

Inhaltsverzeichnis

1	Einleitung	1
2	Der Algorithmus	1
2.1	Die Geschichte	1
2.2	Algorithmus	2
2.3	Graphen	2
2.4	Die Kernidee	2
2.4.1	Wahrscheinlichkeit für eine Kante	3
2.4.2	Die Pheromonenupdateregeln	3
2.5	Das travelling salesman problem	3
2.6	Die Anwendung	4
3	Anwendungen im Aufbauspiel	4
3.1	Das Problem	4
3.2	Die ACO Regeln	5
3.3	Die Implementation	6
4	Fazit	8

1 Einleitung

Beim der Entwicklung von Algorithmen stellt sich oft die Frage nach der besten Lösung. Es gibt tausende von Möglichkeiten ein gegebenes Problem anzugehen. Eine der besten Inspirationsquellen ist oft die Natur selbst, mit ihren erprobten Methoden liefert sie oft Vorbilder. Eins dieser Vorbilder sind Ameisenkolonien, sie liefern Ideen für das Kommunizieren, für das konstruieren und eben auch für das Wegfinden.

Ant Colony Optimization (auch ACO) nimmt ein Problem als Graphen und findet mögliche Lösungen zu einem Problem. Hierbei hat die Futtersuche der Ameisen als Inspiration gedient, so hinterlassen Ameisen auf der Suche nach Nahrung Pheromone, denen dann wiederum anderen Ameisen folgen. Dadurch können durch relative Simple Abfolgen komplexe Ziele erreicht werden.

Aber warum jetzt Wegfindung in Aufbausimulationen? In Aufbausimulation müssen oft Wege zwischen verschiedenen Punkten gefunden werden. Daher bietet sich hier ACO an. Besonders oft werden Waren zwischen unterschiedlichen Produktionsstätten transportiert werden und dann ist der Träger nichts anderes als die Ameise, der Startpunkt der Ameisenbau und das Ziel die Nahrung. Aufgrund der Iterativen Natur ACO bleiben meine Routen nicht statisch, aber erprobte Wege bleiben erhalten.

2 Der Algorithmus

2.1 Die Geschichte

Der Ant Colony Optimization Algorithmus wurde erstmalig von Marco Dorigo in seiner Doktorarbeit 1992 vorgestellt. Diese Variante ist als das Ant System (AS) bekannt. Seitdem gab es einige Weiterentwicklungen des ursprünglichen Algorithmus, wie zum Beispiel das Ant Colony System oder das Elitist Ant System. Ursprünglich wurde das Ant System anhand des travelling salesman problem (TSP) (s. 2.5) vorgestellt.

2.2 Algorithmus

Eine Algorithmus ist eine "Rechenvorgang nach einem bestimmten [sich wiederholenden] Schema"[5]. Daher werden im den folgenden Abschnitten die benötigten Abläufe für ACO definiert.

2.3 Graphen

Als Graphen versteht man eine Anzahl an Knoten(V_x) und Kanten(E_0). Eine Kante zwischen zwei Knoten wird als (V_a, V_b) bezeichnet, wobei V_a und V_b die beiden verbundenen Knoten sind (Abbildung 1). Werden den Kanten eine Richtung bzw. beide zugewiesen so spricht man von einem gerichteten Graph. Weiterhin

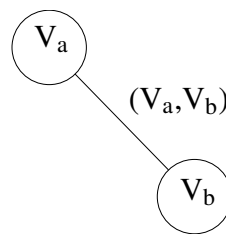


Abbildung 1: Einfacher Graph

kann den Kante oder den Knoten ein Gewicht, also den Kostenmultiplikator für die jeweilige Kante bzw. Knoten. Bei ersterem spricht man von einem kantengewichteten Graph, bei letzterem von einem knotengewichteten Graph.

Zwei Knoten gelten als benachbart wenn es (V_a, V_b) oder (V_b, V_a) gibt.

2.4 Die Kernidee

"In ACO, a number of artificial ants build solutions to an optimization problem and exchange information on their quality via a communication scheme that is reminiscent of the one adopted by real ants."[4]¹. Die Ant Colony Optimization besteht grob aus zwei Bestandteilen, der Wahrscheinlichkeit für eine gegebene Kante und der Pheromonenupdaterregel. Die hier vorgestellten Formel entsprechen der des Ant Systems (AS) (s. 2.1).

¹DE: Bei dem Ant Colony Optimization Algorithmus, entwickeln künstliche Ameisen eine Lösung zu einem Optimierungsproblem und kommunizieren die Qualität ihrer Lösung via eines Verfahrens, welches von Ameisen adaptiert wurde.

