in A} x_{ij}, y_{ij} - \sum_{(j,k)\in A} x_{jk} = storage_j$$

(6)
$$x_{i,j} \leq storage_i, \forall (i,j) \in A$$

In (1) wird die Kostenfunktion beschrieben, die Gesamtkosten sind die Summe aller Kosten der Kanten des Netzwerkes. Dabei ist $g(x_{i,j})$ die Kostenfunktion, die in (2) erklärt wird. Die Werte für $a_{i,j}$, $b_{i,j}$ und $c_{i,j}$ stammen dabei aus dem Datensatz der OR-Library. Aufgrund dieser Werte wurde auch der die Menge der existierenden Kanten M erzeugt, die in (3) genutzt wird. In (3) wird beschrieben, dass nur existente Kanten genutzt werden sollen. (4) bedeutet, dass keine negative Mengen über den Kanten geschickt werden dürfen, damit der Graph weiterhin gerichtet bleibt. $storage_{i/j}$ ist eine Angabe, wieviel Waren an einem Punkt angekommen sind und auch wieviele Waren von diesem Punkt aus verteilt werden können (5). Dabei ist Anfangs der Demand, als negativer $storage_{i/j}$ gesetzt.

OR-Library Testdaten

Die Test-daten sind von der Seite "OR-Library" von J. E. Beasley, unter http://people.brunel.ac.uk/~mastjjb/jeb/orlib/netflowccinfo.html ist es möglich eine Datensatz-Sammlung verschieden großer Datensätze herunterzuladen.

Die Daten werden als einfache, unterteilte Liste bereitgestellt und enthalten die Anzahl an Knoten, einen Demand der Knoten und anschließend verschiedene Kosten für die Kanten. Jedoch können die Daten über Funktionen in drei Matrizen umgewandelt werden. Der letzte Knoten hat keinen Demand, dementsprechend kann er als Quelle angeshen werden.

Die Kosten für die Kanten sind in drei verschiedene Komponenten unterteilt, die Komponente ist A, welche variabel ist und für eine nicht-vorhandene Kante den Wert 0 hat. Die zweite Komponente ist B, die ebenfalls variabel ist, aber für eine nicht-vorhandene Kante den Wert 50000000 annimmt. Die letzte Komponente ist C, die unabhängig von dem sog. Flow ist.

Zusätzlich werden drei verschiedene polynomiale konkave Funktionen bereitgestellt, die als Teil einer Güte-Funktion genutzt werden können.

Hillclimbing im Netzwerk

Der Hillclimbing-Algorithmus basiert auf der Annahme, dass in der Nachbarschaft eines Individuums ein besseres Individuum gefunden werden kann. Zusätzlich basiert er auf der Annahme, dass ausgehend von diesem Individuum ein weiteres gefunden werden kann. In diesem Fall ist das Individuum ein sog. Flow, also wieviele Waren über welche Kante transportiert werden. Dabei wird der Flow in diesem Fall im Programm durch eine Matrix dargestellt.

Das Individuum ist ein struct, also ein Objekt, das einerseits die Flow-Matrix des Individuums beinhaltet, aber außerdem noch den Demand der einzelnen Knoten sowie eine aktuelle Verteilung der waren auf die Knoten, sowie den aktuellen Güte-Wert.

Die Flow-Matrix des Individuums wird durch eine Funktion initialisiert, die verschiedene Sachen wie das Vorhandensein von Kanten, den sog. Demand und auch die Quellenkapazität berücksichtigt. Die Quellenkapazität wird dabei über die Summe der Demands berechnet, da sie nur den Demand befriedigen muss. Die Position der Kanten wird über die Kosten-Matrizen berechnet. Der Demand wird als negatives Lager berücksichtigt, der Überschuss dient anschließend als Obergrenze für eine Zufallsfunktion, die die restlichen Waren weiterverteilt. Anschließend wird der Güte-Wert berechnet.

Der Nachbar wird über eine Funktion gefunden, die genau die gleichen Faktoren berücksichtigt. Jedoch ist der Anfangspunkt nicht die Quelle, sondern eine zufällig ausgewählte Kante. Anschließend werden alle folgenden Punkte über eine Zufallsfunktion mit dem Überschuss als Obergrenze neu verteilt. Dadurch sind

zwar Nachbarn relativ weit voneinander entfernt, aber die Chance tatsächliche Verbesserungen zu erreichen ist deutlich erhöht. Statt Güte-Werten, die konstant im neun- bis zehn-stelligen Bereich sind, wurden dadurch auch Werte im mittleren sieben-stelligen Bereich gefunden.

Der Gütewert wird über die oben genannte Funktion berechnet. Zusätzlich gibt es jedoch einen Strafterm, der von den an den Knoten gespeicherten werden abhängig ist.

Wie es beim Hillclimbing üblich ist, werden die Gütewerte der Individuen verglichen und für das Individuum mit dem besseren, also kleineren, Gütewert wird ein neuer Nachbar gefunden.

Evolution im Netzwerk

Die Evolution im Netzwerk wurde über einen genetischen Algorithmus versucht, greift aber auch teilwise auf die gleichen oder abgewandelte Funktionen des Hill-Climbing zu. Jedoch wurden auch verschiedene Funktionen ausprobiert, die unabhängig von dem Hillclimber entstanden sind.

