



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

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Diese Versicherung bezieht sich auch auf die bildlichen Darstellungen.

05. November 2018

Samuel Dressel

Inhaltsverzeichnis

1. Einleitung	4
2. Grundlagen	4
2.1. Der Ameisenalgorithmus (Ant Colony Optimization)	4
2.1.1. Biologische Grundlagen	4
2.1.2. Der Ameisenalgorithmus	4
2.2. Das Travelling-Salesman-Problem	5
2.2.1. Das Problem im Allgemeinen	5
2.2.2. Graphentheoretische Grundlagen	5
2.2.3. Ansätze und Algorithmen zur Lösung des Problems	6
2.2.4. TSPLIB als Quelle für bekannte Probleme	8
2.2.5. Möglichkeiten der Distanzberechnung	8
2.3. Das Travelling-Salesman-Problem und der Ameisenalgorithmus	8
3. Implementierung des Problems in C++	11
3.1. Programmstruktur und Funktionsweise	11
3.1.1. Wahrscheinlichkeitsalgorithmus zum Finden der nächsten Stadt	12
3.1.2. Iterative Tourkonstruktion	13
3.1.3. Iterative Tourkonstruktion mit eingeschränkter Pheromonaktualisierung	14
3.1.4. Parallele Tourkonstruktion	15
3.2. Vorgehensweise zur Untersuchung von verschiedenen Datensätzen mit verschiedenen Implementierungen	16
4. Ergebnisse der Durchführung	18
4.1. Einfluss der Parameter auf die implementierter Algorithmen	18
4.1.1. Parametereinfluss auf den iterativen Algorithmus	18
4.1.2. Parametereinfluss auf den iterativen Algorithmus mit eingeschränkter Pheromonaktualisierung	19
4.1.3. Parametereinfluss auf den parallelen Algorithmus	20
4.2. Auswirkung des gewählten Wahrscheinlichkeitsalgorithmus	20
4.3. Ergebnisse der Algorithmen bei verschiedenen TSP-Probleminstanzen	22
5. Auswertung und Vergleich	24
5.1. Vergleich der iterativen und parallelen Implementierung	24
5.2. Vergleich mit der optimalen Lösung	24
5.3. Vergleich mit der Laufzeit und Komplexität von anderen Algorithmen	24
6. Zusammenfassung und Fazit	24
A. Erklärung der implementierten grafischen Oberfläche	25
B. Hinweise zur Nutzung des Programms als Konsolenapplikation	27
Literatur	28

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 kristallisiert 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.

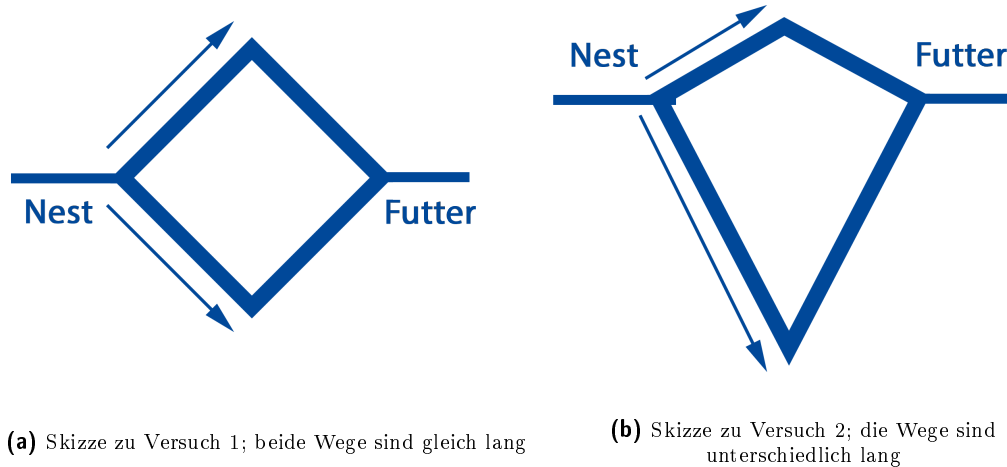


Abb. 1: Double-Bridge-Experiment nach Deneubourg [2]

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 $V = \{1, \dots, n\}$ 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 Kno-

ten 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* $W = (v_0v_1, v_1v_2, \dots, v_{r-1}v_r)$ 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*.

Enthält 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* π (auch *Tour*) von G [7]. Ein Hamiltonkreis π 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 π , 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:    $z_i := \min_{v \in V'} c(\{v_{i-1}, v\})$ 
6:    $V' := V' \setminus \{v_i\}$ 
7:    $i := i + 1$ 
8:  $\pi := (v_1, \dots, v_n, v_1)$ 

```

Ein weiterer bekannter heuristischer Algorithmus ist die *Minimal-Spanning-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:   Finde  $u \in U$  und  $v \in V - U$  mit  $(u, v)$  ist minimal
6:    $T := T + \{(u, v)\}$ 
7:    $U := U + \{v\}$ 

```

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 Ver-

bindung 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:

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

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 $r = 6378,388 \text{ km}$ und liegen die Koordinaten in der Dezimalschreibweise (beispielsweise $Lat = 45.345^\circ$, $Long = -3.234^\circ$) 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 benötigt 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 τ_{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 η_{ij} . Die heuristische Information ergibt sich dabei aus der Distanz:

$$\eta_{ij} = 1/d_{ij} \quad (2)$$

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 α und β Parameter, die im Fall von α die Wichtigkeit der Pheromonkonzentration und im Fall von β die Wichtigkeit der Distanzinformationen zur Entscheidungsfindung angeben. N_i^k ist die Menge der erreichbaren unbesuchten Knoten der Ameise m_k .

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [1/\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [1/\eta_{il}]^\beta} \quad \text{if } j \in N_i^k \quad (3)$$

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

$$p_{ij}^k = [\tau_{ij}]^\alpha [1/\eta_{ij}]^\beta \quad (4)$$

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 τ_{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 jeweiligen Kante $\{i, j\}$ ablegt. Zum anderen ist das der Parameter für die Pheromonpersistenz ρ , 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:

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

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 Komponente 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.