2.4.1 Wahrscheinlichkeit für eine Kante

Die Wahrscheinlichkeit für eine Kante gibt an wie hoch die Wahrscheinlichkeit (p) ist, dass eine gegebene Ameise (k) die Kante (V_x, V_y), wobei V_x die momentane Position ist. Sie setzt sich aus der Menge an Pheromonen auf der Kante (τ) und der Effektivität der Kante (η) zusammen

$$P_{xy}^k = (\tau_{xy}^\alpha) * (\eta_{xy}^\beta)$$

. Der Wert wird dann in einen die relative Wahrscheinlichkeit umgerechnet, indem P_{xy}^k durch die Summen der absoluten Wahrscheinlichkeiten aller möglichen Kanten (V_x, V_z) mit V_z als $V_z \in \text{Nachbarn}_x$. Daher ist:

$$p_{xy}^k = \frac{P_{xy}^k}{\sum_{z \in \text{Nachbarn}_x} P_{xz}^k}$$

2.4.2 Die Pheromonenupdaterregel

Der zweite Teil ist die Pheromonenupdaterregel. Sie ist für das Aktualisieren aller Kanten zuständig. Daher hat diese Regel einen großen Einfluss auf das Verhalten der Ameisen. Die einfachste Regel besagt:

$$\tau_{xy} = (1 - \rho) * \tau_{xy} + \sum_k^m \Delta \tau_{xy}^k$$

Wobei ρ angibt wie viel von dem Duftstoff verdunstet, m die Anzahl an Ameisen ist und $\Delta \tau_{xy}^k$ die Menge an Pheromonen ist, die die Ameise k auf der Kante (V_x, V_y) hinterlassen hat. Dementsprechend ergibt sich $\Delta \tau_{xy}^k$ wie folgt wenn die Ameise k über die Kante gegangen ist:

$$\Delta \tau_{xy}^k = Q / L_k$$

Wenn die Ameise nicht über die Kante gegangen ist, dann ist:

$$\Delta \tau_{xy}^k = 0$$

2.5 Das travelling salesman problem

Das travelling salesman problem (in deutsch: Reisehändlerproblem) ist ein relativ bekanntes Problem in der Computertechnik. Es stellt die Frage: “*Given a list of cities and the distances between each pair of cities, what is the shortest possible*

route that visits each city exactly once and returns to the origin city?"[9]². Der Ursprung dieser Fragestellung ist unbekannt.

Soll dieses Problem mit Hilfe von AS gelöst werden, so entspricht die Stadt einem Knoten und alle Knoten sind untereinander verbunden. Danach können die oben beschriebenen Regeln angewandt werden (s. 2.4). Als Beispiel soll der Weg in einem Netzwerk mit sechs Städten gefunden werden, der Graph sähe dann wie in Abbildung 2 aus. An jeder Stadt wird eine Ameise gestartet. Der kürzeste Weg [7] sieht dann wie folgt aus 2 (rote Linien).

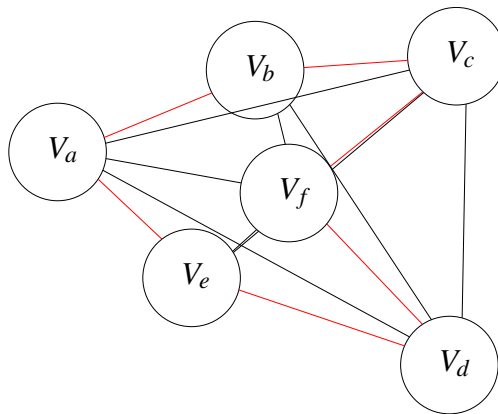


Abbildung 2: Netzwerk an Städten

2.6 Die Anwendung

Nachdem nun der theoretische Teile betrachtet wurde, sind natürlich auch die tatsächlichen Anwendungen für ACO interessant. Einer der offensichtlichen Bereiche ist, der der Wegfindung. So wird ACO unter anderem in der Netzwerktechnik genutzt um Wege für die Pakete zu finden. Aber auch in den Naturwissenschaften findet ACO Anwendung, als Bei-

spiel in der Biologie zur Proteinfaltung oder in der Nanotechnik.

3 Anwendungen im Aufbauspiel

Nachdem nun der Algorithmus betrachtet wurde, folgt jetzt die Übertragung auf den Bereich der Aufbauspiele.

