
# Ameisenalgorithmus und Algorithmus von Dijkstra im Vergleich auf OpenStreetMap

Master-Thesis zur Erlangung des Grades

## Master of Science, Informatik

eingereicht am 15.03.2016

*Eingereicht von* **Timo Jürgens**

*Betreuer:* **Prof. Dr. Sebastian Iwanowski**
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Telefon: (041 03) 80 48-63

*Zweitgutachter:* **Prof. Dr. Ulrich Hoffmann**
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Telefon: (041 03) 80 48-41

# Inhalt

# 1 Einleitung

Dieses Kapitel gibt einen Überblick über das Thema und die Inhalte dieser Arbeit. Außerdem begründet es die Themenwahl.

## 1.1 Motivation und Zielsetzung

Verkehrsnavigation ist ein immer wichtiger werdendes Themenfeld. Sind Navigationssysteme schon lange Zeit auf dem Vormarsch, hat das Aufkommen des Smartphones Navigationssoftware noch allgemein verfügbarer gemacht. Mit dem steigenden Angebot an entsprechenden Produkten, steigen auch die Ansprüche an diese. Es reicht nicht mehr aus, einfach nur den Weg von Start zum Ziel anzuzeigen. Heutzutage ist es wichtig, flexibel auf sich ändernde Umstände reagieren zu können. Es ist für das Erreichen des Ziels in möglichst kurzer Zeit notwendig, dass sich beim Bilden von Staus die dem Nutzer angezeigte Route dynamisch ändert.

Immer wieder werden neue Algorithmen gesucht, die das Navigationsproblem möglichst effizient lösen und gleichzeitig die Ansprüche an die Dynamik erfüllen. Dabei stellt sich die Frage, in wie fern spezielle Verfahren überhaupt nötig sind. Viele bereits vorhandene allgemeinere Ansätze sind sehr vielversprechend. Sie erlauben es, die Erfahrungen, die mit diesen Algorithmen in zahlreichen Anwendungen bereits gesammelt wurden, zu nutzen, um daraus auf das Verkehrsproblem angepasste Lösungen zu entwickeln.

Diese Arbeit beschäftigt sich mit dieser Möglichkeit, in dem sie untersucht, in wie fern der gut erforschte und solide Algorithmus von Dijkstra sich im direkten Vergleich mit einem dynamischen und auf die Problemstellung angepassten Ameisensystem schlägt. Es soll unter anderem festgestellt werden, welche Herangehensweise unter welchen Umständen die bessere ist. Die Ergebnisse von theoretischen Vergleichen sollen anschließend anhand von ausführlichen Tests validiert werden. Als Grundlage für diese Tests soll das auf einem Ameisenalgorithmus basierende Verkehrssimulationsprogramm AntScout angepasst und so erweitert werden, dass eine spezifische und problemadäquate Testumgebung entsteht. AntScout muss lernen, Verkehrsprobleme nicht nur mit dem Ameisenverfahren, sondern auch mit dem Dijkstra-Algorithmus lösen zu können. Nur dann können konkrete in der Realität auftretende Verkehrssituationen, die moderne Navigationssysteme teilweise vor besondere Probleme stellen, damit untersuchen werden. Durch diese Untersuchungen kann anschließend festgestellt werden, welche Randbedingungen und Sonderfälle bestimmte Algorithmentypen besonders unterstützen oder behindern.

## 1.2 Übersicht und Gliederung

In Kapitel 2 werden die theoretischen Grundlagen präsentiert, die für das Verständnis der weiteren Kapitel nötig sind. Es werden die für diese Arbeit essentiellen Verfahren Dijkstra-Algorithmus und Ameisensystem erklärt und die Besonderheiten von AntNet erläutert. Außerdem wird das Programm AntScout vorgestellt, auf dem diese Arbeit aufbaut.

Kapitel 3 beschäftigt sich detailliert mit der Konzeption und Implementierung von AntScout. Dabei wird unterschieden, was bereits vor dieser Arbeit vorliegt und was im Rahmen dieser Arbeit neu entsteht. Die einzelnen Klassen werden vorgestellt und es wird erläutert, wie diese miteinander interagieren. Außerdem werden die Dateien erklärt, die beim Arbeiten mit der Software benötigt oder erzeugt werden.

Kapitel 4 vergleicht den Dijkstra-Algorithmus und das AntNet-Verfahren sowohl theoretisch als auch experimentell anhand von verschiedenen Fällen miteinander. Die dabei erzielten Ergebnisse werden präsentiert und ausgewertet. Außerdem werden die wichtigsten Details der Hard- und Software, die die Testumgebung darstellen, kurz dargestellt.

Das fünfte und letzte Kapitel fasst die Erkenntnisse dieser Arbeit zusammen und bildet ein Fazit. Außerdem wird ein Ausblick auf die Möglichkeiten und potentiellen zukünftige Folgeprojekte gegeben, die sich aus dieser Arbeit ergeben können.

# 2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen behandelt, die für die korrekte Einordnung der restlichen Arbeit notwendig sind. Außerdem liefert es einen Überblick über das Programm AntScout, auf dem diese Arbeit aufbaut.

## 2.1 Shortest Path Problem

Ein beliebtes Problem unter Mathematikern ist es, in einem gerichteten kantengewichteten Graphen den kürzesten Weg von einem bestimmten Knoten zu einem anderen zu finden. Ein gerichteter Graph besteht aus einer Menge V von Knoten und einer Menge geordneter Kontenpaare $E \subseteq V\ x\ V$ von Kanten. Die Kanten $(v, w) \in E$ eines gerichteten Graphen sind gerichtete Kanten. Bei einem kantengewichteten Graphen ist jeder Kante eine reelle Zahl als Gewicht zugeordnet. Zum Thema Graphentheorie gibt es umfangreiche Literatur, z.B. von Reinhard Diestel[1].

Probleme, die sich auf die oben beschriebene Grundsituation abstrahieren lassen, werden *Shortest Path Problem* genannt. Der statische Fall, in dem sich an dem Graphen nach Beginn der Berechnung nichts mehr ändert, ist in zahlreichen Veröffentlichungen behandelt worden, erstmals 1966 von H.C. Joksch[2]. Es existieren daher zahlreiche Algorithmen, um die unterschiedlichen Ausprägungen dieser Aufgabenstellung zu lösen. Ein Beispiel dafür ist der Algorithmus von Dijkstra, der in 2.2 ausführlich erklärt wird.

Komplizierter ist der dynamische Fall, in dem sich die Gewichte der Kanten während der Berechnung verändern. Ein vor der Veränderung als beste Lösung berechneter Weg kann nach der Berechnung stark vom Optimum abweichen. Einen konkreten Fall des dynamischen Shortest Path Problem stellt die Navigation im Straßenverkehr dar. Ein Verkehrsteilnehmer möchte stets den schnellsten Weg von seiner aktuellen Position zu seinem Ziel angezeigt bekommen. Die Lage im Straßennetz ist dabei ständiger Veränderung unterworfen. Es können jederzeit Verkehrsänderungen oder andere Verkehrsstörungen entstehen und sich wieder auflösen. Eine zu Beginn der Fahrt als optimal berechnete Route kann sich also noch während dieser negativ verändern, so dass eine Abweichung von der ursprünglich berechneten Lösung die Reisezeit verkürzt. Ein Algorithmus, der dieses Problem lösen soll, muss sich also schnell genug an die sich verändernde Lage anpassen können. Beispiele für

[1] (Diestel, 2010).
[2] (Joksch, 1966)

Algorithmen zum Lösen von dynamischen Problemen sind Ameisenverfahren, die in 2.3 ausführlich erklärt werden.

## 2.2 Algorithmus von Dijkstra

Der Dijkstra-Algorithmus ist ein Verfahren zum Lösen des statischen Shortest Path Problem und gehört zur Klasse der Greedy-Algorithmen. Entwickelt wurde er vom Niederländer Edsger W. Dijkstra, der ihn 1959 erstmals beschreibt[3]. Der Algorithmus kann eingesetzt werden, um den kürzesten Weg von einem Startpunkt entweder zu allen möglichen Punkten (*single-source shortest path)* oder nur zu einem Zielpunkt (*single-pair shortest path)* zu berechnen. Der Ablauf für die Variante *single-pair shortest path* wird im Folgenden skizziert.

### Initialisierung

Zu Beginn liegen alle Knoten in der Menge *unexplored.* Die Menge *explored* ist leer. Alle Knoten besitzen die Eigenschaften *Distanz* und *Vorgänger.* Bei allen Knoten beträgt die *Distanz* ∞*.* Beim Startknoten beträgt die *Distanz 0.*

### Vorgehensweise

Schritt 1. Es wird der Knoten K aus *unexplored* gewählt, dessen *Distanz* minimal ist. K wird von *unexplored* in *explored* verschoben. Handelt es sich dabei um den Zielknoten, endet der Algorithmus.

Schritt 2. Für jeden Nachbarknoten Kn von K, der in *unexplored* liegt, wird die Summe $S_{Kn}$ aus dem Gewicht der Kante (K,Kn) und der *Distanz* von K berechnet. Ist die *Distanz* von Kn größer als $S_{Kn}$, erhält sie den Wert von $S_{kn}$ und K wird *Vorgänger* von Kn.

Schritt 3. Beginne wieder bei Schritt 1.

### Weitere Betrachtungen

Im Rahmen dieser Arbeit wird der Algorithmus so implementiert, dass er effizient in der Verkehrsnavigation eingesetzt werden kann. Auf die Implementierung wird in Kapitel 3 näher eingegangen. Der Algorithmus funktioniert nur einwandfrei, wenn der Graph zusammenhängend ist und keine negativen Kantengewichte besitzt. Beides ist in einem Verkehrsnetz sichergestellt. Der allgemeine Dijkstra-Algorithmus besitzt im schlechtesten Fall eine Laufzeiteffizienz von $O(n^2 \log n)$, wobei n die Anzahl an Knoten im Graphen darstellt. Da in Verkehrsnetzen die Anzahl an Kanten pro Knoten stark begrenzt ist, kann der Knotengrad als konstant behandelt werden, was die Laufzeiteffizienz auf $O(n \log n)$ verbessert.

---

[3] (Dijkstra, 1959)

## 2.3 Ameisenalgorithmen

Ameisenalgorithmen sind eine Gruppe von metaheuritischen Verfahren zum Lösen von kombinatorischen Optimierungsproblemen. Heuristische Verfahren haben gemein, dass sie Optimierungsprobleme nur näherungsweise lösen. Der genaue Grad der Exaktheit der gefundenen Lösung hängt von diversen Parametern ab.

Ameisensysteme basieren auf dem Vorbild von natürlichen Ameisen und ihrem Verhalten bei der Nahrungssuche. Die Ameisen geben auf ihrem Weg vom Bau zum Ziel und wieder zurück als Pheromone bezeichnete Duftstoffe ab, die andere Ameisen wahrnehmen können. Jeder mögliche Weg zur Nahrungsquelle wird zu Beginn von durchschnittlich gleich vielen Ameisen verwendet. Die Ameisen auf dem kürzesten Weg erreichen das Ziel früher als die anderen und kehren schneller wieder zum Bau zurück. Ihr Weg wird dadurch auch früher als die anderen mit einer neuen Pheromondosis versehen. Nachfolgende Ameisen wählen dann bevorzugt diesen Weg und verstärken die Pheromone darauf noch einmal zusätzlich. Mit der Zeit entsteht so eine Ameisenstraße.

Dieses Verhalten wird mit künstlichen Ameisen, die auf einem Graphen operieren, imitiert. Pheromone werden als Werte an den Knoten simuliert. Ameisen werden an einem Knoten erzeugt und bewegen sich entlang der Kanten von Knoten zu Knoten bis zu einem Zielknoten. Sie werden dabei von den Pheromonwerten an den Knoten in der jeweiligen Wahl des Folgeknoten beeinflusst. Der zurückgelegte Weg wird sich gemerkt. Am Ziel angekommen, werden die Pheromonwerte nachträglich an den passierten Knoten in Abhängigkeit von der Lösungsgüte aktualisiert. Die nachträgliche Aktualisierung ist ein Vorteil der künstlichen gegenüber den natürlichen Ameisen.

Sowohl bei natürlichen als auch bei künstlichen Ameisen findet im Laufe der Zeit eine Verdunstung von Pheromonen statt, so dass aktuellere Pheromone ein stärkeres Gewicht besitzen. Auf diese Weise können sich die Ameisen an dynamische Änderungen innerhalb des Graphen anpassen.

Ameisenalgorithmen sind für ein breites Feld von Optimierungsproblemen einsetzbar. Sie müssen aber jeweils an die konkrete Aufgabenstellung angepasst werden. Daher gibt es unterschiedliche Implementierungen, die jeweils verschiedene Ziele anstreben. Aufgrund der heuristischen Natur der Verfahren sind sie besonders für NP-vollständige Probleme geeignet. Die Probleme dieser Komplexitätsklasse lassen sich nach Expertenmeinung vermutlich nicht effizient lösen. Der Beweis, ob ein NP-vollständiges Problem tatsächlich nicht effizient gelöst werden kann, wurde bisher aber noch nicht erbracht. Eine sehr bekannte

Implementierung eines Ameisenverfahrens ist AS (Ant System) von Marco Dorigo, die zur Lösung des NP-vollständigen Traveling Salesman Problem entwickelt wurde[4].

Der aktuelle Zustand eines Ameisensystems kann jederzeit abgefragt werden und liefert auch zu jedem Zeitpunkt eine Lösung. Jede Änderung von außen kann sofort eine Reaktion des Ameisensystems auf die geänderten Umstände bewirken. Aus diesem Grund gibt es diverse Implementierungen, die flexibel auf die jeweils zu erwartenden dynamischen Änderungen der konkreten Problemstellung während der Laufzeit reagieren können. Diese Eigenschaft machen sie interessant für das dynamische Shortest Path Problem.

Eine Weiterentwicklung von AS stellt das Ameisensystem AntNet dar. Dieses wurde von Marco Dorigo und Thomas Stützle im Jahr 2004 vorgestellt und diente ursprünglich dem Lösen von Routingproblemen in Kommunikationsnetzwerken.[5] Im Rahmen dieser Arbeit wird eine auf Verkehrsprobleme angepasste Version dieses Verfahrens als Grundlage verwendet. Diese Version wird detailliert in der Masterthesis von Alexander Bertram vorgestellt[6]. Der konkrete Ablauf wird im Folgenden skizziert.

Schritt 1. In gewissen Zeitabständen werden an jedem Knoten so viele Ameisen erzeugt, wie es andere Knoten gibt und jeder Ameise einer dieser Knoten als Zielknoten zugeordnet.

Schritt 2. Jede Ameise entscheidet sich für den nächsten zu besuchenden Knoten. Die Entscheidung basiert auf dem lokalen Modell des aktuellen Knotens und der Pheromonmatrix. Das lokale Modell bildet die Verkehrssituation aus der Sicht dieses Knotens ab. Die Pheromonmatrix gibt durch die gespeicherten Pheromonwerte in Abhängigkeit vom Zielknoten an, welche abgehende Kante wie attraktiv ist. Dadurch ergibt sich für jede Kante ein Wahrscheinlichkeitswert, auf dessen Basis eine Kante zufällig ausgewählt wird. Die Ameise wandert entlang der ausgewählten Kante zum nächsten Knoten. Dieser Schritt wird so lange wiederholt, bis die Ameise den Zielknoten erreicht (weiter bei Schritt 3) oder die maximale Lebenszeit abgelaufen ist (weiter bei Schritt 4).

Schritt 3. Am Ziel angekommen, erfolgt die Aktualisierung der Datenstrukturen der besuchten Knoten. Die Aktualisierung basiert auf den Informationen aus dem Gedächtnis der Ameise und der bisherigen Pheromonwerte.

[4] (Dorigo & Gambardella, 1997)
[5] (Dorigo & Stützle, 2004)
[6] (Bertram, 2012) S.10 - 14

Schritt 4. Nach der Aktualisierung aller besuchten Knoten oder nachdem die maximale Lebenszeit der Ameise abgelaufen ist, wird diese aus dem System entfernt. Die Begrenzung der Lebenszeit vermeidet die Bildung von Zyklen, die ein Problem bei echten Ameisen darstellt.

## 2.4 AntScout

AntScout ist ein auf Java basierendes Verkehrssimulationsprogramm, das mit Hilfe der in 2.3 beschriebenen angepassten Version des Ameisenalgorithmus *AntNet* den schnellsten Weg zwischen zwei Punkten auf Straßenkarten von OpenStreetMap bestimmt und anzeigt. Es wurde im Rahmen der Masterthesis von Alexander Bertram entwickelt.[7] OpenStreetMap ist ein freies Projekt[8], das Geodaten sammelt und diese in einer Datenbank jedem frei zur Verfügung stellt. Basierend auf diesen Daten können Straßenkarten erzeugt und angezeigt werden, was sich AntScout zu Nutze macht, wie in Abbildung 1 zu sehen ist. Die Abbildung zeigt die webbasierte Oberfläche von AntScout. Dem Benutzer wird das Straßennetz eines in einer Konfigurationsdatei festgelegten Kartenausschnitts in Form eines Graphen angezeigt. Die Konfigurationsdatei wird in Abschnitt 3.9 noch behandelt. Jeder Knoten kann als Start-(A) oder Endpunkt(B) der Wegsuche gewählt werden. In der rechten oberen Ecke werden Länge und benötigte Fahrzeit für den aktuell am besten bewerteten Weg angezeigt.

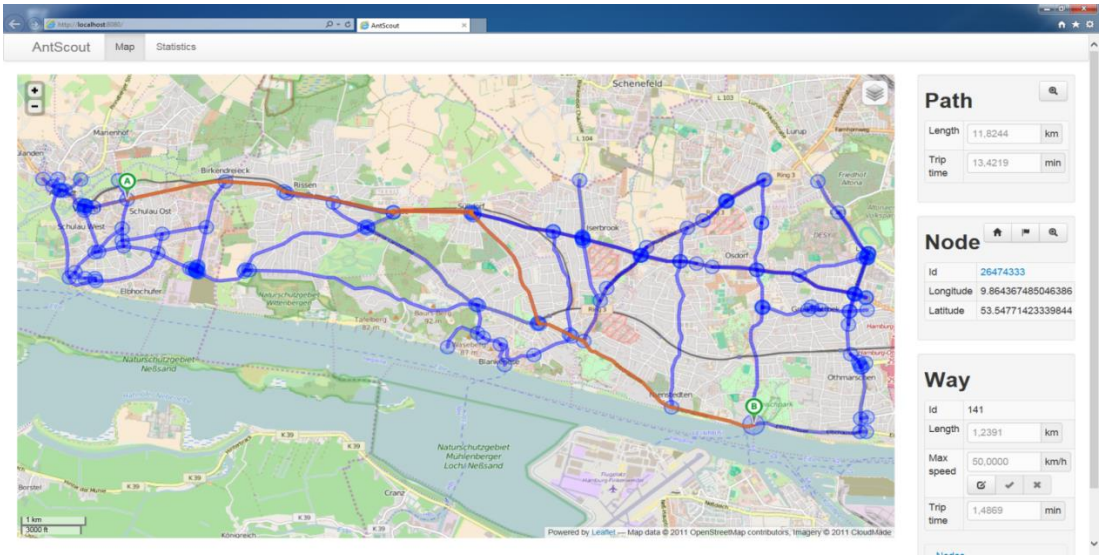


**Abbildung 1: AntScout**

[7] (Bertram, 2012)
[8] http://www.openstreetmap.de/

AntScout berechnet den schnellsten Weg und reagiert dabei dynamisch auf Veränderungen der Durchschnittsfahrgeschwindigkeit an einzelnen Kanten. Während der Ausführung wird die präsentierte Lösung an die geänderten Umstände angepasst. Solche Änderungen stellen z.B. sich bildende Staus dar und werden manuell vom Benutzer über die Oberfläche eingegeben. Die Ameisen werden fortlaufend neu erstellt, aufgrund der aktuellen Pheromonverteilung durch den Graphen geschickt und nach Ablauf ihrer Lebensspanne wieder entfernt.

Da es sich bei Ameisenalgorithmen um heuristische Verfahren handelt, ist nicht beweisbar, wie gut die aktuell von AntScout gefundene Lösung im Vergleich mit dem tatsächlichen Optimum ist. Aufgrund der konkreten Implementierung der Ameisen in AntScout ist es wahrscheinlich, dass die präsentierte Lösung zwischen mehreren ähnlich guten Wegen schwankt, ohne dass sich eine auf Dauer als beste Lösung durchsetzt. Eine aus diesem Verhalten resultierende weiter gestreute Pheromonverteilung vereinfacht die Anpassung, wenn es zu dynamischen Veränderungen kommt. AntScout kann so schneller auf diese reagieren.

Die Implementierung von AntScout arbeitet mit dem Aktorensystem Akka[9]. Aktoren sind nebenläufige Einheiten, die ausschließlich über *Message passing* kommunizieren. Unter *Message passing* wird der Austausch von Nachrichten mit Hilfe von Mailboxen verstanden. Nebenläufige Programme leiden häufig unter verschiedenen Problemen, wie z.B. aufwändiges Prozessmanagement und die Synchronisierung von gemeinsam genutzten Ressourcen. Diese werden mit *Message passing* vermieden. Da es sich bei dem innerhalb von AntScout umgesetzten AntNet wie bei allen Ameisensystemen um einen parallelen Algorithmus handelt, profitiert dieser sehr von der nebenläufigen Implementierung, die Akka ermöglicht. Die Verwaltung des Nachrichtensystems erzeugt allerdings gewisse Mengen an Overhead, die in kleinen Graphen verhältnismäßig stark negativ auffallen können.

AntScout wurde in der sowohl funktionalen als auch objektorientierten Programmiersprache Scala entwickelt. Der vom Compiler aus dem Scala-Code erzeugte Byte-Code wird auf der Java VM ausgeführt, wodurch bei der Arbeit mit Scala auch vollwertige Java-Bibliotheken verwendet werden können. Der Hauptgrund für die getroffene Programmiersprachenwahl ist die von Scala angebotene Unterstützung für Akka. In der zum Entwicklungszeitpunkt aktuellen siebten Version von Java ist diese Unterstützung hingegen noch nicht enthalten, sondern wurde erst mit der später erschienenen achten Version eingeführt. Bei

[9] http://akka.io/

längerer Laufzeit von AntScout wächst die produzierte Datenmenge stetig an, so dass bei längerem Betrieb der Arbeitsspeicher sehr belastet wird. Dies kann eine Folge der Implementierung unter Scala sein, da Scala bei bestimmten Operationen große Mengen an temporären Datenstrukturen anlegt.

# 3 Softwarekonzeption und –implementierung

Im Rahmen dieser Arbeit wird wie bereits in 1.1 erläutert, eine Testumgebung zum Vergleichen eines Ameisenverfahrens mit dem Algorithmus von Dijkstra geschaffen. Mit dem in AntScout realisierten AntNet-Verfahren liegt bereits eine vollwertige Implementierung eines Ameisensystems vor, die im Rahmen dieser Arbeit angepasst und erweitert wird. Alle dabei entstehenden Änderungen und Ergänzungen an AntScout werden, wie AntScout selbst, in Scala entwickelt oder beruhen auf importierten Java-Bibliotheken. In diesem Kapitel wird der Aufbau von AntScout gezeigt und die verschiedenen Komponenten erklärt.

## 3.1 Aufbau

Die im Rahmen dieser Arbeit neu entstehenden Klassen fügen sich in die bereits vorhandene Klassenstruktur von AntScout ein. Abbildung 2 zeigt den relevanten Ausschnitt, an dem die hierarchische Ordnung zu erkennen ist. Der Dijkstra-Algorithmus wird durch die Klassen *Dijkstra-Service* und *Graph* repräsentiert. *JamGen* koordiniert Funktionalitäten zur automatisierten Simulation von Verkehrsbehinderungen. Die anderen dargestellten Klassen waren bereits Bestandteil von AntScout und werden nur angepasst, um die Kompatibilität mit den neuen Funktionen zu unterstützen. Die genauen Funktionsweisen der einzelnen Klassen werden in den folgenden Abschnitten erklärt. Die im Rahmen dieser Arbeit von Grund auf neu entwickelten Klassen sind in Abbildung 2 mit dem Hinweis **neu** und in den folgenden Abschnitten mit dem Hinweis **(neu)** versehen.

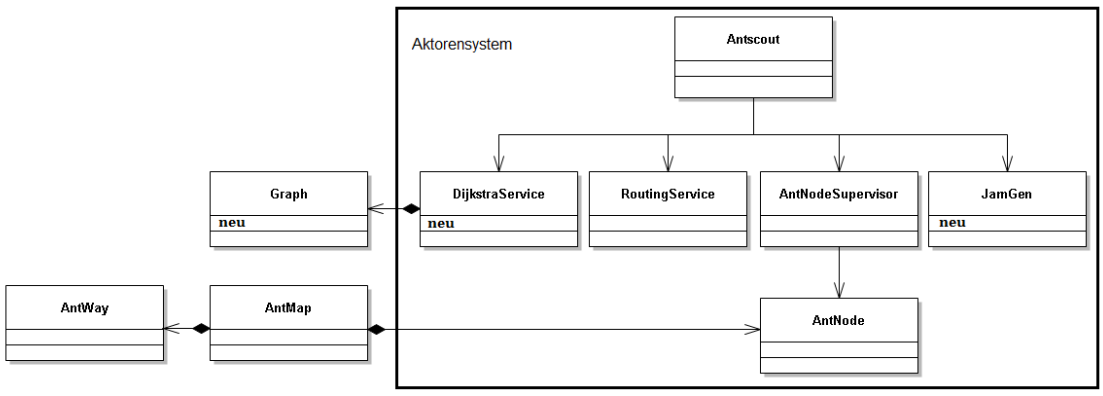


**Abbildung 2: Klassenstruktur**

## 3.2 AntScout

Die Klasse *AntScout* fungiert als zentraler Ankerpunkt des Programms. Die darin bereits vorhandene Datenstruktur wird weiterverwendet und erweitert. Beim Start des Programms übernimmt AntScout in Abhängigkeit von den Einstellungen in einer Konfigurationsdatei (siehe 3.9) die Initialisierung von entweder *DijkstraService* (falls der Dijkstra-Algorithmus verwendet werden soll) oder *RoutingService* (falls der AntNet-Algorithmus verwendet werden soll). Die Klasse verwaltet viele der als Aktoren vorliegenden Objekte, wie in Abbildung 2 angedeutet wird.

## 3.3 AntNodeSupervisor

Die Klasse *AntNodeSupervisor* ist ein Aktor, der die *AntNode*-Aktoren erzeugt und überwacht. Er initialisiert diese bei Programmstart mit Hilfe des Floyd-Warshall-Algorithmus, der die kürzesten Pfade zwischen allen Knotenpaaren bestimmt. Die Laufzeit dieses Algorithmus beträgt $O(n^3)$, wobei n die Anzahl der Knoten ist. Basierend auf dessen Ergebnis, werden alle unerreichbaren Knoten entfernt und bei allen anderen Knoten die anfängliche Pheromonverteilung vorgenommen.

## 3.4 Serviceklassen

Um eine stringente Testumgebung zu gewährleisten, muss sich AntScout unabhängig davon, auf welche Weise die beste Route bestimmt wird, immer vergleichbar verhalten. Darum gibt es mit *RoutingService* und *DijkstraService* für beide Routingverfahren jeweils einen Aktor, der Anfragen annimmt und beantwortet. Die Schnittstelle ist in beiden Fällen identisch, so dass es für den Aufrufer unerheblich ist, mit welcher Serviceklasse er kommuniziert.

### 3.4.1 RoutingService

Wird das Ameisensystem für das Routing verwendet, beantwortet die Klasse *RoutingService* alle Anfragen nach dem schnellsten Weg. Realisiert wird die Klasse als Aktor. Erhält der *RoutingService* eine Anfrage nach dem besten Weg, wird mit Hilfe der auf den Pheromonwerten aller Knoten basierenden Routingtabelle die als beste Route markierte Folge von *AntWays* an den Aufrufer zurück geschickt.

Erhält der *RoutingService* eine Anfrage nach einer Kantenänderung, wird der relevante Teil der Routingtabelle neu berechnet. Falls die sich ändernde Kante ein Teil des Pfades ist, wird die aktuell beste Route aktualisiert[10].

### 3.4.2 DijkstraService (neu)

Analog dazu nimmt die ebenfalls als Aktor implementierte Klasse *DijkstraService* Anfragen entgegen, wenn der Dijkstra-Algorithmus verwendet werden soll. Die von AntScout verwendete Repräsentation von Kanten und Knoten als *AntWays* und *AntNodes* ist aufgrund ihrer verhältnismäßig komplexen Struktur nicht mit einer effizienten Verwendung durch den Dijkstra-Algorithmus kompatibel. Aus diesem Grund und um die Funktionalität besser zu kapseln, wird die vorhandene Struktur in ein Objekt vom Typ *Graph (siehe 3.5.2)* überführt. Innerhalb dieses Objektes findet auch das eigentliche Routing statt. Die als Ergebnis der Optimierung gelieferte Kantenfolge wird in eine Folge von *AntWays* überführt und an den Aufrufer zurück gesendet. *DijkstraService* und *RoutingService* bedienen auf diese Weise dieselbe Schnittstelle und für den Aufrufer spielt es keine Rolle, welchen Service er tatsächlich aufgerufen hat. Sowohl *RoutingService* als auch *DijsktraService* arbeiten aufgrund ihrer Natur als Aktoren asynchron zu allen anderen Prozessen.

## 3.5 Graphenklassen

Das AntNet-Verfahren und der Dijkstra-Algorithmus haben unterschiedliche Anforderungen an die Datenstrukturen, um Graphenoperationen möglichst effizient auszuführen. In AntScout existieren daher zwei dementsprechende Modelle parallel nebeneinander.

### 3.5.1 AntMap, AntWay, AntNode

Die aus *OpenStreetMap* ausgelesenen und für AntScout aufbereiteten Straßenkarten werden als Graph durch die Klasse *AntMap* repräsentiert. Kanten werden durch den Typ *AntWay,* Knoten durch den Typ *AntNode* dargestellt. Die Instanzen von *AntNodes* werden von der übergeordneten Klasse *AntNodeSupervisor* erstellt und überwacht. In den *AntNodes* werden die Ameisen erzeugt und entlang der *AntWays* von einem *AntNode* zum anderen geschickt. Dabei aktualisieren sie die Pheromonwerte, die ebenfalls dort gespeichert werden.

[10] Details in (Bertram, 2012) S24. - 27

### 3.5.2 Graph (neu)

Die Klasse *Graph* repräsentiert eine alternative Implementierung eines Graphen, die sich von der oben beschriebenen Struktur unterscheidet. Der Graph ist definiert als zweidimensionale Tabelle und lässt sich als Funktion

$$f: Knoten \in N \rightarrow Knoten \in N \rightarrow Kantengewicht \in R$$

repräsentieren. Wie das innerhalb von AntScout implementiert ist, wird in Abbildung 3 gezeigt. Die Variable *matrix* stellt f dar. Es existieren keine separaten Klassen für Kanten und Knoten. Kanten werden indirekt über zwei Knoten mittels f definiert und Knoten als einfache Ganzzahl dargestellt. Kantengewichte sind als Fließkommazahlen dargestellt und lassen sich mit der Methode *setDistance* setzen und ändern. Diese Darstellung ist wesentlich schlanker als die in AntScout vorherrschende und für die Nutzung durch den Dijkstra-Algorithmus, der selbst ebenfalls in dieser Klasse implementiert ist, optimiert. Dies erlaubt eine effizientere Ausführung. Ist die vorhandene Struktur aus *AntWays* und *AntNodes* erst einmal durch den *DijkstraService* in diese Form transformiert, ist die komplette notwendige Funktionalität und Datenstruktur zum Anwenden des Verfahrens, mit Ausnahme der Ausgabe, in einer Instanz von *Graph* gekapselt. Sie werden dadurch vor Veränderungen an Kanten in AntScout während der Rechenzeit geschützt.

```
type Node = Int
type Distance = Double

var nextNode: Node = 0
var matrix = Map[Node, Map[Node, Distance]]()

def mkNode(): Node =
  {
    var node: Node = nextNode
    matrix += (node -> Map())
    nextNode += 1
    node
  }

def getNodes() = matrix.keys

def setDistance(from: Node, to: Node, distance: Distance) =
  {
    matrix.get(from) match {
      case Some(m) => matrix += (from -> (m + (to -> distance)))
      case None => // if this happens, you used an undefined node
    }
  }
```

**Abbildung 3: Ausschnitt aus der Klasse Graph**

## 3.6 JamGen (neu)

AntScout unterstützt nur eine manuelle Eingabe von Verkehrsänderungen. Über die Oberfläche kann die Durchschnittsfahrgeschwindigkeit einzelner Straßenabschnitte geändert und einzelne Kantengewichte dadurch angepasst werden. In der Realität ändert sich die Verkehrslage allerdings ständig und schneller, als dies durch manuelle Änderungen realistisch dargestellt werden kann. *JamGen* ist eine als Aktor implementierte Klasse, die der Verwaltung einer automatisierten Stauerzeugung dient. Während der Laufzeit von AntScout werden mit parametrisierbarer Frequenz über einen Scheduler gesteuert Verkehrsänderungen erzeugt und an die Oberfläche geschickt. Dies wirkt sich in der jeweiligen Änderung des Kantengewichts, also der Durchschnittsfahrgeschwindigkeit, einer Kante aus. Für die Oberfläche macht es keinen Unterschied, ob eine Änderung manuell oder von JamGen angestoßen wird. Die Verkehrsänderung kann eine Anpassung der besten Route erforderlich machen. Die jeweils betroffene Kante kann von *JamGen* entweder per Zufall oder anhand einer Staudatei ausgewählt werden. Welche Kanten im Fall von zufälliger Erzeugung berücksichtigt werden sollen, lässt sich ebenfalls beeinflussen. So kann z.B. eingestellt werden, dass nur Kanten, die Teil des aktuell besten Weges sind, ausgewählt werden können. Alle Einstellungen von *JamGen* werden über eine Konfigurationsdatei vorgenommen, die in Abschnitt 3.9 ausführlich erklärt wird. Auf diese Weise lassen sich sehr spezifische Testfälle erstellen. Erzeugte Verkehrsänderungen lassen sich aufzeichnen, um später wiederum *JamGen* als Staudatei zu dienen. AntScout kann dadurch beliebig oft mit der exakt selben Verkehrsänderungsreihenfolge ausgeführt werden, was die Vergleichbarkeit der Ausführungen garantiert. Verkehrsänderungen können positiv oder negativ ausfallen.

*JamGen* verwaltet zusätzlich noch die Ausgabe des aktuell besten Weges. Über einen Scheduler gesteuert, wird während der Laufzeit von AntScout mit parametrisierbarer Frequenz die benötigte Fahrzeit auf dem aktuell besten Weg in eine Datei geschrieben. Die Gesamtheit der Fahrzeiten dient als Vergleichsgrundlage zwischen den unterschiedlich parametrisierten AntScout-Ausführungen. Details dazu werden in 3.8 und 3.9.4 behandelt.

## 3.7 Kommunikationskonzept

Die Kommunikation innerhalb der Klassenstruktur erfolgt fast komplett über Nachrichtenaustausch, da die meisten Klassen als Aktoren realisiert sind. Wie die dabei entstehende Kommunikationsstruktur genau aussieht, wird in diesem Abschnitt am Beispiel des *DijsktraService* erklärt und in Abbildung 4 dargestellt.

*JamGen* erzeugt eine zufällige Verkehrsänderung und schickt diese in Form einer Kantenänderungsanfrage an das *FrontEnd.* Dort wird die geforderte Änderung an der Oberfläche vorgenommen und eine neue Nachricht mit der getätigten Kantenänderung erzeugt. Die Nachricht wird an die aktuelle Serviceklasse gesendet, was in diesem Beispiel der *DijkstraService* ist. Dieser führt die Änderung an dem entsprechenden *AntWay* durch und sendet eine Pfadanfrage an den *Graphen.* Dort wird der Dijkstra-Algorithmus auf den dafür zuvor transformierten Graphen angewendet und der berechnete Pfad anschließend an die Serviceklasse zurückgesendet. Diese schreibt den Pfad als neue beste Route in eine Sessionvariable und sendet ihn an das *FrontEnd.* Dort wird der Pfad auf der Oberfläche gezeichnet. Bei der Verwendung des *RoutingService* verhalten sich die Vorgänge analog dazu.

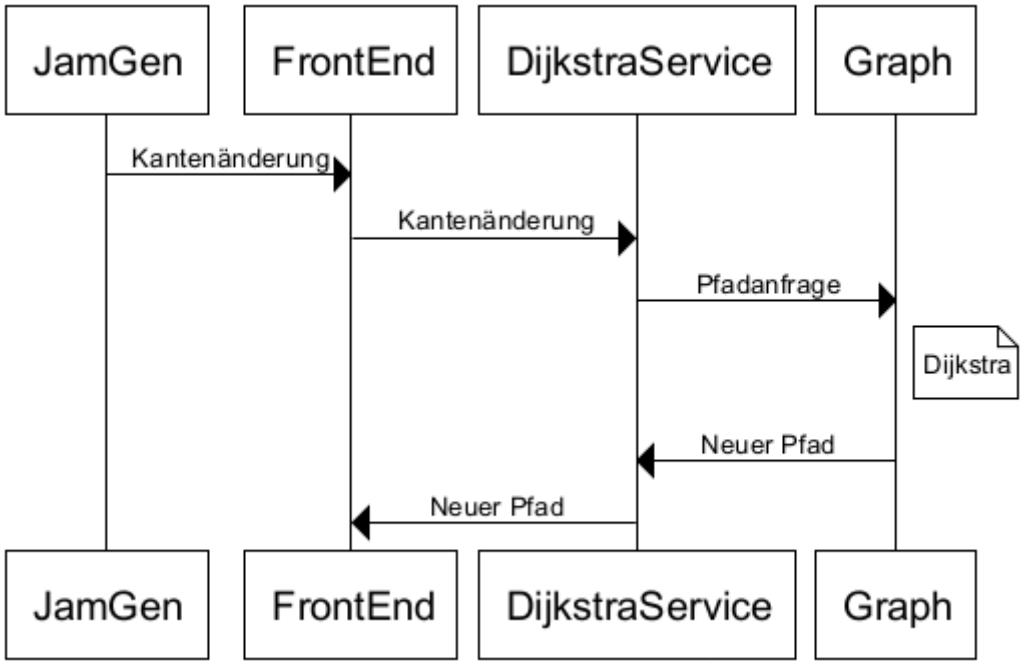


**Abbildung 4: Kommunikation bei Kantenänderung anhand des DijkstraRoutings**

## 3.8 Erzeugte Dateien (neu)

Um die Ergebnisse von AntScout-Ausführungen miteinander vergleichen zu können, müssen diese gespeichert werden. Diese Aufgabe übernehmen die Stau- und Ausgabedateien. Beide Dateitypen werden im Folgenden erklärt.

### 3.8.1 Staudatei

Um das Verhalten von Dijkstra- und AntNet-Verfahren vergleichbar zu machen, müssen beide Algorithmen mit denselben Verkehrsänderungen arbeiten können. Daher lässt sich eine Datei angeben, in der alle erzeugten Verkehrsänderungen gespeichert werden. Ein Ausschnitt aus einer solchen Staudatei ist in Abbildung 5 dargestellt. Er zeigt die Verkehrsänderungen in der Reihenfolge, in der sie erzeugt wurden. Jede Zeile enthält die relevanten Informationen zu einem Stau. Vor dem Komma steht die ID der Kante, auf den sich der Stau bezieht. Hinter dem Komma steht die Durchschnittsgeschwindigkeit, auf die der Stau den Wert der Kante ändert, in der Einheit Kilometer pro Stunde.

Soll AntScout zu einem späteren Zeitpunkt mit den gleichen Verkehrsänderungen noch einmal ausgeführt werden, können die Verkehrsänderungen in der gespeicherten Reihenfolge nacheinander ausgelesen werden. Sie werden dann als Verkehrsänderungen angewendet, so als wären sie gerade erst erzeugt worden. Sowohl das Erzeugen als auch das Auslesen von Verkehrsänderungen geschieht in der Klasse *JamGen* (siehe 3.6), die das Resultat unabhängig von dem Auslöser als Veränderung einer Kante weitergibt. Für den Rest von AntScout spielt es keine Rolle, ob eine Veränderung zufällig oder mittels einer Staudatei erfolgt.

```
101,40.00000000000001
191,40.000032
56,83.33333333333333
118,83.33333333333333
127,30.0000000000000004
83,250.0
201,250.0
139,83.33333333333333
198,250.0
8,20.000000000000004
207,42.00000000000001
152,34.99992
166,35.00000000000001
115,30.0000000000000004
19,349.99920000000003
85,15.000012000000003
79,10.000008
205,40.00000000000001
201,75.00000000000001
119,20.0000000000000004
187,166.66666666666666
42,24.999999999999996
187,16.666666666666664
170,166.6667759896245
189,12.0000000000000002
203,30.000024000000007
```

**Abbildung 5: Ausschnitt aus einer Staudatei**

### 3.8.2 Ausgabedatei

Um die Ergebnisse verschiedener AntScout-Ausführungen vergleichen zu können, sind Ausgabedateien erforderlich, aus denen die Ergebnisse auslesbar sind. Auf diese Weise wird Persistenz der Ergebnisse erreicht. In Abbildung 6 ist ein Ausschnitt aus einer solchen Ausgabedatei zu sehen. Ganz oben sind die wichtigsten Parameterwerte, die bei der Entstehung der Ausgabe eine Rolle spielen, zusammengefasst. Die Bedeutung der einzelnen Einträge wird in 3.9 detailliert erläutert. Darunter stehen die in regelmäßigen Abständen erzeugten Ausgaben. Sie geben die jeweiligen Fahrzeiten zur gerade besten Route in Minuten an. Anhand dieser Werte lassen sich Vergleiche bezüglich der Lösungsgüte anstellen. Die Ausgabe ist darauf optimiert, besonders leicht von einem automatisierten Testtool ausgewertet werden zu können.

```
positive Staus erlaubt=false
ungelenkter Zufall=false
Stauerzeugungsfrequnz=500
Ausgabefrequenz=250
Grundfaktor=0.1
Maximalfaktor=9
13.421930530505389
13.421930530505389
13.425497127423462
13.425497127423462
13.472548635787366
13.4782808031252
13.54789228922571
13.54789228922571
13.55762770652556
13.55762770652556
14.038239341096505
14.038239341096505
14.410777759669422
14.410777759669422
14.414344904031354
14.414344904031354
15.311836380298125
15.311836380298125
15.31505396802314
15.31505396802314
15.385722465164049
15.385722465164049
15.413249733371039
15.413249733371039
16.184650112865153
16.184650112865153
16.500966449773582
16.500966449773582
16.76797291342411
16.76797291342411
17.04905748473815
```

**Abbildung 6: Ausschnitt aus einer Ausgabedatei**

## 3.9 reference.conf

Die Konfigurationsdatei *reference.conf* dient der Verwaltung diverser Parameter zur Steuerung von AntScout. So lässt sich zum Beispiel die zu verwendende OpenStreetMap-Karte auswählen oder die Standardgeschwindigkeit auf bestimmten Straßenformen einstellen. Es lassen sich auch unterschiedliche Parameter für die Feinjustierung der Ameisenerzeugung und –steuerung ändern. Diese werden aber für alle im Rahmen dieser Arbeit durchzuführenden Ausführungen von AntScout auf den Werten belassen, die sich laut Dorigo und Stützle[11] als besonders geeignet herausgestellt haben.

Abbildung 7 zeigt einen Ausschnitt aus der Konfigurationsdatei. In den ersten Zeilen ist der boolesche Parameter *dijkstra* zu sehen. Mit diesem kann ausgewählt werden, ob das AntNet-Verfahren oder der Dijkstra-Algorithmus beim Ausführen von AntScout verwendet werden soll. Über den booleschen Parameter *gen* kann die Funktionalität von *JamGen* abgeschaltet werden, so dass Verkehrsänderungen nur noch manuell erzeugt werden können. Der Ganzzahl-Parameter *maxOutput* kontrolliert die Laufzeit von Stau- und Ausgabeerzeugung. Alle anderen Parameter, die im Rahmen dieser Arbeit relevant sind, werden im Folgenden detailliert erläutert. Jeder Parameter wird unter dem Namen eingeführt, unter dem er in Kapitel 4 im Rahmen der Versuchsbeschreibung verwendet wird.

```
# Soll Dijkstra statt Ameisen verwendet werden?
dijkstra = false

# Millisekunden, nach denen jeweils die Güte des aktuellen Pfades ausgegeben wird
frequencyOutput = 50

# Maximale Anzahl an Ausgaben
maxOutput = 1200

# Parameter für die Stauerzeugung
jam-gen {

  # Soll die automatische Stauerzeugung verwendet werden?
  gen = true

  # Sollen Staus vollkommen zufällig sein? Wenn nicht, dann werden Staus nur entlang des aktuell kürzesten Weges erzeugt
  trueRand = true

  # Aus welcher Datei sollen Staus geladen werden? Ist das Feld empty, werden sie nicht geladen, sondern zufällig erzeugt.
  loadJam = stau63.txt

  # In welcher Datei sollen Staus gespeichert werden? Ist das Feld empty, werden sie nicht gespeichert.
  saveJam = empty
```

**Abbildung 7: Ausschnitt aus reference.conf**

[11] (Dorigo & Stützle, Ant Colony Optimization, 2004)

### 3.9.1 Positive Verkehrsänderungen erlauben

Der boolesche Parameter *positive* steuert, welche Arten von Verkehrsänderungen *JamGen* erzeugen darf. Beträgt der Wert *false*, so sind nur Verkehrsänderungen erlaubt, die das Gewicht der beeinflussten Kante negativ verändern. Beträgt der Wert *true*, darf die Kantenbeeinflussung negativ oder positiv ausfallen. Die Unterscheidung in negativ und positiv wird im Weiteren als Ausprägung bezeichnet. Sind beide möglichen Ausprägungen erlaubt, entscheidet bei jeder erzeugten Verkehrsänderungen der Zufall, welche von beiden Ausprägungen in diesem Fall verwendet wird. Die eigentliche Staustärke wird von Grundfaktor und Maximalfaktor bestimmt (siehe 3.9.5).

### 3.9.2 Ungelenkter Zufall

Der boolesche Parameter *trueRand* beeinflusst, auf welchen Kanten Verkehrsänderungen erzeugt werden. Beträgt der Wert *false,* so werden von *JamGen* bei der zufälligen Auswahl nur Kanten berücksichtigt, die Teil der gerade besten Route sind. Beträgt der Wert *true*, wird die jeweils zu beeinflussende Kante vollkommen zufällig aus allen vorhandenen gewählt.

### 3.9.3 Stauerzeugungsfequenz

Über den Ganzzahl-Parameter *frequency* lässt sich einstellen, wie häufig Verkehrsänderungen auftreten sollen. Ab dem Zeitpunkt, wo durch Auswahl von Start- und Zielpunkt die erste Route berechnet werden kann, werden Verkehrsänderungen in regelmäßigen Abständen erzeugt. Bei *frequency* wird die Länge dieses Abstands in Millisekunden angegeben.

### 3.9.4 Ausgabefrequenz

Der Ganzzahl-Parameter *outputFrequency* beeinflusst die Ausgabe von AntScout. Ab dem Zeitpunkt, wo durch Auswahl von Start- und Zielpunkt die erste Route berechnet werden kann, wird die Fahrzeit über die gerade beste Route in regelmäßigen Abständen in eine Datei geschrieben (siehe 3.8.2). Bei output*Frequency* wird die Länge dieses Abstands in Millisekunden angegeben.

### 3.9.5 Grundfaktor und Maximalfakor

Der Fließkomma-Parameter *factor* und der Ganzzahl-Parameter *maxChange* bestimmen zusammen die Staustärke. Diese wird zufällig erzeugt und mit dem aktuellen Kantengewicht der beeinflussten Kante multipliziert. So ergibt sich die durch den Stau veränderte neue Durchschnittsfahrgeschwindigkeit des Abschnitts.

Es wird eine Zufallszahl zwischen 1 und dem Wert von *maxChange* erzeugt. Diese wird mit dem Wert von *factor* multipliziert. Der Parameter *factor* bestimmt somit die minimale Staustärke. Das Produkt aus den beiden Parametern stellt die maximale Staustärke dar und sollte den Wert 1 nicht übersteigen. Abhängig von der Ausprägung (siehe 3.9.1) wird eventuell noch der Kehrwert der bestimmten Staustärke gebildet und statt der eigentlichen Staustärke verwendet. Das Ergebnis wird dann auf die ebenfalls zufällig bestimmte Kante (siehe 3.9.2) angewendet.

### 3.9.6 Dateien

Die in 3.8 beschriebenen Dateien werden ebenfalls über Parameter in *reference.conf* verwaltet. Für die Ausführungen mit AntNet-Verfahren und Dijkstra-Algorithmus lassen sich zwei getrennte Ausgabedateien festlegen. In Abbildung 7 sind unten die Parameter zu sehen, die darüber entscheiden, ob eine Staudatei dem Auslesen von bereits vorhandenen oder dem Speichern von neu erzeugten Verkehrsänderungen dient. Werden Verkehrsänderungen aus der Staudatei ausgelesen, spielen alle hier aufgeführten Parameter mit Ausnahme der Ausgabefrequenz (siehe 3.9.4) keine Rolle. Wird als Parameter eine Datei angegeben, die noch nicht vorhanden ist, wird diese neu angelegt. Ist sie hingegen bereits vorhanden, wird der neue Inhalt an die Datei angehängt. Ein Überschreiben der Datei findet nicht statt.

# 4 Vergleich der Algorithmen

In diesem Kapitel sollen der Dijkstra-Algorithmus und das AntNet-Verfahren direkt miteinander verglichen werden. Dabei werden verschiedene Fälle unterschieden. Jeder Fall wird erklärt und seine Relevanz für ein reelles Verkehssystem erörtert. Außerdem werden die theoretischen Vor- und Nachteile der beiden Algorithmen, bezogen auf den speziellen Fall, untersucht und daraus eine Vermutung bezüglich des Verhaltens beider Verfahren gestellt. Diese wird anschließend experimentell überprüft und das Ergebnis anhand von Testreihen und Graphen präsentiert. Alle Tests werden auf dem Verkehrsgraphen durchgeführt, der in Abbildung 8 zu sehen ist und Wedel sowie die Hamburger Elbvororte repräsentiert. Dies ist der größte Graph, auf dem AntNet in (Bertram, 2012) erfolgreich getestet werden konnte. Die Gesamtheit der die Testparameter betreffenden Begriffe, die in diesem Kapitel verwendet werden, sind in 3.9 erklärt.

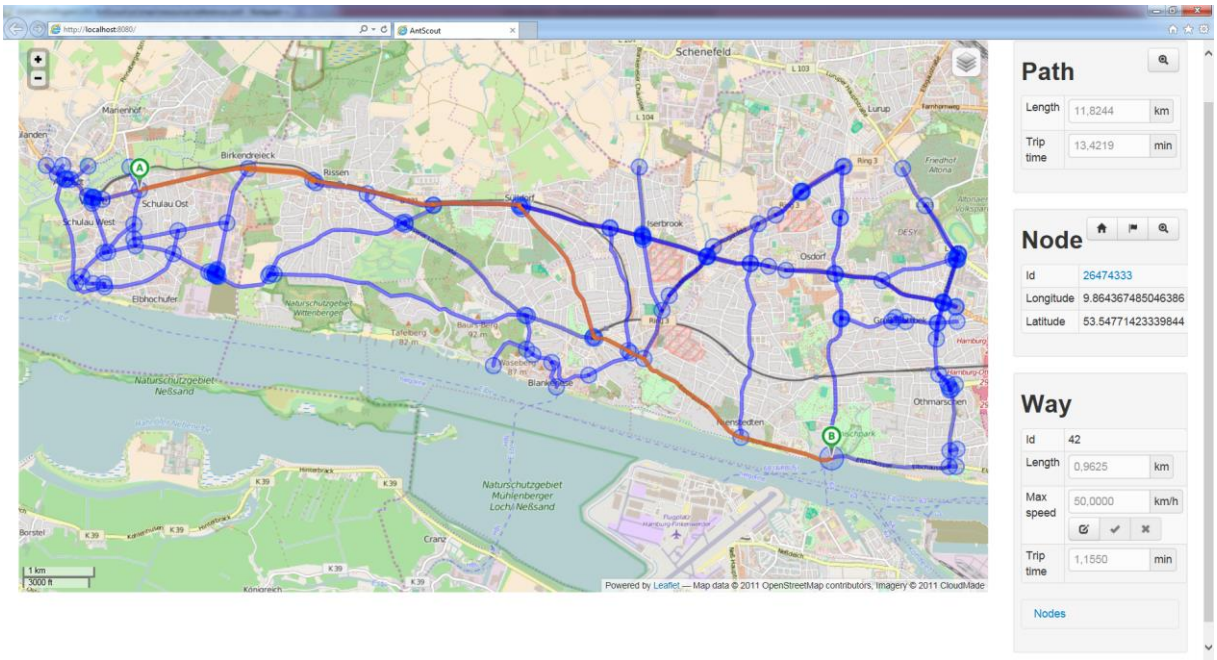


**Abbildung 8: Verwendeter Verkehrsgraph mit Start- und Zielpunkt**

## 4.1 Testumgebung

Nachfolgend ist die Spezifikation des Computers aufgeführt, auf dem die Untersuchungen durchgeführt werden.

### 4.1.1 Hardware

| | |
|---|---|
| Prozessor: | Intel Core i7-3770 CPU, 4 × 3.40 GHz |
| Arbeitsspeicher: | 8 GB |

### 4.1.2 Software

| | |
|---|---|
| Betriebssystem: | Microsoft Windows 7, 64 Bit |
| Java Development Kit (JDK): | Oracle JDK 1.8.0, 64 Bit |

## 4.2 Statisches Routing

Der einfachste Fall ist das statische Routing. Die Situation bleibt während der kompletten Laufzeit konstant. Ein dynamisches Verhalten wird von den beiden Verfahren nicht gefordert.

### 4.2.1 Theoretische Überlegung

Der Dijkstra-Algorithmus wurde für statische Routingprobleme, wie in diesem Fall eines vorliegt, entwickelt und liefert für diese Probleme die optimale Lösung. Dieser Fall ist daher die Idealbedingung für das Verfahren.

Der AntNet-Algorithmus als heuristisches Verfahren kann keine optimale Lösung garantieren. Es ist also durchschnittlich schlechter als der Dijkstra-Algorithmus und maximal genauso gut wie dieser. Der Grad der Differenz bei der Lösungsqualität hängt von der Ähnlichkeit der Fahrzeit des schnellsten Weges zu der von anderen Wegen ab. Im statischen Fall reicht eine einzige Anwendung des Dijkstra-Algorithmus, um das Optimum zu finden, daher beträgt die Laufzeit O(n log n). Da das AntNet-Verfahren nebenläufig implementiert ist, lässt sich die theoretische Laufzeit nicht auf trivialem Weg vorhersagen.

### 4.2.2 Versuchsbeschreibung

Beide Algorithmen werden mit den in Abbildung 8 dargestellten Start- und Zielpunkten ausgeführt. Der Staugenerator wird nicht verwendet.

*Ausgabefrequenz*: 250

### 4.2.3 Auswertung

Abbildung 8 zeigt die optimale Route für den statischen Fall. Wie in Abbildung 9 zu sehen, liefert der Dijkstra-Algorithmus über die komplette Laufzeit konstant das optimale Ergebnis, während das AntNet-Verfahren stetig wechselnde Ergebnisse liefert. Diese sind zu jedem Zeitpunkt schlechter als die von Dijkstra. Durchschnittlich sind sie über 10% schlechter. Da der Dijkstra-Algorithmus im Gegensatz zum AntNet-Verfahren immer das Optimum berechnet, ist für den statischen Fall immer das Dijkstra-Verfahren zu bevorzugen. Die weiter oben aufgestellte Vermutung wurde bestätigt. Die schnellen Wechsel in den Ergebnissen von AntNet liegen in der Natur der Ameisenimplementierung begründet. Dies wird ausführlich in (Bertram, 2012) erläutert. AntNet kann nur in einem dynamischen System den Vergleich gewinnen. Es werden daher entsprechende Fälle untersucht.

```
Ausgabefrequenz=250
1 13.421930530505389
2 13.421930530505389
3 13.421930530505389
4 13.421930530505389
5 13.421930530505389
6 13.421930530505389
7 13.421930530505389
8 13.421930530505389
9 13.421930530505389
10 13.421930530505389
11 13.421930530505389
12 13.421930530505389
13 13.421930530505389
14 13.421930530505389
15 13.421930530505389
16 13.421930530505389
17 13.421930530505389
18 13.421930530505389
19 13.421930530505389
20 13.421930530505389
21 13.421930530505389
22 13.421930530505389
23 13.421930530505389
24 13.421930530505389
25 13.421930530505389
26 13.421930530505389
27 13.421930530505389
28 13.421930530505389
29 13.421930530505389
30 13.421930530505389
31 13.421930530505389
32 13.421930530505389
33 13.421930530505389
34 13.421930530505389
35 13.421930530505389
36 13.421930530505389
37 13.421930530505389
38 13.421930530505389
39 13.421930530505389
40 13.421930530505389
```

```
Ausgabefrequenz=250
1 15.688207713706257
2 15.688207713706257
3 14.234521736725984
4 14.927395527651099
5 15.197404717984883
6 14.927395527651099
7 15.138463679836809
8 15.138463679836809
9 13.482284053087216
10 13.482284053087216
11 14.986336565799173
12 14.927395527651099
13 14.175580698577912
14 13.482284053087216
15 13.482284053087216
16 14.901256471679549
17 14.978672495493312
18 14.234521736725984
19 14.842315433531475
20 14.986336565799173
21 15.063752589612935
22 14.901256471679549
23 14.987850392961354
24 14.901256471679549
25 15.274820741798646
26 14.978672495493312
27 15.006325378627045
28 15.004811551464861
29 14.927395527651099
30 14.927395527651099
31 20.258294765964056
32 14.986336565799173
33 14.986336565799173
34 15.138463679836809
35 15.138463679836809
36 15.004811551464861
37 15.215879703650574
38 15.004811551464861
39 13.934158096524177
40 13.934158096524177
```

**Abbildung 9: Fahrzeit in Minuten, links für das Dijkstra-, rechts für das AntNet-Verfahren**

## 4.3 Allgemeines dynamisches Routing

Beim allgemeinen dynamischen Fall treten Verkehrsänderungen vollkommen zufällig auf. Dies gilt für Stärke, Ort und Ausprägung. Start- und Zielpunkt sind konstant. Die Ausführzeit variiert.

