

Wege im Aufbauspiel Ant Colony Optimization

Adrian Kumbrink

11. März 2021

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	TSP	4
2.6	Die Anwendung	4
3	Anwendungen im Aufbauspiel	4
3.1	Das Problem	5
3.2	Die ACO Regeln	5
3.3	Die Implementation	6
4	Bewertung	10
4.1	Nachteile	10
4.2	Vorteile	11
4.3	Erweiterungen	11
5	Fazit	11
6	UMLs	12
7	Literatur	13

1 Einleitung

Bei 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 die Natur selbst. Mit ihren erprobten Methoden liefert sie oft Vorbilder. Eins dieser Vorbilder sind Ameisenkolonien. Sie geben Ideen für das Kommunizieren, für das Konstruieren und eben auch für das Wegfinden.

Ein typisches Problem ist die Routenplanung, wo in einem Netzwerk (Graph) der optimalste Wege gefunden werden soll. Ant Colony Optimization (auch ACO) nimmt ein solches Problem als Graphen und findet mögliche Lösungen zu eben diesem Problem. Hierbei hat die Futtersuche der Ameisen als Inspiration gedient. In der Natur zeigen sie durch ihre Schwarmintelligenz, dass viele einzelne Ameisen die weite ihres Lebensraumes auf wenige effizient gewählte Straßen vereinfachen können. So hinterlassen sie auf der Suche nach Nahrung Pheromone, denen dann wiederum anderen Ameisen folgen, daher Straßen bilden. Dadurch können relative Simple Abfolgen komplexe Ziele erreichen.

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 und dann ist der Träger nichts anderes als die Ameise, der Startpunkt der Ameisenbau und das Ziel die Nahrung. Aufgrund der iterativen Natur der ACO bleiben die 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

Ein Algorithmus ist ein "Rechenvorgang nach einem bestimmten [sich wiederholenden] Schema"[5]. Daher werden in 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). Das Bewegen zwischen V_a nach V_b über (V_a, V_b) wird durch die Kosten bewertet. Diese sind im einfachsten Fall die

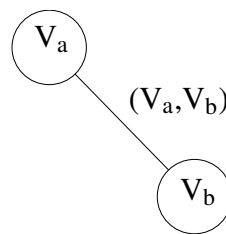


Abbildung 1: Einfacher Graph

Länge der Kante. Werden den Kanten eine Richtung bzw. beide zugewiesen so spricht man von einem gerichteten Graphen. Weiterhin kann den Kanten oder den Knoten ein Gewicht, also ein Kostenmultiplikator für die jeweilige Kante bzw. Knoten zugewiesen werden. Bei ersterem spricht man von einem kantengewichteten Graphen, bei letzterem von einem knotengewichteten Graphen.

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 Formeln 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) nimmt, wobei V_x die momentane Position ist. Sie setzt sich aus der Menge an Pheromonen, der Attraktivität, auf der Kante (τ) und der Effektivität der Kante (η) zusammen. Der Einfluss der beiden Faktoren kann durch den Nutzer in α bzw. β beeinflusst werden.

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

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

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

2.4.2 Die Pheromonenupdateregeln

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

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

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:

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

L_k sind die Kosten, die die Ameise k aufwenden musste um die Kante (V_x, V_y) zu passieren. Q ist ein weiterer Faktor, mit dessen Hilfe der Nutzer den Algorithmus beeinflussen kann. 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: Problem des reisenden Händler) 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 jede 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 (s. Abbildung 2 rote Linien).

