TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
Die Ressourcenuniversität. Seit 1765.

Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Künstliche Intelligenz und Datenbanken

**Bakkalaureatsarbeit**

# Eine Studie zur kombinatorischen Optimierung mit Ameisenalgorithmen

**Samuel Dressel**

Angewandte Informatik
Vertiefung: Künstliche Intelligenz

Matrikel: 59 963

05. November 2018

Betreuer/1. Korrektor:
Prof. Dr. H. Jasper

2. Korrektor:
M. Sc. V. Göhler

# Inhaltsverzeichnis

# 1. Einleitung

# 2. Grundlagen

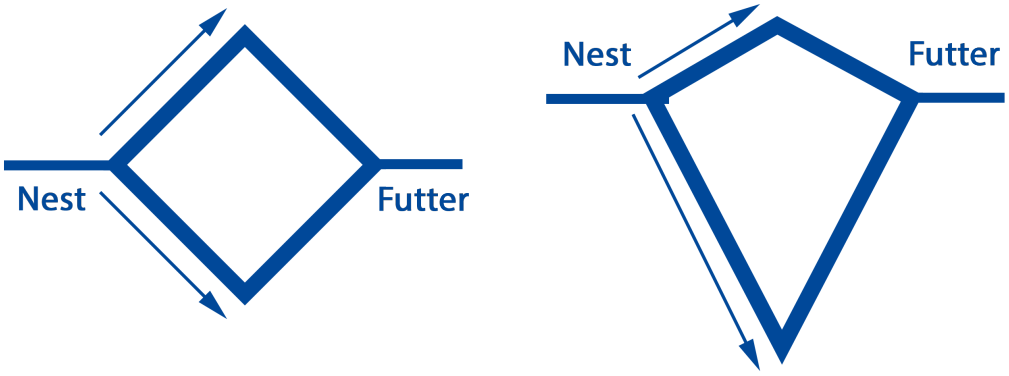
## 2.1. Der Ameisenalgorithmus (Ant Colony Optimization)

### 2.1.1. Biologische Grundlagen

Für das weitere Verständnis des *Ameisenalgorithmus* (auch Ant Colony Optimization Algorithm oder kurz ACO) ist zunächst ein Blick auf die biologischen Grundlagen notwendig. Ein Tierstaat, wie er bei den Ameisen zu finden ist, funktioniert nur mit einer effektiven und sinnvollen Kommunikation. Methoden zur Verständigung wie das Kommunizieren über Vibrationen und Berührungen sind eher die Ausnahme und kommen nur in speziellen Situationen zum Tragen [1]. Dagegen kommt zum größten Teil der Informationsaustausch über Duftstoffe (sog. Pheromone) zur Anwendung. Diese werden durch verschiedene Drüsen erzeugt und wiederum in unterschiedlicher Kombination und Konzentration abgegeben. Diese Pheromone werden benutzt, um Nestgenossen zu erkennen oder um bei Gefahren Kampf- und Abwehrverhalten auszulösen. Hauptsächlich jedoch nutzen Ameisen die Pheromone um eine Duftspur über ihren Hinterleib abzugeben. Diese dient ihnen und dem restlichen Tierstaat als Orientierungshilfe. Zum einen werden damit Straßen zu anderen Kolonien gebildet - zum anderen dienen sie dazu, anderen Ameisen den Weg zu einer Nahrungsquelle zu zeigen. Die Tatsache, dass ein Weg mit einer höheren Pheromonkonzentration bevorzugt wird, ist Grundlagen des Ameisenalgorithmus.

### 2.1.2. Der Ameisenalgorithmus

Der historische Ursprung des Algorithmus findet sich in den Versuchen von Jean-Louis Deneubourg und seine Kollegen [2]. Das sogenannte „Double-Bridge-Experiment“ zeigte, dass Ameisen den kürzesten Weg aufgrund der Pheromonmarkierung finden. In diesem Experiment ist eine Kolonie von Argentinischen Ameisen mit einer Nahrungsquelle durch zwei Brücken verbunden, die gleichzeitig den einzigen Zugang zu dieser Nahrungsquelle bilden. [3]. Dabei können die Ameisen die Futterquelle nur über diese zwei Brücken erreichen. Im ersten Teil des Versuchs sind diese beiden Brücken jeweils gleich lang (siehe Abbildung 1a). Zu Beginn erkunden die Ameisen die Umgebung der Kolonie bis sie eine Entscheidung über die Auswahl der Brücke treffen müssen. Lässt sich aufgrund einer noch nicht stattgefundenen Begehung der Brücken keine Pheromonspur feststellen, so entscheiden die Ameisen rein zufällig welche Brücke sie wählen. Die Wahrscheinlichkeit für beide Wege liegt bei gleichen Bedingungen bei 50 Prozent. Wird der Versuch über längere Zeit durchgeführt, so wird durch Zufall die Pheromonkonzentration der einen Brücke höher sein als die der anderen. Diese wird dadurch attraktiver für die Ameisen und wird somit letztenendes der favorisierte Weg zur Nahrungsquelle.
Im zweiten durchgeführten Versuch sind die beiden Brücken unterschiedlich lang (siehe Abbildung 1b). Auch hier liegt die Wahrscheinlichkeit für beide Brücken zu Anfang bei 50 Prozent. Da die Ameisen, die sich für die kürzere Brücke entscheiden, schneller wieder zurück am Nest sind, steigt das Pheromonlevel auf diesem Weg deutlich schneller an. Nach einiger Zeit kristalliert sich die kürzere Brücke als optimale Route heraus, während der längere Weg durch eine natürliche Pheromonverdunstung immer unattraktiver wird. Im Vergleich zum ersten Versuch geschieht der Ausbau der Pheromonspur wesentlich schneller und effektiver.




**(a)** Skizze zu Versuch 1; beide Wege sind gleich lang

**(b)** Skizze zu Versuch 2; die Wege sind unterschiedlich lang

**Abb. 1**: Double-Bridge-Experiment nach Deneubourg [2]

## 2.2. Das Travelling-Salesman-Problem

### 2.2.1. Das Problem im Allgemeinen

Das *Travelling Salesman Problem* (im Folgenden mit TSP abgekürzt) ist eines der bekanntesten und meist untersuchten Optimierungsprobleme [4]. Kerninhalt des Problems ist dabei folgender: Ein Handlungsreisender soll in einer Rundreise $n$ verschiedene Städte besuchen. Der Reisende startet dabei (zufällig) in einer dieser Städte und am Ende seiner Reise kehrt er auch wieder in diese Stadt zurück. Dabei sollen alle $n$ Städte nur einmal besucht werden und die Weglängen bzw. die Kosten der gesamten Reise minimal sein.
Seinen geschichtlichen Ursprung hat das TSP im Jahr 1832, als in Deutschland ein Buch mit dem Titel „Der Handlungsreisende, wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolges in seinen Geschäften gewiss zu sein“erschien. Dieses Buch und dessen Inhalt diente als Grundlage für die Erforschung des Problems. Die erste Benutzung des Ausdrucks „Traveling Salesman Problem“erfolgte in mathematischen Kreisen etwa um das Jahr 1931 [5]. Dies geschah als mehrere amerikanische Mathematiker sich des Problems annahmen - wichtige Vertreter waren dabei Merrill Flood und Hassler Whitney, die die Überlegungen des österreichischen Mathematikers Karl Menger als Grundlage nahmen. Nach und nach wurde das Problem durch zahlreiche Veröffentlichungen immer präsenter und bis heute ist das TSP eines der prominentesten und am besten untersuchten Probleme in der Mathematik.

### 2.2.2. Graphentheoretische Grundlagen

Um das Problem formal als ein graphentheoretisches Problem darzustellen, werden im Folgenden die dafür benötigten Begriffe definiert [6]:

$G = (V, E, c)$ sei zunächst ein *ungerichteter Graph*. Dabei beschreibt $V = \{1, ..., n\}$ die Menge der Knoten und $E$ die Menge der Kanten, welche letztenendes eine Menge von ungeordneten Paaren $e \in E = \{i, j\}$ mit $i, j \in V$ ist. $c$ beschreibt die Gewichtung der einzelnen Kanten in $E$.

Ist $e = ij = \{i, j\}$ eine Kante von $G$, dann verbindet $e$ die Knoten $i$ und $j$. Diese Knoten $i, j$ heißen dann *adjazent*. Die Kante $e$ dagegen heißt dann *inzident* zu $i$ und $j$.

Die Menge $N(v)$ aller Knoten von $G$, die zu $v$ adjazent sind, nennt man *Nachbarschaft* von $v$.

Ein weiterer wichtiger Begriff ist der *Grad* $d$ eines Knotens $v \in V$. Dabei ist der Grad $d(v)$ die Anzahl der zu $v$ inzidenten Kanten.

Eine *Kantenfolge* $W = (v_0v_1, v_1v_2, ..., v_{r-1}v_r)$ eines Graphen $G$ ist eine Folge von Kanten. Wenn alle Kanten in dieser Kantenfolge verschieden sind, dann heißt diese Folge *Kantenzug*. Sind zudem noch alle Knoten paarweise verschieden, so erhält man einen *Weg* $P$.

Gilt $v_0 = v_r$, so heißt der Weg geschlossen oder auch *Kreis*.

Enhält so ein Kreis $C$ nicht unbedingt jede Kante $e$ von $G$, aber dafür jeden Knoten $v$ von $G$ genau einmal, so nennt man diesen Kreis einen *Hamiltonkreis* $\pi$ (auch *Tour*) von $G$ [7]. Ein Hamiltonkreis $\pi$ heißt dann *optimal*, wenn die Summe der Kantengewichte minimal wird.

Existiert in $G$ ein Kantenzug $Z$, der alle Kanten enthält und zudem geschlossen ist, dann heißt $Z$ *Eulertour* und $G$ *eulerscher Graph* [6].

Für die Betrachtung von möglichen Lösungsalgorithmen des TSP müssen nun noch abschließend der Begriff des Baums und des Matchings definiert werden:
Ein *Baum* ist ein zusammenhängender, kreisloser Graph. Ein Baum $T$ heißt *Spannbaum* von $G$, wenn er $G$ ganz aufspannt, d.h. wenn $V(T) = V(G)$ ist. Der Spannbaum ist *minimal*, wenn seine Länge minimal ist.

$M$ ist ein *Matching* von $U \subseteq V$, wenn jeder Knoten $U$ mit einer Kante aus $M$ inzidiert.

Auf dieser mathematischen Grundlage lässt sich das TSP nun wie folgt definieren:
$G = (V, E, c)$ sei Graph und $F$ die Menge aller Hamiltonkreise in $G$. Ziel ist nun das Finden des Hamiltonkreises $f \in F$, für den die Summe der einzelnen Kantenkosten minimal wird [8].

Unter der bislang unbewiesenen Annahme, dass die Komplexitätsklassen $P$ und $NP$ verschieden sind, gehört das TSP zur Klasse der NP-vollständigen Probleme, was bedeutet dass es sich nicht mit einem deterministischen Algorithmus in Polynomialzeit lösen lässt [9]. Alle ansatzweise effektiven Möglichkeiten und Algorithmen zur Lösung von TSP-Instanzen mit sehr vielen Städten basieren deshalb auf heuristischen Verfahren. Für Probleme mit weniger Städten gibt effektive Direktlöser, die das optimale Ergebnis in durchaus annehmbarer Zeit liefern.