3.1 Das Problem

Ein Aufbauspiel soll aus zwei Teilen bestehen. Der erste sind die Gebäude die Waren verbrauchen und produzieren. Sie bestehen aus einer Liste an benötigten

²DE: Wenn eine List an Städten und deren Entfernungen zueinander bekannt ist, was ist dann der kürzeste Weg um alle Städte genau einmal zu besuchen und zum Anfang zurück zu kehren?

und produzierten Waren, einer Produktionsgeschwindigkeit, die angibt nach welcher Zeit die produzierten Waren, wenn die benötigten Waren vorhanden sind und im Inventar platz für die produzierten ist, in das Inventar hinzugefügt werden. Weiterhin bestehen Gebäude aus einer Inventurkapazität. Der zweite Teil sind die Waren selbst. Sie bestehen nur aus ihrem Volumen, daher wie viel Platz sie in dem Inventar einnehmen.

Als Ausgangslage dient das bauen von Produktionsketten. Da es unterschiedlichste Ketten mit variierender Komplexität gibt, soll Beispielhaft nur die der Bretterproduktion betrachtet werden.

Daher ergeben sich Zwei notwendige Gebäude. Ersteres ist der Holzfäller der in einem festen Zeitintervall von $2t^3$ 1 Holzstamm⁴ produziert, ohne dabei benötigte Ressourcen⁵ zu haben. Die Lagerkapazitäten des Holzfällers sind $1L^6$ Zweiteers Gebäude ist das Sägewerk das alle $1t^3$ aus einem Holzstamm ein Brett⁷ produziert. Das Sägewerk hat eine Lagerkapazität von $2L^6$. Als drittes Gebäude wird noch das Lagerhaus hinzugefügt, es dient nur dazu die produzierten Waren zu sammeln und die benötigten auf die Verbraucher zu verteilen. Weiterhin können die Lagerhäuser auch als Zwischenlager verwendet werden, um überschüssige Waren zu Lagern. Weiterhin dienen sie den Ameisen als Ziel und Ausgangspunkt ihrer Reise.

Die unterschiedlichen Gebäude werden durch Straßen miteinander verbunden.

3.2 Die ACO Regeln

Damit der Standard Ant System Algorithmus für Aufbauspiele funktioniert, müssen gewisse Regeln erweitert werden.

Die Regel für die Effektivität τ besteht nicht nur aus der Distanz, damit die Ameisen nicht die Waren über die ganze Karte tragen, sonder auch aus der Bilanz des Zieles. Die Bilanz beinhaltet wie viel Ressourcen in einer Zeiteinheit t^3 verbraucht (in negativ) und produziert werden (in positiv). Diese Regel soll verhindern, dass die Ameisen nicht ihre Ware zu einem Lagerhaus bringen, das einen Überschuss

³t: Beliebige Zeit

⁴Platz eingenommen: 2 VE

⁵Das fällen von Bäumen und deren Existenz werden zur Vereinfachung weggelassen

⁶L: Volumen des Lagers

⁷Platz eingenommen: 1 VE

an dieser Ware hat.

3.3 Die Implementation

Nachdem die Grundlage für das Spiel gesetzt wurde, soll nun die einzelnen Formeln und Abläufe in Pseudocode übersetzt und erläutert werden.

Ein zentraler Teil des Algorithmus ist die Berechnung der absoluten Wahrscheinlichkeit P .

Funktion *berechneP(kante : Kante) : Dezimalzahl* **ist**
 $P = (\text{kante.Attraktivität} * \alpha) * (\text{effizienz}(\text{kante}) * \beta) : \text{Dezimalzahl}$
 gib P zurück
Ende

Algorithmus 1: Wahrscheinlichkeit P

Diese Funktion ist eine programmatische Umsetzung der unter 2.4.1 beschriebenen Formel für P . Die Funktion *Effizienz(kante)*, den Regeln aus 3.2 folgend, verläuft wie folgt:

Funktion *effizienz(kante : Kante) : Dezimalzahl* **ist**
 ziel = null : Knoten
 wenn *kante.A gleich momentanePosition* **dann**
 ziel = kante.B
 sonst
 ziel = kante.A
 Ende
 $d = 1 : \text{Dezimalzahl}$
 $d = \sqrt{(kante.A.x - kante.B.x)^2 + (kante.A.y - kante.B.y)^2}$
 wenn *ziel.gebäude ist Variante von Lagerhaus und*
 ziel.gebäude.balance enthält transportierteRessource **dann**
 $d = d + \theta * \text{ziel.gebäude.balance}(\text{transportierteRessource})$
 gib $\frac{1}{d}$ zurück
Ende

Algorithmus 2: Effizienz

Im ersten Teil der Funktion wird entschieden welche der beiden Knoten der Kante genommen betrachtet werden soll. Da nicht zu der eigenen Position gegangen werden soll, müssen daher die Werter des anderen Knotens verwendet

werden. Als nächster Schritt ist dann die Distanz zwischen den beiden Knoten zu berechnen. Sollte dann das Gebäude an dem Zielknoten ein Lagerhaus sein, so wird der Verbrauch bzw. die Produktion der transportierten Ressource auf die Distanz ab bzw. aufgeschlagen. Der Rückgabewert entspricht $1/d$, damit größere Strecken unattraktiver sind.