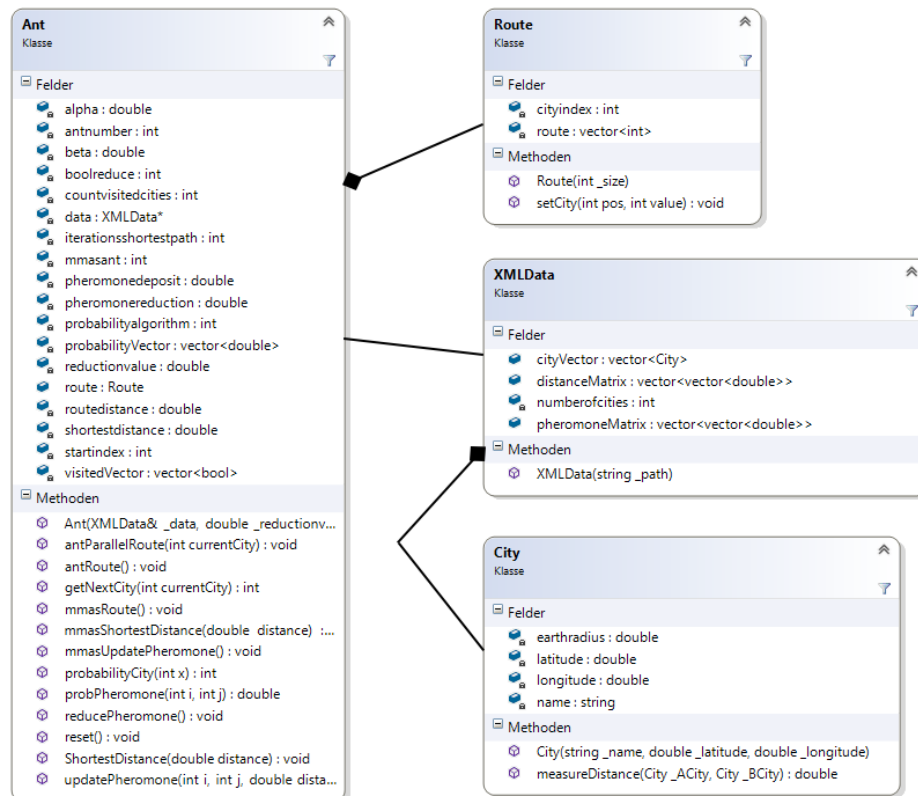


Abb. 2: Klassendiagramm zur Umsetzung des C++-Algorithmus mit den wesentlichen Attributen und Methoden

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. 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.

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

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, dessen Größe 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 $v_{opt} = v_i$ 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.

Die untenstehende Abbildung 3 zeigt ein Beispiel für diesen Algorithmus.

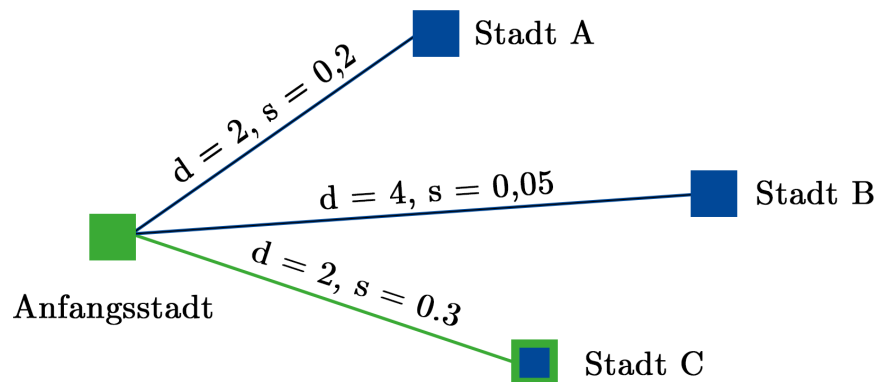


Abb. 3: Beispiel für den implementierten Wahrscheinlichkeitsalgorithmus. Die Ameise m ermittelt anhand der Parametern $\alpha = 0,9$ und $\beta = 0,3$ und der gegebenen Daten für d und s die jeweilige Wahrscheinlichkeit p und reist dann in die Stadt mit dem höchsten Wert für p ; in diesem Fall die Stadt C.

Die Ameise m befindet sich in der als *Anfangsstadt* bezeichneten Stadt. Sie hat nun die Möglichkeit in eine von drei Städten zu reisen. Sie ermittelt nun die Besuchswahrscheinlichkeiten von allen drei Städten. Dazu dienen die Angaben d für die Distanz und s für den Pheromonwert als Grundlage.

Wird ein Parameter Alpha α von 0,9 und ein Parameter Beta β mit 0,3 gewählt, ergeben sich durch nutzen von Formel 4 folgende Wahrscheinlichkeiten p_i :

$$\begin{aligned} p_{StadtA} &= [\tau_{ij}]^\alpha [1/\eta_{ij}]^\beta = 0,2^{0,9} \cdot (1/2)^{0,3} = 0,191 \\ p_{StadtB} &= [\tau_{ij}]^\alpha [1/\eta_{ij}]^\beta = 0,05^{0,9} \cdot (1/4)^{0,3} = 0,045 \\ p_{StadtC} &= [\tau_{ij}]^\alpha [1/\eta_{ij}]^\beta = 0,3^{0,9} \cdot (1/2)^{0,3} = 0,356 \end{aligned}$$

Die Ameise reist nun in die Stadt, welche den größten Wert für p besitzt, dies ist in unserem Beispiel die Stadt C mit $p_{StadtC} = 0,356$.

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:  $v_{opt} := -1$ 
2:  $P := \{p_0, \dots, p_n\}$  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:  $v_{opt} := v_i$  mit  $\max(p_i \in P)$ 
```

3.1.2. Iterative Tourkonstruktion

Bei der ersten implementierten Methode (Algorithmus 5) 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. Der Algorithmus beinhaltet eine Abbruchbedingung: Wurde in n Iterationen keine kürzere Route gefunden, bricht der Algorithmus ab. Diese Iterationsschwelle n wird dem Algorithmus als Parameter übergeben. Vor dem Beginn der Tourkonstruktion wird zum einen eine Hilfsvariable j definiert und mit 0 initialisiert, welche für die derzeitige Iteration steht. Zum anderen wird die insgesamt kürzeste gefundene Routenlänge d_s mit ∞ initialisiert.

Dann startet die Tourkonstruktion. Jede Ameise $m \in M$ startet zunächst in einer zufällig ausgewählten Stadt v_0 . Diese Stadt stellt auch die erste Stadt r_{m_0} der Route r dar. An diese Stelle wird eine zweite Hilfsvariable b hinzugefügt, welche die Anzahl der besuchten Städte zählt. Solange diese Variable b nicht der Anzahl der Städte v entspricht, wird nun die Tour iterativ konstruiert. Welche Stadt v_b dabei als nächstes besucht wird, ergibt sich aus dem oben vorgestellten Wahrscheinlichkeitsalgorithmus. Jeder Index einer besuchten Stadt v_b wird in einen Routenvector r geschrieben; die Distanz d_m wird beim Erreichen einer neuen Stadt mit der Länge der bereisten Kante erweitert und b wird um 1 erhöht. Hat eine Ameise alle Städte besucht ($b = v$), wird die Tourlänge mit der Distanz von der letzten Stadt r_{m_b} zur ersten Stadt r_{m_0} erweitert. Nun wird die gefundene Routenlänge d_m mit der insgesamt kürzesten Routenlänge d_s verglichen. Ist d_m kleiner als d_s , wird d_s mit d_m ersetzt und der Wert der derzeitigen Iteration j wird wieder auf 0 gesetzt. Abschließend aktualisiert die Ameise m die

Pheromonmatrix S und j wird um 1 erhöht. Hat eine Ameise ihre Tourkonstruktion abgeschlossen, 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 n von Iterationen keine kürzere Route mehr gefunden wurde.

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:  $j := 0$  (derzeitige Iteration nachdem das letzte Mal eine kürzere Route gefunden wurde)
2:  $d_s := \infty$  (kürzeste gefundene Routenlänge)
3: for each Ameise  $m \in M$  do
4:   if  $j \leq n$  then
5:     Starte in einer zufälligen Stadt  $v_0$ 
6:      $r_{m_0} := v_0$ 
7:      $b := 1$  (Anzahl der besuchten Städte)
8:     while  $b \neq v$  do
9:       Ermittle die nächste Stadt  $v_b$  und gehe dorthin
10:       $r_{m_b} := v_b$ 
11:       $d_m := d_m + D_{r_{m_{b-1}}, r_{m_b}}$ 
12:       $b := b + 1$ 
13:       $d_m := d_m + D_{r_{m_b}, r_{m_0}}$ 
14:      if  $d_m \leq d_s$  then
15:         $d_s := d_m$ 
16:         $j := 0$ 
17:      Aktualisiere Pheromonmatrix  $S$ 
18:       $j := j + 1$ 

```

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 Stützle und Holger H. Hoos [15]. Die gesamte Tourkonstruktion erfolgt bei diesem Algorithmus notwendigerweise mehrmals.

Wie bei obigen Algorithmus der normalen iterativen Tourkonstruktion wird auch hier eine Hilfsvariable j mit 0 initialisiert, welche im weiteren Verlauf des Algorithmus für die aktuelle Iteration steht. Weiterhin wird die Variable d_s für die kürzeste gefundene Distanz mit ∞ initialisiert. Zusätzlich wird eine Variable m_{opt} eingeführt, welche den Index der Ameise mit der kürzesten Route jedes Iterationsschritts enthält. Dieser wird zunächst der Wert -1 zugewiesen.

Bei der nun beginnenden Tourkonstruktion konstruieren alle Ameisen iterativ $am \in M$ ihre Tour. Jede Ameise beginnt in einer Stadt v_0 . Diese Stadt ist gleichzeitig auch die erste Stadt der Route r . Ein Zählvariable b zählt die Anzahl der besuchten Städte. Solange $b \neq v$ gilt, also die jeweilige Ameise nicht alle Städte v besucht hat, ermittelt sie die nächste Stadt v_b . Die Ermittlung der jeweils nächsten Stadt erfolgt mit dem oben vorgestellten Wahrscheinlichkeitsalgorithmus (siehe Algorithmus 4). Nach der Reise in die nächste Stadt wird der Zähler

b um 1 erhöht und die bisherige Routenlänge d_m entsprechend erhöht.

Hat eine Ameise alle Städte besucht, reist sie wieder in ihre Anfangsstadt zurück und auch die damit bereiste Kante wird zu der Routenlänge d_m addiert. Abschließend wird diese Länge d_m mit der insgesamt kürzesten Routenlänge d_s verglichen. Wurde mit d_m eine kürzere Tour gefunden, so wird d_s mit dieser ersetzt und m_{opt} enthält den Index dieser Ameise.

Wurden alle Touren in einem Iterationsschritt konstruiert, aktualisiert die Ameise m_{opt} die Pheromonmatrix S und der Iterationszähler j wird um 1 erhöht.

Dieser gesamte Vorgang wird n -mal wiederholt, d.h. wenn $j \leq n$ gilt. 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 gefundenen Distanz d_s

```

1:  $j := 0$  (Derzeitige Iteration)
2:  $d_s := \infty$  (Kürzeste gefundene Routenlänge)
3:  $m_{opt} := -1$  (Index der Ameise mit der kürzesten Route)
4: while  $j \leq n$  do
5:   for each Ameise  $m \in M$  do
6:     Starte in einer zufälligen Stadt  $v_0$ 
7:      $r_{m_0} := v_0$ 
8:      $b := 1$  (Anzahl der besuchten Städte)
9:     while  $b \neq v$  do
10:      Ermittle die nächste Stadt  $v_b$  und gehe dorthin
11:       $r_{m_b} := v_b$ 
12:       $d_m := d_m + D_{r_{m_{b-1}}, r_{m_b}}$ 
13:       $b := b + 1$ 
14:       $d_m := d_m + D_{r_{m_b}, r_{m_0}}$ 
15:      if  $d_m \leq d_s$  then
16:         $d_s := d_m$ 
17:         $m_{opt} := m_i$ 
18:   Ameise  $m_{opt}$  aktualisiert Pheromonmatrix  $S$ 
19:    $j := j + 1$ 

```

3.1.4. Parallele Tourkonstruktion

Bei der dritten in dieser Arbeit betrachteten Methode (Algorithmus 7) konstruieren die Ameisen ihre Route parallel. Als erstes wird hierfür eine Hilfsvariable j definiert und mit 0 initialisiert. Diese dient als Zähler für die derzeitige Stadt v_j . Weiterhin wird eine Variable für die kürzeste gefundene Distanz d_s mit ∞ initialisiert.

Dann startet die Tourkonstruktion, die mithilfe einer **For**-Schleife realisiert wird. Jede Ameise startet in einer zufälligen Stadt v_0 die gleichzeitig auch die erste Stadt r_{m_0} der Route ist. Nun ermittelt jede Ameise m zunächst die erste Stadt v_1 . Die 1. Stadt der Route r einer jeden Ameise m ist nun die Stadt v_1 . Dann wird die Tourlänge d_m mit der Distanz von der Stadt v_0 zur Stadt v_1 verlängert. Abschließend aktualisiert jede Ameise die Pheromonkonzentration auf der begangenen Kante und der Wert von j wird um 1 erhöht.