### 2.2.3. Ansätze und Algorithmen zur Lösung des Problems

Da das Problem wie oben schon erwähnt zu den NP-vollständigen Problemen gehört, ist ein Algorithmus zur Lösung des TSP schnell oder er findet eine optimale Tour - aber er wird nicht beide Eigenschaften besitzen [5]. Deswegen unterscheidet man wie auch allgemein bei anderen kombinatorischen Optimierungsproblemen zwischen exakten und heuristischen Algorithmen. Der einfachste Algorithmus zur exakten Lösung des TSP ist die sogenannte *"brute-force"* oder naive Methode [4]. Dieser Algorithmus betrachtet nacheinander alle möglichen Touren

und deren Länge und ermittelt durch den Vergleich derselben die optimale und kürzeste Tour. Ist der Graph $G$, der dieses Problem modelliert, ein ungerichteter Graph, so muss man mit diesem Algorithmus $\frac{1}{2} \cdot (n-1)!$ verschiedene Rundreisen betrachten. Bei neun verschiedenen Städten ergeben sich daraus 20160 verschiedene Touren, bei 16 Städten dagegen schon ca. 653 Milliarden Routen, dies macht Algorithmus für die meisten Optimierungsprobleme völlig unbrauchbar. Auch andere exakte Methoden haben oft einen hohen Rechen- und Zeitaufwand. Man entscheidet sich deswegen häufig dafür effiziente heuristische Algorithmen zu konstruieren, die zwar nicht immer eine optimale Tour finden, aber zumindest eine nahezu optimale Tour. Es existiert eine Vielzahl solch heuristischer Verfahren Eines der intuitivsten ist der *Nearest-Neighbor-Algorithmus* [10]. Hierbei wird ein zufälliger Knoten $v_1$ als Startknoten ausgewählt. Danach wird iterativ immer der Knoten $v_i$ ausgewählt, der dem zuletzt ausgewählten Knoten am nächsten liegt und noch nicht in der schon besuchten Knotenmenge $V'$ enthalten ist. Der Algorithmus ist beendet, wenn alle Knoten besucht wurden. Das Problem hierbei ist, dass die letzte Kante, die den letzten Knoten mit dem Startknoten verbindet und den Hamiltonkreis vervollständigt, eine mehr oder weniger beliebige Länge haben kann und somit die Bestmöglichkeit der gefundenen Lösung wesentlich einschränkt. Formal lässt sich der Algorithmus wie folgt darstellen:

**Algorithmus 1** Nearest-Neighbor-Algorithm

**Eingabe:** $G = (V, E, c)$
**Ausgabe:** Tour $\pi$, die alle Knoten $v \in V$ besucht

1: $z_1 := v \in V$
2: $V' := V \setminus \{v_1\}$
3: $i := 2$
4: **while** $V' \neq \emptyset$ **do**
5: $\quad z_1 := min_{v \in V'}\, c(\{v_{i-1}, v\})$
6: $\quad V' := V' \setminus \{v_i\}$
7: $\quad i := i + 1$
8: $\pi := (v_1, ..., v_n, v_1)$

Ein weiterer bekannter heuristischer Algorithmus ist die *Minimal-Spaaning-Tree-Heuristik, kurz MST*. Dabei wird zunächst ein minimaler Spannbaum $T$ für den Graphen $G$ ermittelt [11]. Zur Ermittlung dieses minimalen Spannbaumes stehen mehrere Algorithmen zur Verfügung (Algorithmus von Prim, Algorithmus von Kruskal, ...) [12].

**Algorithmus 2** Algorithmus von Prim

**Eingabe:** $G = (V, E, c)$
**Ausgabe:** Minimaler Spannbaum T

1: $T := \emptyset$
2: $r := v \in V$
3: $U := \{r\}$
4: **while** |U| < |V| **do**
5: $\quad$ Finde $u \in U$ und $v \in V - U$ mit $(u, v)$ ist minimal
6: $\quad T := T + \{(u, v)\}$
7: $\quad U := U + \{v\}$

Danach wird jede Kante innerhalb des Spannbaumes verdoppelt; es entsteht der eulersche Graph $G'$. Nun wählt man einen beliebigen Startknoten und folgt den Kanten im Sinne einer

Eulertour. Bereits besuchte Kanten werden dabei gestrichen und stattdessen die direkte Verbindung zwischen den jeweils verbleibenden Knoten gewählt. Das *Verfahren von Christofides* baut auf diesem Algorithmus und erweitert diesen dadurch, dass der minimale Spannbaum nicht verdoppelt wird, sondern das minimale Matching von den Knoten in $B$ sucht, die einen ungeraden Grad haben. Nach der Dreiecksungleichung gilt, dass die Summe der Längen zweier Seiten $a$ und $b$ stets mindestens so groß ist wie die Länge der dritten Seite (formal: $c \leq a+b$). Die MST-Heuristik ist deswegen höchstens doppelt so lang, die Christofides-Heuristik höchstens höchstens 1,5-mal so lang wie die optimale Lösung [11].

### 2.2.4. TSPLIB als Quelle für bekannte Probleme

Die in dieser Arbeit untersuchten Probleminstanzen stammen aus der TSPLIB der Universität Heidelberg. Bekannte Probleme wurden dort in ein einheitliches Datenformat gebracht und eignen sich deswegen gut zur Auswertung von Implementierungen des TSP. Die Daten werden als XML-Datei (Extensible Markup-Language-File) angeboten, was die Anwendung der Lösungsalgorithmik auf viele verschiedene Probleme vereinfacht. [13].

### 2.2.5. Möglichkeiten der Distanzberechnung

Um das TSP lösen zu können, benötigt man Informationen über die Distanzen bzw. die Wegkosten zwischen den einzelnen Knoten. In der TSPLIB (siehe 2.2.4) sind dabei die Kantenwerte schon berechnet worden. Dieser Berechnung hängt vom Format der Positionsinformationen der einzelnen Knoten ab. Die zwei wichtigsten Distanzarten, die auch den Daten in dieser Arbeit zu grunde liegen, sind die *Euklidische Distanz* und die *Geographische Distanz*. Die Euklidische Distanz (auch euklidischer Abstand) ist der triviale Abstand zwischen zwei Punkten den man auch durch Messung mit einem einfachem Längenmessgerät wie einem Lineal ermittelt [14]. Die Distanz $d_{xy}$ der Punkte $x$ und $y$ in einer Ebene mit den Koordinaten $x = (x_1, x_2)$ und $y = (y_1, y_2)$ ergibt sich dabei aus folgender Formel:

$$d_{xy} = \|x - y\|_2 = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2} \tag{1}$$