Generell wurde die Kosten-Funktion beibehalten, die weiterhin eine Güte zurückgibt, die zu minimieren ist. Der Strafterm wurde auch beibehalten.

Abbruchbedingungen

Es gibt für den genetischen Algorithmus in diesem Fall zwei Abbruch-Bedingungen. Die eine Abbruchsbedingung ist an die Anzahl an Generationen geknüpft, die über die cfg.yml eingestellt werden können. Danach gibt der Algorithmus das beste Individuum aus.

Die zweite Abbruchsbedingung ist abhängig von den Güte-Werten. Unter bestimmten Bedingungen können diese sehr hoch werden und sind größer als die Implementation erlaubt. Da go jedoch in der Version 1.14.2 dann nicht abbricht, sondern der Wert als ein anderer negativer Wert interpretiert wird, wird der Algorithmus abgebrochen falls ein Güte-Wert negativ wird. Der Algorithmus wird dann abgebrochen, weil die Wahrscheinlichkeit wieder gute Werte zu erlangen niedrig ist.

Es gibt keine Abbruchsbedingung, die den Algorithmus bei einem bestimmten Minimal-Wert abbricht, weil bei diesem Problem kein Optimal-Wert nicht bekannt ist.

Erzeugen einer Population

Die Population wurde über eine Funktion erzeugt, die, je nach eingestellter Populationsgröße, eine feste Anzahl an Individuen mit der gleichen Funktion,

wie die Individuen des Hillclimbers, erzeugt. Dadurch entsteht eine Population, die einerseits relativ niedrigen Gütewerte hat, aber noch verbessert werden kann. Es wurde sich auch hier gegen eine komplett zufällige Initiierung entschieden, da diese oft verhältnismäßig schlechte Gütewerte produziert, die teilweise den Rahmen des Datentyps gesprengt hat und dann als negativer Wert interpretiert wurden, die dann als besonders gut bewertet wurden. Mit Hilfe der zweiten Abbruchsbedingung wurde eine Verfälschung in Testläufen verhindert.

Selektion von Flows

Für die Selektion wurden verschiedene Selektionsarten ausprobiert. Einerseits wurde die Rang-basierte Selektion ausprobiert, die in der Vorlesung erwähnt wurde:

$$PR[A^{(i)}] = \frac{2}{r}(1 - \frac{i-1}{r-1})$$

Jedoch ist diese für Selektion für Populationengrößen von 2000 nicht gut geeignet, da das beste Individuum nur eine Auswahl-Wahrscheinlichkeit von 1/1000 erreichen kann. Dadurch haben schlechtere Individuen eine höhere Chance für die nächste Generation ausgewählt zu werden. Mit der Rang-basierten Selektion wurde mehrmals die zweite Abbruchbedingung erreicht.

Als Alternative dazu wurde eine Abwandlung der Turnier-Selektion genutzt, die jedes Individuum gegen eine eingestellte Anzahl an zufällig ausgewählten Individuen antreten lässt und diese ersetzt, wenn es gewinnt. Bei dieser Selektion wurden deutliche Verbesserungen erreicht.

TODO: Grafik

Rekombination von Flows

Als Rekombination von Flows wurde das klassische Crossover in Form eines One-Point-Crossovers und eines Two-Point-Crossovers genutzt. Dabei wurden die Crossover-Punkte immer zufällig gewählt.

TODO: Grafiken (One&Two-Point)

Flow-Mutation

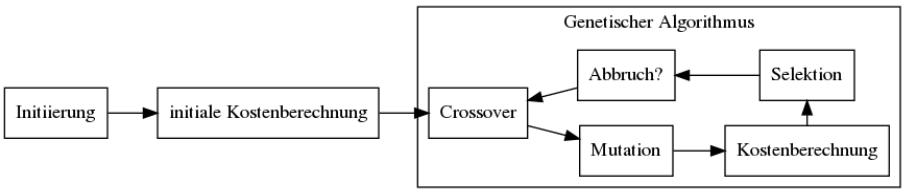
Die Mutation wurde über zwei verschiedene Arten und Weisen versucht. Einerseits wurde eine fast komplett zufällige Mutation ausprobiert, die durch alle möglichen Kanten gegangen ist und sie abhängig vom Mutationsdruck mutieren ließ. Dabei ist der neue Kantenwert abhängig von der storage. Dabei wird jedoch nur eine Kante mutiert und die folgenden Kanten nicht korrigiert.

Andererseits wurde eine bessere Mutation ausprobiert, die durch alle existenten Kanten geht und diese abhängig vom Mutationsdruck und der storage des Anfangsknoten mutiert. Dabei entscheidet der Mutationsdruck nur, ob die Kante verändert wird, während der storage-Wert die Obergrenze des neuen Wertes bestimmt. Was diese Art der Mutation unterscheidet ist, dass nach der Mutation einer Kante die folgenden Kanten angepasst werden.

TODO: Grafiken

Ablauf des Algorithmus

Der Algorithmus ruft bis eine der Abbruchbedingung erreicht werden die einzelnen Funktionen auf, dabei wird jedoch nur einemal eine Population erstellt, danach wird mutiert, rekombiniert und anschließend selektiert. Es wurde sich für diese Reihenfolge entschieden, da sie bessere Ergebnisse liefert und auch mehr Mutation ermöglicht.



Vergleich von Hillclimbing und genetische Algorithmen im Netzwerk