Dieses Vorgehen geschieht analog für jeden weitere Stadt v_j . Wenn jede Ameise jede Stadt

einmal besucht hat, endet der Algorithmus. Abschließend wird ein weiteres Mal durch die Menge M der Ameisen iteriert, um zunächst die Tourlänge mit der Distanz von der letzten Stadt r_{m_j} zur ersten Stadt r_{m_0} zu erweitern. Die insgesamt kürzeste Route d_s ergibt sich abschließend durch den Vergleich der Routenlängen aller Ameisen.

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:  $d_s = \infty$ 
3: while  $j < v$  do
4:   for each Ameise  $m \in M$  do
5:     if  $j = 0$  then
6:        $r_{m_0} := v_0$ 
7:       Starte in einer zufälligen Stadt  $v_0$ 
8:       Ermittle die nächste Stadt  $v_j$  und gehe dorthin
9:        $r_{m_j} := v_j$ 
10:       $d_m := d_m + D_{r_{m_{j-1}}, r_{m_j}}$ 
11:      Aktualisiere Pheromonmatrix  $S$ 
12:     $j := j + 1$ 
13: for each Ameise  $m \in M$  do
14:    $d_m := d_m + D_{r_{m_j}, r_{m_0}}$ 
15:   if Tourlänge  $d_m < d_s$  then
16:      $d_s := d_m$ 

```

} Parallel

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 vier symmetrische Probleminstanzen entnommen: **burma14**, **dantzig42**, **gr120** und **si535**. 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 heranzukommen. Sowohl die Laufzeit als auch die Effektivität hängen von mehreren Parametern ab:

1. Parameter α , der die Wichtigkeit der Pheromonkonzentration auf einer Kante bei der Wahl der nächsten zu bereisenden Stadt festlegt
2. Parameter β , 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 ρ , der den Wert der Pheromonpersistenz angibt; $1 - \rho$ steht für die Pheromonverdunstung

Zunächst wird deshalb 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.

4. Ergebnisse der Durchführung

Die Untersuchung der implementierten Algorithmen und der Einfluss der einzelnen Parameter wurde auf einem 3800 Mhz QuadCore-Rechner mit 8 Gigabyte RAM durchgeführt.

Zur besseren Analyse werden alle Daten automatisch in eine CSV-Datei geschrieben und von dort aus ausgewertet.

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 bzw. minimale Abstand und der Durchschnitt in mehreren Programmdurchläufen erfasst und prozentual ermittelt.

Für alle nachfolgenden Experimente wurde die Pheromonkonzentration auf allen Kanten mit dem Wert 0,01 initialisiert.

4.1. Einfluss der Parameter auf die implementierter Algorithmen

In diesem Abschnitt wird zunächst der Einfluss der in Abschnitt 3.2 besprochenen wesentlichen Parameter diskutiert und grafisch dargestellt. Die jeweiligen nicht untersuchten Parameter in den einzelnen Experimenten wurden empirisch auf $\alpha = 1$ und $\beta = 0,25$ bzw. $Q = 40$ und $\rho = 0,15$ festgelegt.

4.1.1. Parametereinfluss auf den iterativen Algorithmus

Der iterative Algorithmus ist die klassische Umsetzung zur Lösung einer TSP-Probleminstanz mit dem Ameisenalgorithmus. Deswegen wird dieser und der Einfluss der einzelnen Parameter zuerst betrachtet.

Bei der Untersuchung wurden zunächst die Parameter Alpha α und Beta β für das Intervall $0,05 \leq \{\alpha, \beta\} \leq 1$ mit einer Schrittweite von 0,05 iteriert. Dies wurde mit einer doppelten For-Schleife umgesetzt:

```
1 for (double alpha = 0.05; alpha <= 1.0; alpha += 0.05)
2     for (double beta = 0.05; beta <= 1.0; beta += 0.05)
3         RunAlgorithm();
```

Somit ergibt sich für jeden Wert von Alpha α die Abhängigkeit von allen Werten für Beta β . Dies gilt natürlich auch umgekehrt.

Die gleiche Vorgehensweise wurde analog für die Parameter Q (Menge des abgelegten Pheromons) und ρ (Pheromonpersistenz durchgeführt. Hierbei wurde für Q das Intervall $2 \leq Q \leq 42$ und für ρ das Intervall $0,05 \leq \rho \leq 0,95$ gewählt.

Die Ergebnisse finden sich grafisch dargestellt in Abbildung 4. Es lässt sich deutlich erkennen, dass sowohl der Parameter α , der über die Wichtigkeit der Pheromonspur Auskunft gibt, als auch der Parameter Q , der die abgelegte Pheromonmenge darstellt, bei Veränderung im gewählten Wertebereich keinen großen Einfluss auf die Tourlängen hat. Mit einem steigenden Einfluss von Parameter β werden die Routen insgesamt kürzer. Am meisten beeinflusst jedoch der Parameter ρ für die Pheromonpersistenz die gefundenen Tourlängen. Umso mehr sich dieser Parameter dem Wert 1 nähert, umso kürzer werden die Routen im Durchschnitt.

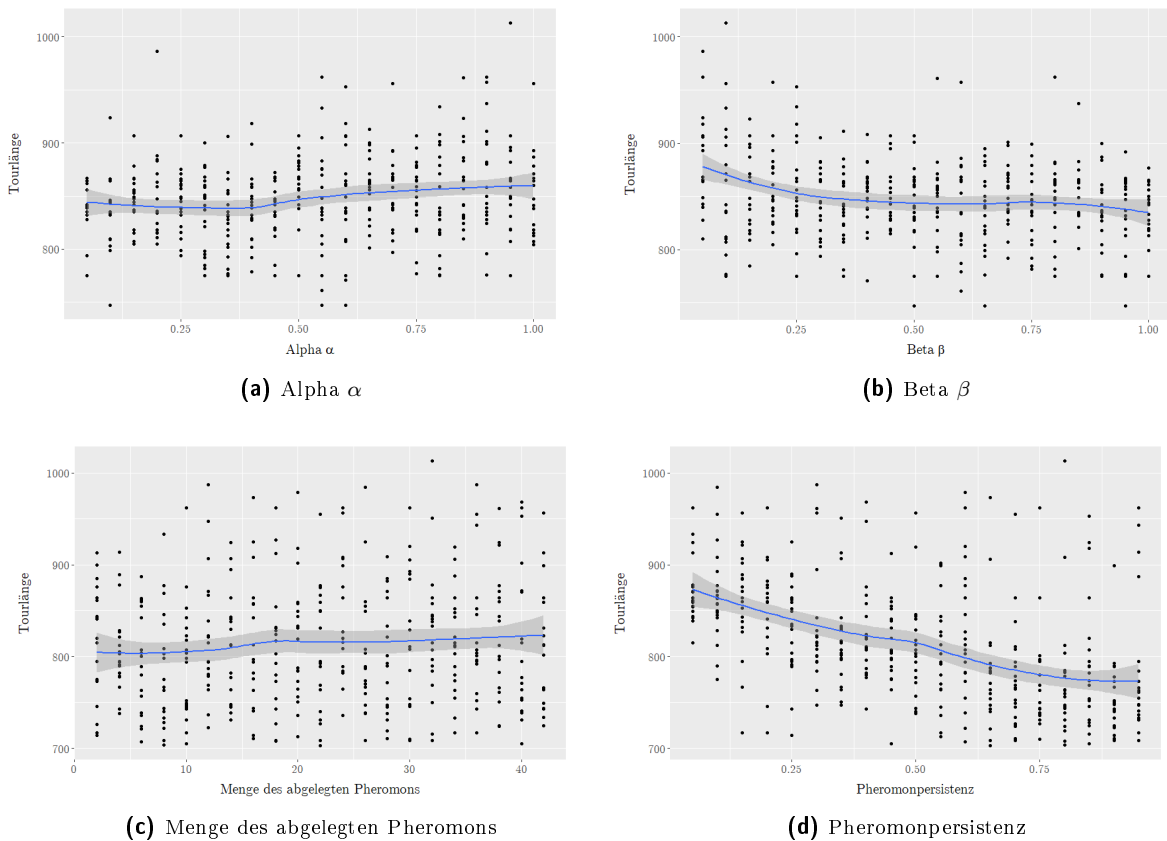


Abb. 4: Abhängigkeit der Tourlänge von den einzelnen Parametern bei der iterativen Tourkonstruktion

4.1.2. Parametereinfluss auf den iterativen Algorithmus mit eingeschränkter Pheromonaktualisierung

Wie bei dem oben untersuchten iterativen Algorithmus wurden auch hier die Parameter Alpha α und Beta β sowie Q und ρ kombiniert untersucht. Dazu wurden wie schon oben gezeigt jeweils eine doppelte **For**-Schleife verwendet. Aufgrund der höheren Komplexität und damit der höheren Laufzeit wurden die Schrittweiten zwischen den einzelnen Werten doppelt so groß gewählt. Der Wertebereich für die einzelnen Parameter ist für Alpha α und Beta β derselbe; für die Pheromonablagemenge wurde ein Wertebereich von $2 \leq Q \leq$, für die Pheromonpersistenz ein Wertebereich von $0,1 \leq \rho \leq 1$ gewählt.

Auch wenn dieser Algorithmus vom Namen her auf eine ähnliche Tourlängenbeeinflussung durch die einzelnen Parameter wie der iterative Algorithmus ohne eingeschränkte Pheromonaktualisierung schließen lässt, gestaltet sich diese wesentlich anders. Dies geht aus Abbildung 5 hervor. Der Parameter Beta β , der die Wichtigkeit der Distanz bei der Ermittlung der nächsten Stadt während der Tourkonstruktion angibt, lässt das Finden leicht kürzerer Routen bei Annäherung gegen 1 zu. Der Parameter Alpha α dagegen beeinflusst das Finden kurzer Touren bei Annäherung gegen 1 eher negativ. Jedoch ist die Streuung bei kleinen Werten für α wesentlich höher.

Über den Einfluss der Menge des abgelegten Pheromons lässt sich bei diesem Algorithmus sagen, dass diese ab circa ab einem Wert von 10 die besten Werte mit einer sehr geringen Streuung erzielt. Für einen Wert von $Q = 2$ erreicht der Algorithmus teilweise Tourlängen mit einer Abweichung von fast 300% vom Optimum. Die Pheromonpersistenz ρ steigt für höhere Werte leicht an und ist für einen Wert im Bereich von 0,05 bis 0,10 optimal im Hinblick auf

die Tourlänge sowie vor allem auf eine geringe Streuung.

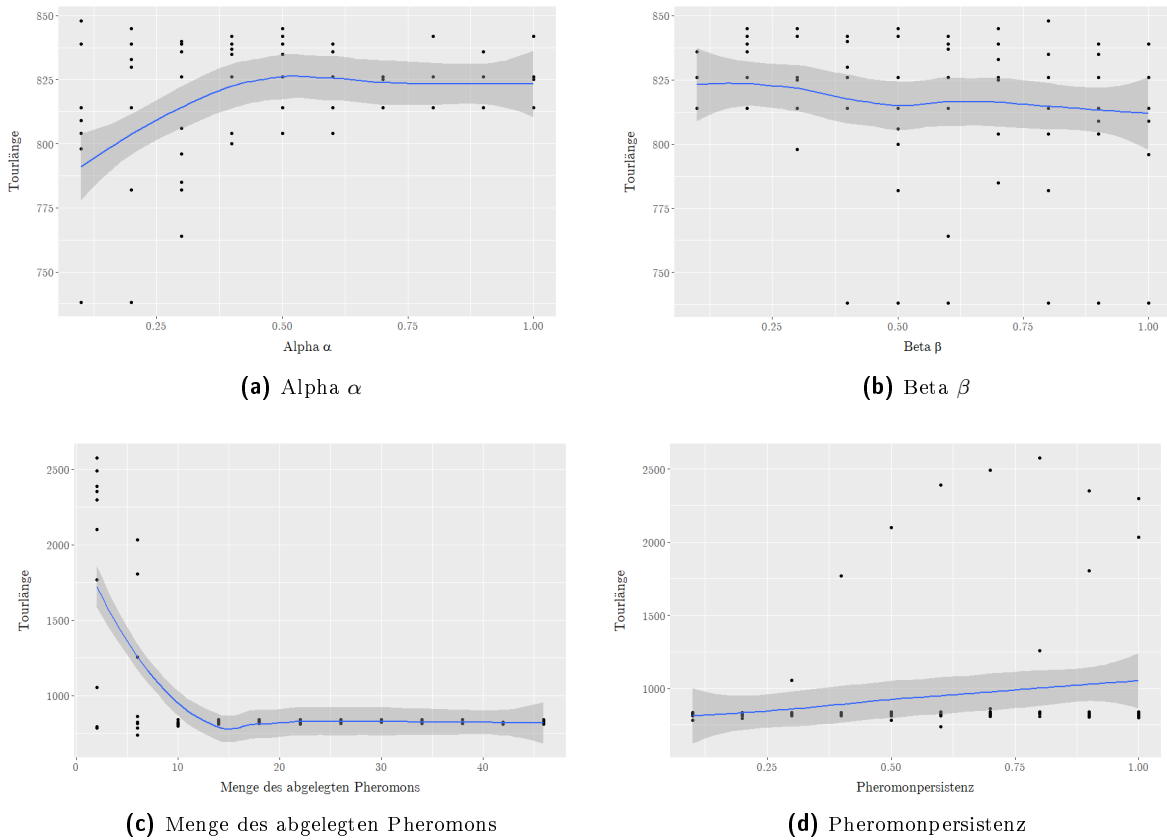


Abb. 5: Abhängigkeit der Tourlänge von den einzelnen Parametern bei der iterativen Tourkonstruktion mit eingeschränkter Pheromonaktualisierung