Die Ermittlung der geographischen Distanz ist dagegen etwas komplizierter. Dies hat die Ursache, dass die Koordinaten auf einer annähernd idealen Kugel liegen. Betrachtet man die Erde als ideale Kugel mit einem Radius $r = 6378,388\,km$ und liegen die Koordinaten in der Dezimalschreibweise (beispielsweise $Lat = 45.345°$, $Long = -3.234°$) vor, ergibt sich für die für zwei Städte $i$ und $j$ folgende Berechnung:

**Algorithmus 3** Geographische Distanz

1: $r = 6378,388$
2: $latitude_i = latitude_i \cdot \pi/180$; $latitude_j = latitude_j \cdot \pi/180$
3: $longitude_i = longitude_i \cdot \pi/180$; $longitude_j = longitude_j \cdot \pi/180$
4: $q_1 = cos(longitude_i - longitude_j)$
5: $q_2 = cos(latitude_i - latitude_j)$
6: $q_3 = cos(latitude_i + latitude_j)$
7: $d_{ij} = r * arccos(0,5 \cdot ((1 + q_1) \cdot q_2 - (1 - q_1) \cdot q_3)) + 1)$

## 2.3. Das Travelling-Salesman-Problem und der Ameisenalgorithmus

Betrachtet man den Ameisenalgorithmus, so ähnelt dies dem TSP sehr stark. Dies war auch der Grund, warum der Ameisenalgorithmus zuerst als Lösung in Form des *Ant-Systems (AS)* für das TSP zur Anwendung kam [15].
Es bietet sich an, dass TSP wie in Abschnitt 2.2.2 schon näher erläutert als Graphenproblem zu modellieren: Hat man ein TSP mit $n$ Städten, so benötig man neben einer $n \times n$-Matrix $D$ mit den verschiedenen Distanzen zwischen den Knoten auch eine $n \times n$-Matrix $T$ mit den verschiedenen Pheromonkonzentrationen. Dabei ist das Element $\tau_{ij}$ die Pheromonkonzentration auf der Kante zwischen Knoten $i$ und $j$, analog ist die Distanz $d_{ij}$ die Distanz zwischen den Knoten $i$ und $j$. Ist das verwendete TSP symmetrisch, dann ist auch die Pheromonspur symmetrisch; das heißt: $\tau_{ij} = \tau_{ji}$. Abstrakt gesehen werden dann zunächst alle $m$ Ameisen einer Menge $M$ auf verschiedene zufällig gewählte Knoten gesetzt. Danach bewegt sich jede Ameise $m_k$ in jedem darauffolgenden Iterationsschritt zu einem weiteren Knoten, falls sie diesen vorher noch nicht besucht hat und insgesamt nicht alle Knoten schon besucht wurden.

Der Entscheidung, welcher Knoten als nächstes besucht wird, liegen gewisse Regeln zugrunde. Zum einen hat die Pheromonkonzentration der jeweiligen Kanten zu den anderen Knoten Einfluss auf die Entscheidung, zum anderen die heuristische Information $\eta_{ij}$. Die heuristische Information ergibt sich dabei aus der Distanz:

$$\eta_{ij} = {}^{1}/_{d_{ij}} \tag{2}$$

Bei der Wahl des nächsten Knotens bevorzugen die Ameisen solche Knoten, die relativ nah gelegen sind und die durch eine Kante mit hoher Pheromonkonzentration verbunden sind. Die Wahrscheinlichkeit $p_{ij}^k$, mit der eine Ameise $k \in M$ ausgehend von eines Knotens $i$ den nächsten Knoten $j$ besucht, kann durch die Gleichung 3 ausgedrückt werden. Dabei sind $\alpha$ und $\beta$ Parameter, die im Fall von $\alpha$ die Wichtigkeit der Pheromonkonzentration und im Fall von $\beta$ die Wichtigkeit der Distanzinformationen zur Entscheidungsfindung angeben. $N_i^k$ ist die Menge der erreichbaren unbesuchten Knoten der Ameise $m_k$.

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha \, [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha \, [\eta_{il}]^\beta} \quad \text{if } j \in N_i^k \tag{3}$$

Im Rahmen dieser Arbeit wurde zusätzlich eine vereinfachte Formel implementiert die eine reduzierte Form dieser Gleichung darstellt:

$$p_{ij}^k = [\tau_{ij}]^\alpha \, [\eta_{ij}]^\beta \tag{4}$$

Jede Ameise merkt sich zudem die Reihenfolge der besuchten Knoten in einer Liste $L$. Falls eine Ameise $m_k$ alle Knoten $n$ besucht hat, kehrt sie zu ihrem Anfangsknoten zurück und beendet ihre Tour. Anhand der Liste mit den besuchten Knoten lässt sich nun ein valider Hamiltonkreis ableiten.
Außerdem ist diese Liste notwendig, um die Pheromonkonzentration der benutzten Kanten zu aktualisieren.
Für ein Pheromonupdate gibt es verschiedene Möglichkeiten, die für 2., 3. und 4. im Rahmen dieser Bachelorarbeit auch umgesetzt wurden:

1. Aktualisierung der Pheromonkonzentration nachdem alle Ameisen $m \in M$ ihre Tour beendet haben

2. Aktualisierung der Pheromonkonzentration iterativ nach der erfolgreichen Tourkonstruktion jeder einzelnen Ameise $m_k$
3. Aktualisierung der Pheromonkonzentration nur durch die Ameise $m_k$, welche nach Beendigung der Tour die kürzeste Tour gefunden hat
4. Aktualisierung der Pheromonkonzentration für alle Ameisen $m \in M$ parallel, nachdem der jeweils nächste Knoten erreicht wurde

Die Berechnung der neuen Pheromonkonzentration $\tau_{ij}$ zum Zeitpunkt $t+1$ ist für alle Varianten gleich. Eine wichtige Variable, die hierauf Einfluss hat, ist zum einen $Q$. Diese stellt dabei die Menge an Pheromon dar, die eine Ameise $k$ auf der jeweilige Kante $\{i, j\}$ ablegt. Zum anderen ist das der Parameter für die Pheromonpersistenz $\rho$, der für $1 - \rho$ die verdunstende Menge an Pheromon angibt. Dieser Verdunstungsmechanismus hilft über die Zeiten ungünstige Kanten zu „vergessen“. Letztendlich berechnet sich die neue Pheromonkonzentration wie folgt:

$$\tau_{ij}(t+1) = \rho\,\tau_{ij}(t) + \sum_{k=1}^{m} Q \tag{5}$$

# 3. Implementierung des Problems in C++

## 3.1. Programmstruktur und Funktionsweise

Das Programm lässt sich grob gesehen in zwei Komponenten aufteilen - zum einen in eine C++ Anwendung auf Konsolenbasis, welche den Algorithmus an sich ausführt. Die andere Komponononente ist ein in C++/CLI umgesetzte einfache GUI. Wesentlich für das Verständnis der Implementierung ist lediglich die Programmstruktur und der Code der nativen C++-Konsolenanwendung.

Wie in Abbildung 2 ersichtlich ist, besteht das Programm aus vier verschiedenen Klassen. Von grundlegender Bedeutung ist hierbei die Klasse `Ant`. Diese stellt die einzelnen Ameisen mit allen wichtigen Parametern und Methoden zur Wegfindung und Routenfindung dar. Jedes dieser `Ant`-Objekte greift auf ein gemeinsames Datenelement `XMLData` zu, in welchem die Daten der Städte sowie die Kosten- und Pheromonwerte zwischen den einzelnen Städten gespeichert sind. Die Daten der Städte werden in einen Container aus `City`-Objekten gespeichert. Ein solches `City`-Objekt besteht aus dem Namen der Stadt und den Koordinaten. Im Falle der TSPLIB ist die Distanzberechnung aus den Koordinaten überflüssig, da die Distanzwerte in der XML-Datei enthalten sind. Es existiert jedoch eine Methode `measureDistance` zur Berechnung der Distanz zwischen zwei Städten falls die Datendatei nur die Koordinaten liefert. Bei Anlage eines `XMLData`-Objekts werden die eingelesenen Distanzwerte in einer Distanzmatrix abgespeichert sowie eine Matrix für die Pheromonwerte initialisiert.
Desweiteren besitzt die Klasse `Ant` ein Memberobjekt der Klasse `Route`, in der die Abfolge der besuchten Städte gespeichert wird.

Beim Start des Programms wird ein dynamischer Container vom Typ `vector<Ant>` angelegt, in dem sich alle `Ant`-Objekte befinden. Die weitere Arbeitsweise des Programms hängt nun von dem ausgewählten Algorithmus der Pheromonaktualisierung ab ab. Bevor näher auf diese schon im Abschnitt 2.3 erwähnten drei implementierten Algorithmen eingegangen wird, wird im folgenden zunächst der Algorithmus zur Ermittlung der jeweils nächsten Stadt während der Tourkonstruktion diskutiert.

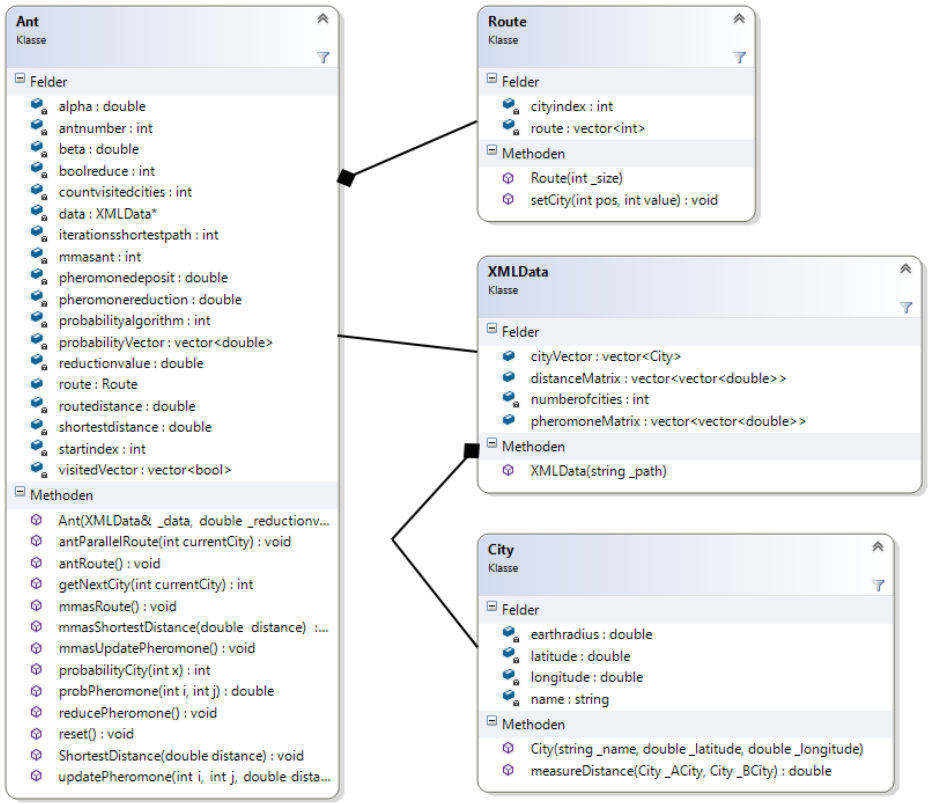



**Abb. 2**: Klassendiagramm zur Umsetzung des C++-Algorithmus mit den wesentlichen Attributen und Methoden

### 3.1.1. Wahrscheinlichkeitsalgorithmus zum Finden der nächsten Stadt

Während der Tourkonstruktion einer Ameise $m$ wird beim Erreichen jeder neuen Stadt die Wahrscheinlichkeiten für die restlichen Städte ermittelt. Dazu wird ein `vector` $P$ angelegt, welcher der Anzahl der Städte entspricht und diese Wahrscheinlichkeiten enthält. Dann wird für alle Städte, die noch nicht besucht wurden, iterativ die Besuchswahrscheinlichkeit berechnet und in $P$ gespeichert. Die Berechnung der Wahrscheinlichkeit erfolgt durch die in Abschnitt 2.3 definierte Formel 3 bzw. 4. Am Ende wird die Stadt $v_o = v_i$ besucht, welche die größte Wahrscheinlichkeit $p_i$ besitzt.
Dieser eigens implementierte Wahrscheinlichkeitsalgorithmus wird von im folgenden von allen drei Algorithmen zur Konstruktion der Tour verwendet.