Im Nächsten Schritt muss den die relative Wahrscheinlichkeit p berechnet werden. Dazu wird die absolute Wahrscheinlichkeit P durch die Summe aller Wahrscheinlichkeiten geteilt.

```
Funktion summeWahrscheinlichkeiten() : Dezimalzahl ist
|   ergebnis = 0 : Dezimalzahl
|   für jedes momentan : Kante in momentanePosition.Kanten tue
|       |   ergebnis = ergebnis + berechneP(momentan)
|   Ende
|   gib ergebnis zurück
Ende
```

Algorithmus 3: Summe der Wahrscheinlichkeiten

Dieser Algorithmus ist das Iterative aufrufen und addieren der Ergebnisse der Funktion P , für jede Kante die von der momentanen Position ausgehen.

Daher Ergibt sich die relative Wahrscheinlichkeit p wie folgt:

```
Funktion berechneP(kante : Kante) : Dezimalzahl ist
|   gib  $\frac{\text{berechneP}(\text{kante})}{\text{summeWahrscheinlichkeiten}()}$  zurück
Ende
```

Algorithmus 4: Wahrscheinlichkeit p

Damit nun eine Kante selektiert werden kann, muss zuerst jeder vom momentanen Knoten ausgehenden Kante eine Wahrscheinlichkeit zugewiesen werden und dann eine ausgewählt werden. Da p eine relative Wahrscheinlichkeit ist, kann ein Wert zwischen 0 und 1 (0% und 100%) bestimmt werden und dann alle Wahrscheinlichkeit aufaddiert werden, bis dieser Schwellwert überschritten ist. Der Pseudocode sieht daher so aus:

Funktion *selektiereKante()* : *Kante* **ist**
 | wahrscheinlichkeiten = Array mit Länge von
 | momentanePosition.Kanten : Dezimalzahlarray
 | index = 0 : Zahl
 | **für jedes** *momentan* : *Kante in momentanePosition.Kanten* **tue**
 | | wahrscheinlichkeiten[index] = berechne(momentan)
 | | index = index + 1
 | **Ende**
 | schranke = Zufallszahl zwischen 0 und 1 : Dezimalzahl
 | summe = 0 : Dezimalzahl
 | index = 0
 | **für jedes** *momentan* : *Dezimalzahl in wahrscheinlichkeit* **tue**
 | | summe = summe + momentan
 | | **wenn** *summe ist größer als schranke* **dann**
 | | | gib momentanePosition.Kanten[index] zurück
 | | **Ende**
 | | index = index + 1
 | **Ende**
 | gib null zurück
Ende

Nachdem die Wegfindung

Algorithmus 5: Selektion einer Kante

4 Fazit

Literatur

- [1] baeldung. *Ant Colony Optimization* | Baeldung. 21. Feb. 2021. URL: <https://www.baeldung.com/java-ant-colony-optimization>.
- [2] Daniel Blum. “Ant Colony Optimization (ACO)”. In: (). URL: <https://ls11-www.cs.tu-dortmund.de/lehre/SoSe03/PG431/Ausarbeitungen/ACO.pdf>.
- [3] Alberto Coloni, Marco Dorigo, Vittorio Maniezzo u. a. “Distributed optimization by ant colonies”. In: *Proceedings of the first European conference on artificial life*. Bd. 142. Paris, France. 1991, S. 134–142.
- [4] Marco Dorigo. “Ant colony optimization”. In: *Scholarpedia* 2.3 (2007), S. 1461.

- [5] Duden. *Duden | Algorithmus | Rechtschreibung, Bedeutung, Definition, Herkunft*. 3. März 2021. URL: <https://www.duden.de/rechtschreibung/Algorithmus>.
- [6] Matthias Teschner. *Algorithmen und Datenstrukturen Graphen - Einführung*. 25. Feb. 2021. URL: https://cg.informatik.uni-freiburg.de/course_notes/info2_15_graph.pdf.
- [7] thiagodnf. *ACO Simulator*. 28. Feb. 2021. URL: <http://thiagodnf.github.io/aco-simulator/#>.
- [8] unkown. *Ant colony optimization algorithms - Wikipedia*. 28. Feb. 2021. URL: https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms.
- [9] unkown. *Travelling salesman problem - Wikipedia*. 27. Feb. 2021. URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem.