Wege im Aufbauspiel
Ant Colony Optimization

Adrian Kumbrink

10. März 2021

Informatik GK Buyken - Conrad von Soest Gymnasium

Inhaltsverzeichnis

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 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 knotengewichtetem Graphen.

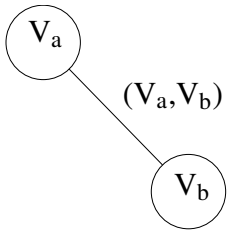


Abbildung 1: Einfacher Graph

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 Pheromonenupdateregel. Die hier vorgestellten Formeln entsprechen der des Ant Systems (AS) (s. 2.1).

[1]DE: 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.

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

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 $V_z \in Nachbarn_x$ geteilt wird.Daher ist:

$$p_{xy}^k = \frac{P_{xy}^k}{\sum_z^{Nachbarn_x} P_{xz}^k}$$

2.4.2 Die Pheromonenupdateregel

Der zweite Teil ist die Pheromonenupdateregel. 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 verdampft, 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).

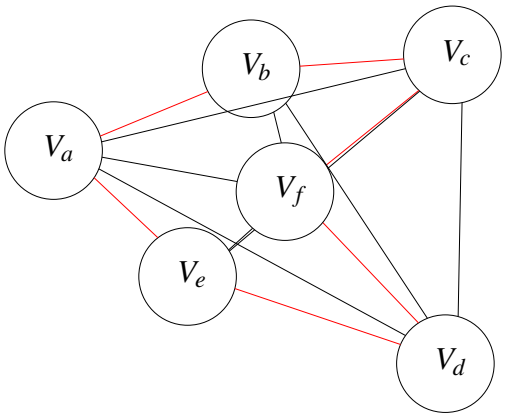


Abbildung 2: Netzwerk an Städten

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.

[2]DE: 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 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 Holzstamm[4] produziert, ohne dabei benötigte Ressourcen[5] zu haben. Die Lagerkapazitäten des Holzfällers sind 1L[6]. Das zweite Gebäude ist das Sägewerk, das alle 1t[3] aus einem Holzstamm ein Brett[7] 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.

[3] t: beliebige Zeit
[4] Platz eingenommen: 2 VE
[5] Das Fällen von Bäumen und deren Existenz werden zur Vereinfachung weggelassen.
[6] L: Volumen des Lagers
[7] Platz 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 = (kante.Attraktivität * α) * (effektivität(kante) * β) : Dezimalzahl
    gib P zurück
Ende

Algorithmus 1: Wahrscheinlichkeit P

Diese Funktion ist eine programmatische Umsetzung der unter 2.4.1 beschrieben 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 = $\sqrt[2]{(kante.A.x - kante.B.x)^2 + (kante.A.y - kante.B.y)^2}$: Dezimalzahl
wenn *ziel.gebäude ist Variante von Lagerhaus und ziel.gebäude.balanceEntält(transportierteRessource)* **dann**
d = d + θ * 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 $\frac{berechneP(kante)}{summeWahrscheinlichkeiten()}$ 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, welch die Ameise in der Runde gehen wird.

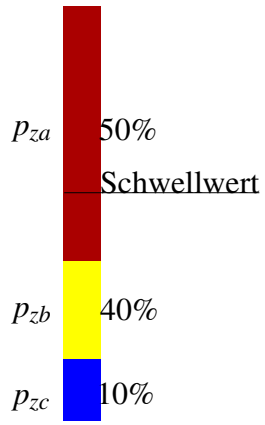


Abbildung 3: Aufbau der Liste an Wahrscheinlichkeiten

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] = berechnep(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 worden ist und diese auch gegangen sind, müssen nun die Pheromonen 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 momentanerKonten.Kanten*
    tue
      wenn *momentaneKante.A gleich momentanerKnoten* **dann**
        momentaneKante.Attraktivität =
        momentatneKante.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 geganeneKanten* **tue**
    d = 1 : Dezimalzahl
    d = $\sqrt[2]{(momentane.A.x - momentane.B.x)^2 + (momentane.A.y - momentane.B.y)^2}$
    momentane.Attraktivität = momentane.Attraktivität * Q / d
  Ende
Ende

Hier nicht gezeigt, aber dennoch von Bedeutung ist das Bewegen der Ameisen, sowie das Entscheiden über welche Ressource und wie viel davon transportiert werden sollen und wann sie wieder abgeben werden sollen.

3.4 Erweiterungen

Der in 3.3 beschriebene Pseudocode kann noch erweitert und verbessert werden. Zuerst einmal gibt es bestimmt noch effizientere Methoden die Formeln umzusetzen. Eine weitere Möglichkeit wäre die Abläufe zu parallelisieren, da die Berechnungen der einzelnen Wahrscheinlichkeiten nicht von einender Abhängig ist.