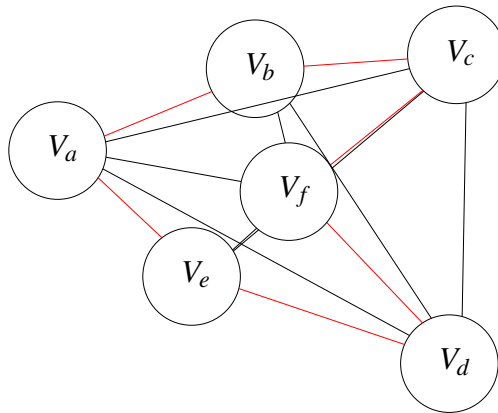


Abbildung 2: Netzwerk an Städten

2.6 Die Anwendung

Nachdem nun der theoretische Teil 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 Datenpakete zu finden. Aber auch in den Naturwissenschaften findet ACO Anwendung, zum Beispiel 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.

²DE: Wenn eine Liste 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?

3.1 Das Problem

Ein Aufbauspiel soll aus zwei Teilen bestehen. Der erste sind die Gebäude die Waren verbrauchen und produzieren. Der zweite Teil sind die Waren selbst. Die Gebäude bestehen aus einer Liste an benötigten und produzierten Waren und einer Produktionsgeschwindigkeit. Diese gibt an nach welcher Zeit die produzierten Waren in das Inventar hinzugefügt werden. Bedingung dafür ist, dass die benötigten Waren vorhanden sind und im Inventar Platz für die produzierten ist. Die Größe des Inventars wird durch die Inventarkapazität angegeben. Die Waren bestehen nur aus ihrem Volumen, das beschreibt wie viel Platz sie in dem Inventar einnehmen.

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

Daher ergeben sich zwei notwendige Gebäude. Ersteres ist die Holzfällerhütte, die in einem festen Zeitintervall von $2t^3$ einen Holzstamm⁴ produziert, ohne dabei benötigte Ressourcen⁵ zu haben. Die Lagerkapazitäten des Holzfällers sind $1L^6$. Das zweite 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. Außerdem 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.

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

Die Regel für die Effektivität τ besteht nicht nur aus der Distanz, damit die Ameisen nicht die Waren über die ganze Karte tragen, sondern auch aus der Bilanz des Zieles. Die Bilanz beschreibt wie viel Ressourcen die benachbarten Gebäude in einer Zeiteinheit t^3 verbrauchen (negativ) und produzieren (positiv). Entsteht ein Überschuss an einer Ware so ist deren Wert positiv. Wird mehr als vorhanden benötigt, so ist der Wert negativ. Diese Regel soll verhindern, dass die Ameisen ihre Ware zu einem Lagerhaus bringen, das einen Überschuss an dieser hat. Den Einfluss von der Bilanz kann von dem Nutzer durch θ beeinflusst werden.

3.3 Die Implementation

Nachdem die Grundlage für das Spiel gesetzt wurde, sollen 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**

$P = (\text{kante.Attraktivität} * \alpha) * (\text{effektivität}(\text{kante}) * \beta) : \text{Dezimalzahl}$
 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 *effektivität(kante)*, den Regeln aus 3.2 folgend, verläuft wie folgt:

Funktion *effektivität(kante : Kante) : Dezimalzahl* **ist**

 ziel = null : Knoten

wenn *kante.A gleich momentanePosition* **dann**

 | ziel = kante.B

sonst

 | ziel = kante.A

Ende

$d = \sqrt{(kante.A.x - kante.B.x)^2 + (kante.A.y - kante.B.y)^2}$:
 Dezimalzahl

wenn *ziel.gebäude ist Variante von Lagerhaus und*

 | *ziel.gebäude.balanceEnthält(transportierteRessource)* **dann**

 | $d = d + \theta * \text{ziel.gebäude.balance(transportierteRessource)}$

 gib $\frac{1}{d}$ zurück

Ende

Algorithmus 2: Effektivität

Im ersten Teil der Funktion wird entschieden, welcher der beiden Knoten der Kante genommen und betrachtet werden soll. Da nicht zu der eigenen Position gegangen werden soll, müssen daher die Werte des anderen Knotens als Zielwerte verwendet werden. Als nächster Schritt ist dann die Distanz zwischen den beiden Knoten zu berechnen. Sollte 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 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 addieren der Ergebnisse der Funktion P für jede Kante,

die von der momentanen Position ausgehen, iterativ.