4.1.3. Parametereinfluss auf den parallelen Algorithmus

Beim dritten und letzten implementierten Algorithmus wird der Parametereinfluss wie bei den zwei vorhergehenden Algorithmen auch durch Kombination der Parameter untersucht. Auch hier wird Alpha α und Beta β zusammen im Intervall von $0,05 \leq \{\alpha, \beta\} \leq 1$ untersucht - jeweils mit einer Schrittweite von 0,05. Anhand von Abbildung 6 lässt sich erkennen, dass für Alpha α bessere Tourlängenwerte bei Annäherung gegen 1 erzielt werden. Für Beta β ist dieser Sachverhalt genau umgekehrt - hier werden die besten Werte bei einem geringen Einfluss der Distanz auf die Tourkonstruktion erzielt.

Für die Pheromonablage- und Pheromonpersistenzwerte wurde ein Wertebereich von $2 \leq Q \leq 42$ bzw. $0,05 \leq \rho \leq 0,95$ gewählt. Für diese Parameter lässt sich keine große Veränderung der Tourlängen feststellen; die gefundenen Tourlängen sind im Mittel konstant für alle unterschiedlichen Werte.

4.2. Auswirkung des gewählten Wahrscheinlichkeitsalgorithmus

Wie schon in Abschnitt 2.3 vorgestellt, wurden für die Erstellung dieser Arbeit zwei verschiedene Algorithmen zur Ermittlung der Wahrscheinlichkeiten der nächsten Städte bei der Tourkonstruktion implementiert. Diesen Algorithmen liegt zum einen die Formel 3 zugrunde, die auch von Thomas Stützle und Holger H. Hoos in ihrem vorgestellten *MAX-MIN Ant*

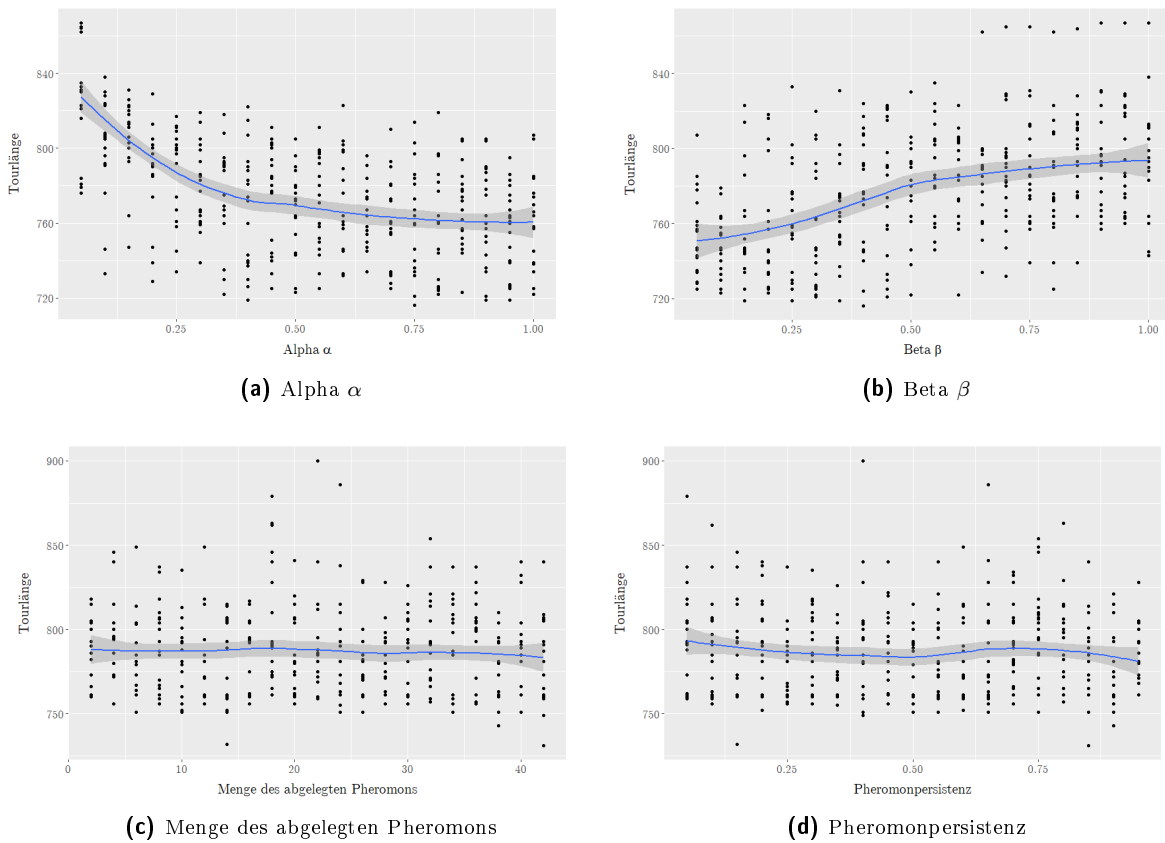


Abb. 6: Abhängigkeit der Tourlänge von den einzelnen Parametern bei der parallelen Tourkonstruktion

System verwendet wurde. Im folgenden wird der Algorithmus, der diese Formel verwendet, der Einfachheit halber *Komplexer Wahrscheinlichkeitsalgorithmus* genannt. Der andere Algorithmus verwendet eine stark vereinfachte Form dieser Formel (Gleichung 4) und wird im folgenden als *Einfacher Wahrscheinlichkeitsalgorithmus* bezeichnet.

Bei der Untersuchung wurden für die einzelnen Parameter folgende empirische Werte verwendet: $\alpha = 1$, $\beta = 0,25$, $Q = 40$ und $\rho = 0,15$. Für die Erhebung von statischen Daten wurde jeder Algorithmus zur Tourkonstruktion mit jedem Wahrscheinlichkeitsalgorithmus zehnmal getestet. Die verwendete Problemistanz war **dantzig42** mit einem Tourlängenoptimum von 699.

Die Ergebnisse sind in folgender Tabelle dargestellt:

Algorithmus	Wahrscheinlichkeitsermittlung					
	Tourlänge (Durchschnitt)	Einfach [%]	Zeit in s	Tourlänge (Durchschnitt)	Komplex [%]	Zeit in s
Iterativ	746,3	6,77	3,49	733,1	4,88	55,46
Iterativ mit eingeschränkter Pheromonaktualisierung	849,7	21,56	5,13	880,5	25,97	71,23
Parallel	818	17,02	23,33	832,6	19,11	447,45

Tab. 1: Ergebnisse für die einzelnen in Algorithmen in Abhängigkeit von dem gewählten Wahrscheinlichkeitsalgorithmus. Die Prozentzahl hinter der Tourlänge zeigt die Abweichung vom Tourlängenoptimum $d = 699$ an

Auffallend ist zu aller erst, dass der komplexe Wahrscheinlichkeitsalgorithmus im Vergleich zu dem einfachen Algorithmus sehr viel mehr Zeit zur Berechnung benötigt. Dieser Unterschied beträgt teilweise bis zu 2000 %. Im Hinblick auf die gefundenen Routenlängen lässt sich kein großer Unterschied feststellen. Bei dem Iterativen Algorithmus mit eingeschränkter Pheromonaktualisierung und der parallelen Tourkonstruktion sind die durchschnittlichen Routenlängen bei Benutzung des einfachen Wahrscheinlichkeitsalgorithmus sogar kleiner.