### 3.1.2. Iterative Tourkonstruktion

Bei der ersten implementierten Methode wird die Pheromonaktualisierung iterativ nach jeder erfolgreichen Tourkonstruktion einer Ameise $m$ vorgenommen. Die nachfolgenden Ameisen können somit für ihre Tourkonstruktion schon die aktualisierten Pheromondaten der vorangegangenen Ameisen nutzen. Zu Beginn des Algorithmus wird eine Menge $M$ von Ameisen definiert.
Jede Ameise $m \in M$ startet zunächst in einer zufällig ausgewählten Stadt $v_0$ und besucht dann nacheinander alle Städte. Welche Stadt dabei als nächstes besucht wird, ergibt sich aus dem oben vorgestellten Wahrscheinlichkeitsalgorithmus. Jeder Index einer besuchten Stadt $v_i$ wird in einen Routenvector $r$ geschrieben. Hat eine Ameise ihre Tourkonstruktion abgeschlos-

**Algorithmus 4** Ermittlung der nächsten Stadt während der Tourkonstruktion einer Ameise

**Eingabe:** Datenobjekt mit $v$ Städten sowie einer Distanzmatrix $D$ und Pheromonmatrix $S$, Ameise $m \in M$ mit einem `vector` $B$ der Indizes aller schon besuchten Städte, Index $v_k$ der aktuellen Stadt

**Ausgabe:** Index der nächstbesten Stadt $v_o$

1: $v_o := -1$
2: $P := \{p_0, ..., p_n\}$ mit $n = v$
3: **for each** Stadt $v_i$ in V **do**
4: **if** $v_i \notin B$ and $v_i \neq v_k$ **then**
5: $p_i$ den Rückgabewert der Wahrscheinlichkeitsformel zuweisen
6: **else**
7: $p_i := 0$
8: $v_o := v_i$ mit $max(p_i \in P)$

sen, startet die nächste Ameise ihre Tour.
Der gesamte Algorithmus endet dann, wenn alle Ameisen in $M$ ihre Tour beendet haben oder wenn nach einer bestimmten Anzahl von Iterationen keine kürzere Route mehr gefunden wurde. Diese Iterationsschwelle $n$ wird ebenso zu Beginn festgelegt.

**Algorithmus 5** Iterative Tourkonstruktion

**Eingabe:** Datenobjekt mit $v$ Städten sowie einer Distanzmatrix $D$ und Pheromonmatrix $S$, `vector` $M$ mit $m$ Ameisen, Iterationsschwelle $n$

**Ausgabe:** Route $r$ mit der kürzesten gefunden Distanz $d_s$

1: $j := 1$ (derzeitige Iteration nachdem das letzte Mal eine kürzere Route gefunden wurde)
2: $d_s := \infty$ (kürzeste gefunden Routenlänge)
3: **for each** Ameise $m_i \in M$ **do**
4: **if** $j \leq n$ **then**
5: $b_m := 1$ (Anzahl der besuchten Städte)
6: $d_m := 0$ (Routenlänge der Ameise $m_i$)
7: Starte in einer zufälligen Stadt $v_0$
8: **while** $b \neq v$ **do**
9: Ermittle die nächste Stadt $v_i$ und gehe dorthin
10: $r_{m_i} := v_i$
11: $d_m := d_m + D_{i-1,i}$
12: $b_m := b_m + 1$
13: **if** $d_m \leq d_s$ **then**
14: $d_s := d_m$
15: $j := 1$
16: Aktualisiere Pheromonmatrix $S$
17: $j := j + 1$

### 3.1.3. Iterative Tourkonstruktion mit eingeschränkter Pheromonaktualisierung

Die zweite implementierte Methode ist der ersten sehr ähnlich. Auch hier konstruieren alle Ameisen nacheinander iterativ ihre Tour. Jedoch wird die Pheromonmatrix nur durch die Ameise aktualisiert, welche die kürzeste Route gefunden hat. Diese Vorgehensweise basiert auf

dem *Max-Min-Ant-System* von Thomas Stüzle und Holger H. Hoos [15]. Die iterative Tourkonstruktion erfolgt bei diesem Algorithmus notwendigerweise mehrmals. Diese Iterationsanzahl $n$ wird vor dem Start des Programms festgelegt.

**Algorithmus 6** Iterative Tourkonstruktion mit eingeschränkter Pheromonaktualisierung

**Eingabe:** Datenobjekt mit $v$ Städten sowie einer Distanzmatrix $D$ und Pheromonmatrix $S$, `vector` $M$ mit $m$ Ameisen, Iterationsanzahl $n$
**Ausgabe:** Route $r$ mit der kürzesten gefunden Distanz $d_s$

1: $j := 1$ (Derzeitige Iteration)
2: $d_s := \infty$ (Kürzeste Routenlänge der Iteration $i$)
3: $m_o := -1$ (Index der Ameise mit der kürzesten Route)
4: **while** $j \leq n$ **do**
5: **for each** Ameise $m_i \in M$ **do**
6: $b_m := 1$ (Anzahl der besuchten Städte)
7: $d_m := 0$ (Routenlänge der Ameise $m_i$)
8: Starte in einer zufälligen Stadt $v_0$
9: **while** $b \neq v$ **do**
10: Ermittle die nächste Stadt $v_i$ und gehe dorthin
11: $r_{m_i} := v_i$
12: $d_m := d_m + D_{i-1,i}$
13: $b_m := b_m + 1$
14: **if** $d_m \leq d_s$ **then**
15: $d_s := d_m$
16: $m_o := m_i$
17: Ameise $m_o$ aktualisiert Pheromonmatrix $S$
18: $j := j + 1$

### 3.1.4. Parallele Tourkonstruktion

Bei der dritten in dieser Arbeit betrachteteten Methode konstruieren die Ameisen ihre Route parallel. Dabei startet jede Ameise in einer zufälligen Stadt $v_0$. Nun ermittelt jede Ameise $m$ zunächst die erste Stadt $v_1$ und aktualisiert die Pheromonkonzentration auf der begangenen Kante. Dieses Vorgehen geschieht analog für jeden weitere Stadt $v_i$. Wenn jede Ameise jede Stadt einmal besucht hat, endet der Algorithmus. Die kürzeste Route ergibt sich nun durch den Vergleich der Routenlänge aller Ameisen.