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 Kante ausgewählt werden. Die Varianz des Algorithmus ergibt sich in dieser Auswahl. Hierzu wird zufällig ein Schwellwert zwischen 0 und 1 (0% und 100%) bestimmt. Dann werden alle errechneten Wahrscheinlichkeiten solange summiert bis dieser Wert überschritten ist. Diese Kante ist dann die Kante, welche die Ameise in der Runde gehen wird.

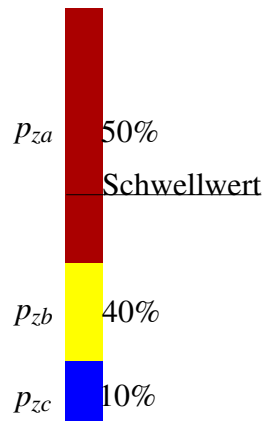


Abbildung 3: Aufbau der Liste an Wahrscheinlichkeiten

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
```

Algorithmus 5: Selektion einer Kante

Nachdem die Kanten für alle Ameisen bestimmt und diese auch gegangen worden sind, müssen nun die Pheromone auf den Kanten neu verteilt werden. Dazu merkt sich das Programm, über welche Kante sich in der jetzigen Runde jeweils eine Ameise bewegt hat. Im ersten Teil wird dann die allgemeine Verdunstung berechnet.

Funktion *verdunste()* : nichts ist

```

    für jedes momentanerKnoten : Knoten in alleKnoten tue
        für jedes momentaneKante : Kante in momentanerKnoten.Kanten
            tue
                wenn momentaneKante.A gleich momentanerKnoten dann
                    momentaneKante.Attraktivität =
                        momentaneKante.Attraktivität * (1 -  $\rho$ );
                Ende
            Ende
        Ende
    Ende

```

Algorithmus 6: Verdunstung

Als zweiter Schritt werden dann alle begangenen Kanten, entsprechend den Regeln, attraktiver gemacht.

Funktion *belohne()* : nichts ist