4.3. Ergebnisse der Algorithmen bei verschiedenen TSP-Probleminstanzen

Abschließend werden nun die drei implementierten Tourkonstruktionsalgorithmen auf vier verschiedenen TSP-Probleminstanzen angewendet, die der TSPLIB95 entnommen wurden. Gewählt wurden die Instanzen `burma14`, `dantzig42`, `gr120` und `si535`. Für alle Algorithmen wurde der eigene, einfache Wahrscheinlichkeitsalgorithmus verwendet.

In der untenstehenden Tabelle 2 sind die Ergebnisse und die jeweiligen verwendeten Werte für die verschiedenen Parameter aufgelistet.

Die Probleminstanz `si535` wurde aufgrund der absehbar hohen Laufzeit nicht mit dem *Iterativen Algorithmus mit eingeschränkter Pheromonaktualisierung* untersucht. Die Angabe der Laufzeit ist in dem Fall ein geschätzter Wert, der durch den Vergleich mit anderen Laufzeiten zustande kommt.

Für die beiden iterativen Algorithmen wurden unterschiedliche Werte der Parameter abhängig von der Anzahl der Städte eines Problems verwendet. Diese Vorgehensweise resultiert aus einem besseren Ergebnis im Hinblick auf die Effektivität. Dabei sind sowohl die einzelnen Parameterwerte als auch die Städtezahl, bei sich diese Parameter ändern, empirisch bestimmte Werte die unter anderem auf den Ergebnissen von Abschnitt 4.1 basieren.

Die Auswertung der Ergebnisse folgt im nächsten Abschnitt dieser Arbeit. Grob gesehen lässt sich aber erkennen, dass der iterative Algorithmus die besten Werte erzielt während der Iterative Algorithmus mit eingeschränkter Pheromonaktualisierung am schlechtesten abschneidet; auch was die Laufzeit des Algorithmus betrifft.

Name	Optimum	Algorithmus	gefundene Routen				Laufzeiten in Sekunden				
			Minimum	[%]	Maximum	[%]	Durchschnitt	[%]	Minimum	Maximum	Durchschnitt
burna14	3323	Iterativ	3336	0,3	3803	14,4	3450,2	3,8	0,53	0,54	0,53
		Iterativ mit eing. PA	3381	1,7	3381	1,7	3381	1,7	3,34	3,38	3,31
		Parallel	3323	0	3323	0	3323	0	0,50	0,55	0,52
dantzig42	699	Iterativ	704	0,7	755	8	721,9	3,2	4,32	5,84	5,51
		Iterativ mit eing. PA	814	16,4	836	19,6	824,2	17,9	22,74	23,59	23,01
		Parallel	731	4,6	764	9,3	749,3	7,2	3,34	3,39	3,37
gr120	6942	Iterativ	8166	17,63	8286	19,3	8221,6	18,4	32,22	44,45	42,77
		Iterativ mit eing. PA	8322	27,2	8832	27,2	8832	27,2	173,53	202,95	182,84
		Parallel	8198	18,1	8778	26,4	8540,2	21,7	24,02	26,08	24,64
si535	48450	Iterativ	50101	3,4	53532	10,4	51529,3	6,35	657,10	687,36	673,59
		Iterativ mit eing. PA	-	-	-	-	-	-	(~1 Stunde)	(~1 Stunde)	(~1 Stunde)
		Parallel	70438	45,3	74246	53,2	72137,8	48,89	466,01	570,11	503,40

Algorithmus		Alpha α			Beta β	Pheromonablagemenge Q			Pheromonpersistenz ρ	Anzahl an Ameisen		Iterationen bzw. Iterationsschwelle	
		1	0,25	40		0,1	0,9	10		1000	1000	500	500
Iterativ	$ V < = 100$	1	0,25	40		0,1	0,9	10		1000	1000	500	500
	$ V > 100$	0,1	0,9	10									
Iterativ mit eing. PA	$ V < = 100$	1	0,25	10		0,1	0,15	10		500	500	8	8
	$ V > 100$	0,1	0,9	10		0,1	0,05	10		500	500	8	8
Parallel		1	0,1	42			0,15			500		-	-