Alle in den Abschnitten 3.1.1 bis 3.1.4 umgesetzten Algorithmen wurden für diese Arbeit entwickelt.

## 3.2. Vorgehensweise zur Untersuchung von verschiedenen Datensätzen mit verschiedenen Implementierungen

Im Rahmen dieser Arbeit werden zur Untersuchung der verschiedenen Algorithmen zur Lösung des TSP der TSPLIB fünf Probleminstanzen entnommen: `burma14`, `dantzig42`, `gr120` und `si535` als symmetrische Probleminstanzen und `ft53` als asymmetrische Probleminstanz. Die unterschiedlichen Größen der gewählten Instanzen dienen einer besseren Beurteilung der Effektivität und Laufzeit des Programms. Der Begriff *Effektivität* bezieht sich dabei im folgenden immer auf die Eigenschaft, im Bezug auf die Tourlänge möglichst nah an das Optimum

**Algorithmus 7** Parallele Tourkonstruktion

**Eingabe:** Datenobjekt mit $v$ Städten sowie einer Distanzmatrix $D$ und Pheromonmatrix $S$, `vector` $M$ mit $m$ Ameisen

**Ausgabe:** Route $r$ mit der kürzesten gefunden Distanz $d_s$

1: $j := 0$
2: **while** $j < v$ **do**
3: **for each** Ameise $m_i \in M$ **do** } Parallel
4: Starte in einer zufälligen Stadt $v_0$
5: Ermittle die nächste Stadt $v_i$ und gehe dorthin
6: $r_{m_i} := v_i$
7: $d_m := d_m + D_{i-1,i}$
8: Aktualisiere Pheromonmatrix $S$
9: $j := j + 1$
10: $d_s = \infty$
11: **for each** Ameise $m_i \in M$ **do**
12: **if** Tourlänge $d_m < d_s$ **then**
13: $d_s := d_m$

heranzukommen. Sowohl die Laufzeit als auch die Effektivität hängen von mehreren Parametern ab:

1. Parameter $\alpha$, der die Wichtigkeit der Pheromonkonzentration auf einer Kante bei der Wahl der nächsten zu bereisenden Stadt festlegt

2. Parameter $\beta$, der die Wichtigkeit der Distanz auf einer Kante bei der Wahl der nächsten zu bereisenden Stadt bestimmt

3. Parameter $Q$, der die Menge des abgelegten Pheromons darstellt

4. Parameter $\rho$, der den Wert der Pheromonpersistenz angibt; $1 - \rho$ steht für die Pheromonverdunstung

Im Rahmen der Untersuchung und Auswertung wird anhand der `dantzig42`-Instanz die Auswirkungen der oben genannten Parameter betrachtet und die Ergebnisse für verschiedene Werte erfasst. Dies geschieht für alle drei umgesetzten Algorithmen unter Berücksichtigung der verschiedenen Implementierungen der Ermittlung der Wahrscheinlichkeit $p_{ij}^k$ (Abschnitt 3.1). Die dadurch gewonnenen Werte für die Parameter werden hinsichtlich der Effektivität verglichen. Die besten Parameterwerte werden dann für die Untersuchung der verbleibenden vier Probleminstanzen verwendet.
Zur besseren Analyse werden alle Daten automatisch in eine CSV-Datei geschrieben und von dort aus ausgewertet.

# 4. Ergebnisse der Durchführung

Um die Laufzeit eines Problems zu messen wird die C++ Bibliothek `chrono` verwendet. Dabei wird die Systemzeit bei Beginn und Ende der Routenberechnung gemessen. Die Laufzeit ergibt sich dann aus der Differenz beider Werte.
Die Effektivität ergibt sich aus der Abweichung zum Tourlängen-Optimum. Dabei wird der maximale Abstand und der Durchschnitt in mehreren Programmdurchläufen erfasst und prozentual ermittelt.