Inhaltlich würde es sich eventuell lohnen den Ameisen mehr Voraussicht zu geben,

daher wenn die Wahrscheinlichkeit für einen Knoten berechnet wird, die Produktion und den Verbrauch der Nachbarn ebendieses Knotens zu beachten. Weiterhin könnte eine Funktion über das Pheromonenlevel gelegt werden, sodass sehr viel benutzte Kanten gemieden werden. Diese hätte eine dynamischere Verteilung der Träger zu folge.

3.5 Nachteile

Natürlich hat die Anwendung des ACO Algorithmus auch ihre Nachteile. So ist er nicht Determinativ, daher wenn zwei mal genau die gleiche Produktionsketten gebaut werden, können sie doch unterschiedlich Effizient sein, da die Träger unterschiedliche Wege gehen. Dies hat nicht nur zur folge das es keine beste Lösung gibt, daher Spieler nicht auf die Straße genau optimieren können, sondern auch, wenn die Faktoren ($\alpha, \beta etc.$) nicht richtig eingestellt sind, dass sich die Träger sehr dämlich verhalten.

Der wohl größte Nachteil ist, dass für diesen Algorithmus alle Wege simuliert werden, daher nicht einfach feste Zahlen in Zeitintervallen addiert werden können.

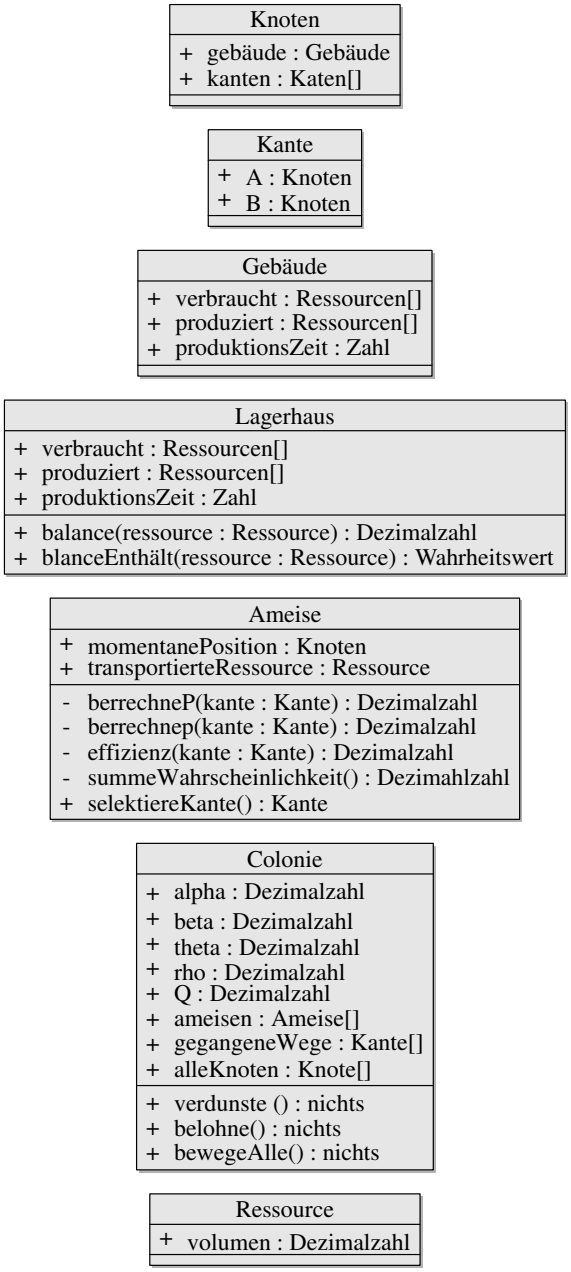
3.6 Vorteile

Der in 3.5 erwähnte Nachteil der Inkonsistenz, hat aber auch zur folge das eine gewisse Dynamik in das Spiel kommt und nicht nur die beste Lösung im Internet gesucht werden kann. Daher wird das eigene Experimentieren gefördert. Außerdem ist der Algorithmus Dynamisch und Konsistenz, daher wenn einmal ein Weg gefunden wurde wird sich diese kaum verändern, doch sollte ein neuer Teil hinzugefügt werden, so würden sich die Träger langsam an die neuen Gegebenheit anpassen.

4 Fazit

Daher ist der ACO Algorithmus durchaus ein valide Option für Aufbauspiele. Allerdings gibt es einige Voraussetzungen, sollte das Spiel komplett berechenbar sein, so funktioniert der Algorithmus nicht. Ebenso bereitet er Problem bei Spielen mit massiver Größe an Simulation (wie z.B. Anno 1800). Wird aber ein Spiel wie die Siedler angestrebt ist ACO durchaus eine Überlegung wert.

5 UMLs



[8]

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

6 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 Colorni, 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 DatenstrukturenGraphen - 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.