Tab. 2: Gefundene Touren von Probleminstanzen aus der TSPLIB95 in Abhängigkeit von dem gewählten Algorithmus. Der *Iterative Algorithmus mit eingeschränkter Pheromonaktualisierung* wird in dieser Tabelle mit „Iterativ mit eing. PA“ abgekürzt. Die Zahl in dem Namen der Problemistanz gibt die Anzahl der Städte in dieser an. Weiterhin stehen die Prozentangaben hinter den Tourlängen für die Abweichung vom jeweiligen Optimum. Die untere Tabelle gibt Auskunft über die verwendeten Parameter beim Versuchsdurchlauf. Um statistisch verwertbare Daten zu erhalten wurde jeder Versuch zehnmal durchgeführt.

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

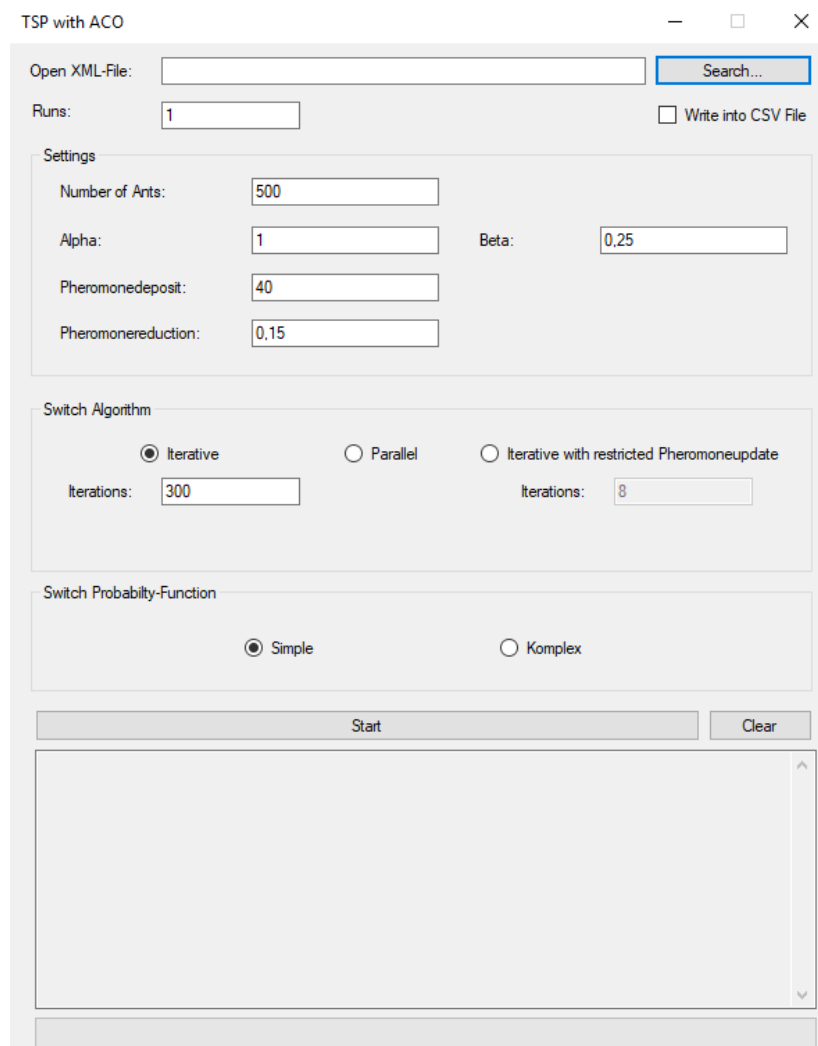


Abb. 7: Grafische Oberfläche des Programms

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 Parameter 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 **Pheromonedeposit** gibt die Menge des abgelegten Pheromons an, während **Pheromonereduction** für die

Verdunstungsmenge steht.

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. Analog gibt er für die Option **Iterative with restricted Pheromoneupdate** 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.

B. Hinweise zur Nutzung des Programms als Konsolenapplikation

Die Implementierung des Ameisenalgorithmus zur Lösung von TSP-Probleminstanzen kann auch auf Konsolenbasis unter Windows genutzt werden. Hierbei wird der Pfad und die einzelnen Optionen als Runtime-Parameter übergeben. Insgesamt müssen dabei jeweils 13 Parameter gesetzt werden.

1. Dateipfad (Datei muss dabei als XML-Datei vorliegen)
2. Anzahl der Ameisen, die eine Tour generieren
3. Gewünschter Algorithmus: „0“ entspricht dem Iterativen Algorithmus, „1“ dem parallelen Algorithmus und „2“ dem Iterativen Algorithmus mit eingeschränkter Pheromonaktualisierung
4. Wert für die Menge an abgelegtem Pheromon Q - sollte zwischen 0.05 und 80 liegen
5. Wert für die Pheromonverdunstung $1 - \rho$ - sollte zwischen 0.01 und 0.99 liegen
6. Wert für den Parameter Alpha α - sollte zwischen 0.01 und 1 liegen
7. Wert für den Parameter Beta β - sollte zwischen 0.01 und 1 liegen
8. Schalter für eine optionale allgemeine Reduzierung der Pheromonmatrix nach jeder Tourkonstruktion - für „0“ ist die Option deaktiviert, für „1“ aktiviert
9. Wert für die optionale allgemeine Reduzierung der Pheromonmatrix - sollte zwischen 0.001 und 0.1 liegen
10. Anzahl der Iterationen für den Iterativen Algorithmus, bei der Wahl von anderen Algorithmen „0“ eintragen
11. Anzahl der Iterationen beim Iterativen Algorithmus mit eingeschränkter Pheromonaktualisierung, bei der Wahl von anderen Algorithmen „0“ eintragen
12. Schalter für den Wahrscheinlichkeitsalgorithmus - bei „0“ wird der einfache Algorithmus verwendet, bei „1“ der komplexe Algorithmus
13. Schalter für die Ausgabe in eine CSV-Datei - dieser ändert den Output des Programms sodass für eine einfache weitere Verwendung in einem CSV-Datensatz - für „0“ ist die Option deaktiviert, für „1“ aktiviert

Für die Verwendung des Iterativen Algorithmus mit 1000 Ameisen und einer Iterationsschwelle von 500 zur Lösung der **burma14**-Probleminstanz könnte der Prozessaufruf so aussehen:

```
1 TSPACO.exe burma14.xml 1000 1 40.0 0.14 1.0 0.25 0 0.001 500 0 0 0
```

Literatur

- [1] Christian Dietrich and Erich Steiner. Das Leben unserer Ameisen - ein Überblick. *Bio-logiezentrum 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 Insect 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.