## 4.1. Resultate bei Iteration einzelner Ameisen nacheinander

## 4.2. Resultate bei paralleler Erschließung

# 5. Auswertung und Vergleich

## 5.1. Vergleich der iterativen und parallelen Implementierung

## 5.2. Vergleich mit der optimalen Lösung

## 5.3. Vergleich mit der Laufzeit und Komplexität von anderen Algorithmen

# 6. Zusammenfassung und Fazit

# A. Erklärung der implementierten grafischen Oberfläche

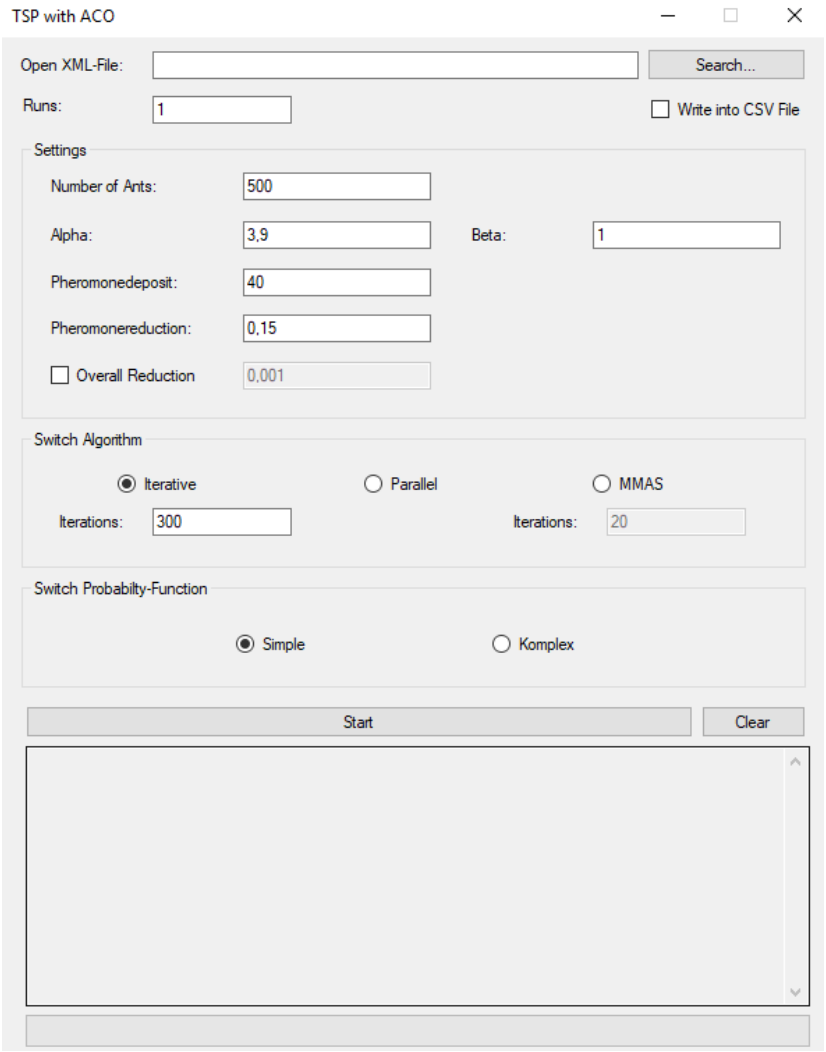



**Abb. 3**: Grafische Oberfläche des Programms

Der Button **Search...** öffnet einen neuen Filedialog, mit dem die gewünschte Probleminstanz im XML-Format geladen werden kann. Alternativ kann der Dateipfad auch in das nebenstehende Textfeld eingegeben werden.

Der Paramater im Textfeld **Runs** gibt an, wie oft das gesamte Programm ausgeführt werden soll. Dies ist hilfreich bei der Ermittlung von Durchschnittswerten für einzelne festgelegte Parameter.

Unter **Settings** befinden sich alle konfigurierbaren Parameter, die Einfluss auf den Algorithmus haben.
**Number of Ants** gibt die Anzahl der Ameisen an, welche die Route konstruieren. **Alpha** steht für die Wichtigkeit der Pheromonspur, **Beta** für die Wichtigkeit der Distanz bei der Ermittlung der nächsten Stadt im Verlauf der Tourkonstruktion. Der Parameter **Pheromondeposit** gibt die Menge des ablegten Pheromons an, während **Pheromonereduction** für die

Verdunstungsmenge steht.
Es existiert die Möglichkeit, abhängig vom ausgewählten Algorithmus eine globale Pheromonreduktion durchzuführen, die durch den Parameter **Overall Reduction** festgelegt wird.

Unter **Switch Algorithm** kann der gewünschte Algorithmus zur Berechnung der kürzesten Route gewählt werden. **Iterations** gibt dabei für den iterativen Algorithmus die Anzahl von Iterationen an, nach denen der Algorithmus abbricht, wenn keine kürzere Route gefunden wurde. Analag gibt er für die Option **MMAS** an, wie oft alle Ameisen ihre Tour konstruieren.

Mit **Switch Probability-Function** kann zwischen einem einfachen und einem komplexeren Algorithmus zur Ermittlung der nächsten jeweiligen Stadt bei der Erstellung der Route ausgewählt werden.

Ein Klick auf den Button **Start** startet den Algorithmus. Die Ausgabe erfolgt in das untenstehende Textfeld, welches mit dem Button **Clear** geleert werden kann.

Möchte man zudem eine Ausgabe in eine Tabellendatei erhalten, muss oben rechts die Funktion **Write into CSV File** aktiviert werden.

# Literatur

[1] Christian Dietrich and Erich Steiner. Das Leben unserer Ameisen - ein Überblick. *Biologiezentrum Linz/Austria*.

[2] J.-L. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of lnsect Behavior, Vol. 3, No. 2*, 1990.

[3] Marco Dorigo. Ant colony optimization. *Scholarpedia*, 2(3):1461, 2007. revision #90969.

[4] Berthold Vöcking, Helmut Alt, Martin Dietzfelbinger, Rüdiger Reischuk, Christian Scheideler, Heribert Vollmer, and Dorothea Wagner. *Taschenbuch der Algorithmen*, chapter Das Travelling Salesman Problem, pages 413–422. Springer-Verlag Berlin Heidelberg, 2008.

[5] E.L. Lawler, J.K.Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem - A Guided Tour of Combinatorial Optimization*. John Wiley and Sons Ltd., 1985.

[6] Arnfried Kemnitz. *Graphentheorie*. 2000.

[7] Reinhard Diestel. *Graphentheorie*. Springer-Verlag Heidelberg, 2006.

[8] G. Gutin and A. P. Punnen. *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers, 2002.

[9] David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The traveling salesman problem: a computational study*. Princeton University Press, 2007.

[10] S. Lotz. Lösung und graphische Darstellung des Traveling Salesman Problems in einer Webapplikation. 2014.

[11] M. Grötschel. Schnelle Rundreisen: Das Travelling Salesman-Problem. *ZIB-Report 05-57*, 2005.

[12] Bang Ye Wu and Kun-Mao Chao. *Spanning Trees and Optimization Problems*. Chapman & Hall/CRC, 2004.

[13] Gerhard Reinelt. TSPLIB - A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 1991.

[14] Elena Deza and Michel Marie Deza. *Encyclopedia of Distances*. Springer-Verlag Berlin Heidelberg, 2009.

[15] Thomas Stützle and Holger H. Hoos. MAX-MIN Ant System. *Future Generation Computer Systems*, 2000.