```

    für jedes momentane : Kante in gegangeneKanten tue
        d = 1 : Dezimalzahl
        d =
             $\sqrt[2]{(\text{momentane.A.x} - \text{momentane.B.x})^2 + (\text{momentane.A.y} - \text{momentane.B.y})^2}$ 
        momentane.Attraktivität = momentane.Attraktivität * Q / d
    Ende

```

Hier wird der Einfachheit halber die Kosten mit der Länge gleichgesetzt.

Hier nicht gezeigt, aber dennoch von Bedeutung ist die Bewegung der Ameisen. Auch die Entscheidung welche Ressource transportiert werden soll, die Menge derer und der Abgabeort werden hier nicht weiter betrachtet.

4 Bewertung

4.1 Nachteile

Natürlich hat die Anwendung des ACO Algorithmus auch ihre Nachteile. So ist er nicht Deterministisch. Dies bedeutet das bei zwei exakt gleichen Graphen, unterschiedliche Wege entstehen können. Dies hat zur Folge das es eine optimale und

reproduzierbare Lösung nicht geben kann. Für ein optimales Ergebnis müssen die Faktoren (α, β etc.) experimentell ermittelt werden.

Da der Algorithmus die einzelnen Warentransporte simuliert, eignet er sich nicht für Simulationen mit großem Transportaufkommen.

4.2 Vorteile

Die Zufallskomponente des Algorithmus fördert die Spieldynamik. Da es eine absolut beste Lösung nicht gibt, lädt der Algorithmus den Spieler zu eigenem Experimentieren ein. Weiterhin passt sich ACO neuen Gegebenheiten automatisch an. Aufgrund der immer fortwährenden Optimierung, erfolgt ein solches Anpassen nur langsam. Dies bildet reale Prozesse eher ab.

4.3 Erweiterungen

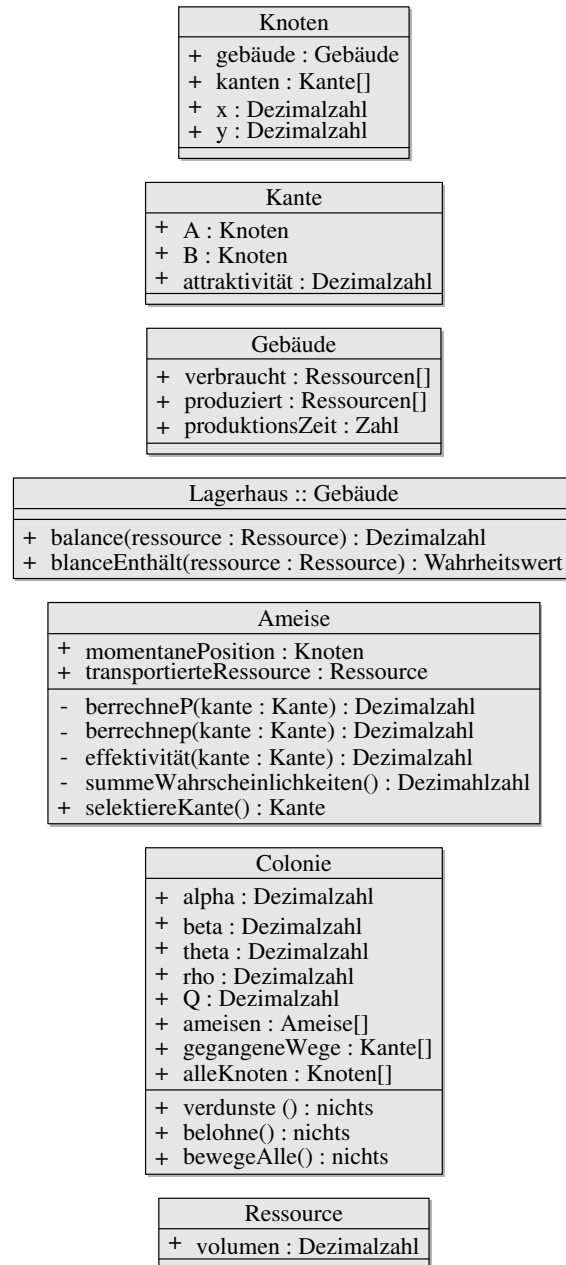
Der in 3.3 beschriebene Pseudocode bietet noch Möglichkeiten der Verbesserung. Eine dieser wäre die Berechnung der Wahrscheinlichkeit zu parallelisieren, da die Berechnungen der einzelnen Wahrscheinlichkeiten nicht von einander Abhängig sind. Auch eine Analyse des Algorithmus, besonders im Hinblick auf sprachliche Möglichkeiten, könnte sinnvoll sein.

Auch der Algorithmus selbst bietet Möglichkeiten für Erweiterung. So könnte die Berechnung der Pheromone so erweitert werden, dass viel genutzte Kannte gemieden werden. Diese hätte eine dynamischere Verteilung der Ameisen zur Folge. Auch eine Betrachtung der Umgebung des Zieles, kann bessere Resultate liefern.

5 Fazit

Der ACO Algorithmus ist durchaus ein valide Option für Aufbauspiele. Allerdings gibt es einige Voraussetzungen. Soll das Spiel komplett vorhersehbar sein, so funktioniert der Algorithmus nicht. Ebenso bereitet er Problem bei Spielen mit massiver Größe an Simulationen (wie z.B. Anno 1800). Wird aber ein Spiel wie die Siedler angestrebt, in denen vielfältiger Warentransport erwünscht ist, ist ACO durchaus eine Überlegung wert.

6 UMLS



8

⁸Die hier gezeigten Diagramme enthalten nur im obigen explizit erwähnte Funktionen und Variablen

7 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/Ausarbeitung/en/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.

Quellcode, Orginaldateien können in dem Github repository eingesehen werden.