### 4.3.1 Theoretische Überlegung

Der Dijkstra-Algorithmus als statisches Optimierungsverfahren kann nicht direkt dynamisch auf Veränderungen reagieren. Durch die Steuerung des Verfahrens über den Aktor *DijkstraService* berechnet es asynchron zum restlichen System seine Lösung. Diese basiert auf der Situation, die zu Beginn der Berechnung vorliegt und stellt die darin optimale Route dar. Tritt während der Berechnung eine neue Verkehrsänderung auf, unterscheidet sich die Situation vor und nach der Berechnung. Es ist möglich, dass die gefundene Lösung für die neue Situation dann nicht die optimale Route darstellt. Erst bei der nächsten Ausführung liefert der Dijkstra-Algorithmus die garantiert optimale Route, sofern während der Berechnung nicht erneut eine Verkehrsänderung entsteht.

Der AntNet-Algorithmus als heuristisches Verfahren liefert grundsätzlich eine Lösung, von der nicht bekannt ist, wie nah sie am Optimum liegt. Beide Verfahren liefern also bei komplett zufälligen Verkehrsänderungen durchschnittlich nicht die optimale Route. Da die auftretenden Verkehrsänderungen sich auch auf Kanten verteilen, die für das Erreichen des Ziels keine Rolle spielen und bei relevanten Kanten auch sehr schwach ausfallen können, ist die optimale Route nach einem neuen Stau häufig die gleiche wie davor. In einem solchen Fall spielt es keine Rolle, dass die Lösung des Dijkstra-Algorithmus erst verspätet zur Verfügung steht, da die zuletzt bestimmte Lösung weiterhin die optimale bleibt. Die Vermutung liegt nahe, dass auch in diesem Fall der Dijkstra-Algorithmus den Vergleich durchschnittlich gewinnt.

### 4.3.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei bei jeder Ausführung neue Verkehrsänderungen zufällig erzeugt werden. Die Erzeugung der Verkehrsänderungen folgt den unten zusammengefassten Parametern. Als Endpunkte dienen die in Abbildung 8 dargestellten. Danach werden mehrere Ausführungen mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden wie zuvor beim Dijkstra-Algorithmus. Jede Ausführung läuft jeweils 5 Sekunden länger als die Ausführung davor, um zu überprüfen, ob die Ausführzeit das Ergebnis beeinflusst.

| | |
|---|---|
| *Positive Verkehrsänderungen erlauben:* | Ja |
| *Ungelenkter Zufall:* | Ja |
| *Stauerzeugungsfequenz:* | *500* |
| *Ausgabefrequenz:* | 250 |
| *Grundfaktor:* | 0.1 |
| *Maximalfaktor:* | 9 |

### 4.3.3 Auswertung

Abbildung 10 zeigt die Veränderung der gerade am besten bewerteten Route im Verlauf der Programmausführung an drei Beispielen. Die Kurven, die eine identische Nummer besitzen, sind das Ergebnis von Ausführungen, die mit denselben Verkehrsänderungen getestet wurden. Wie zu sehen ist, bleiben die von dem Dijkstra-Algorithmus erzeugten Lösungen nach jeder Änderung über längere Zeit konstant. In zwei Fällen bleiben sie dies sogar über die komplette Zeit. Durchschnittlich alle zwei Ausgaben wird ein neuer Stau erzeugt, der eine neue Dijkstra-Berechnung auslöst, um die optimale Lösung für die durch den Stau veränderte Situation zu berechnen. Konstante Ergebnisse über mehr als 2 Ausgaben zeigen, dass die optimale Lösung bereits vorliegt und durch die zuletzt erzeugte Verkehrsänderung nicht verändert wird. Das Dijkstra-Verfahren liefert also über weite Strecken die garantiert optimale Lösung.

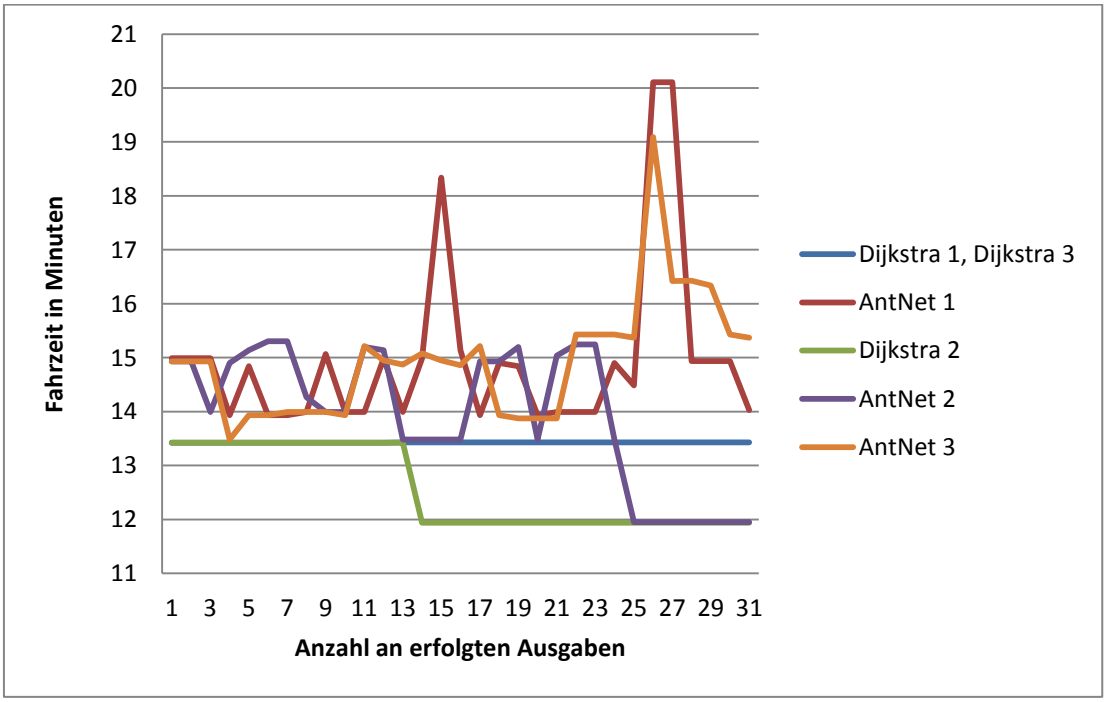


**Abbildung 10: Verschiedene Programmausführungen**

Die Lösungen des Ameisenverfahrens hingegen schwanken bei fast jeder Ausgabe erheblich. Dabei sind die Lösungen bei allen Ausführungen fast durchgehend schlechter als bei dem Dijkstra-Algorithmus. Zwischenzeitlich ist die Lösungsqualität bei Test 2 und 3 zwar mit der des Dijkstra-Verfahrens identisch, aber den Rest der Zeit unterliegen sie ebenfalls. Wie viel schlechter AntNet im Vergleich mit Dijkstra durchschnittlich abschneidet, ist im Folgenden zusammengefasst:

Test 1: Ameisen sind durchschnittlich 11.51 % schlechter als Dijkstra.

Test 2: Ameisen sind durchschnittlich 4.75 % schlechter als Dijkstra.

Test 3: Ameisen sind durchschnittlich 15.75 % schlechter als Dijkstra.

Zusätzliche Ausführungen, die zu diesem Fall durchgeführt wurden, bestätigen die oben angedeutete Tendenz. Auch für den allgemeinen dynamischen Fall gewinnt also der Dijkstra-Algorithmus den direkten Vergleich und bestätigt die oben formulierte Vermutung. Dies ist eine Konsequenz daraus, dass in diesem Fall ein einmal gefundenes Optimum durchschnittlich auch verhältnismäßig lange das tatsächliche Optimum bleibt. Es spielt daher keine große Rolle, dass der Dijkstra-Algorithmus immer erst verzögert nach einem neuen Stau eine neue Lösung präsentieren kann. Es müssen also Fälle untersucht werden, in denen sich die optimale Route häufiger ändert.

Jede der oben dargestellten Ausführungen hatte eine jeweils 5 Sekunden längere Ausführzeit als die jeweilige Ausführung davor. Da die Differenz der Lösungsqualität der beiden Verfahren in Test 2 erheblich sinkt, aber in Test 3 wieder erheblich steigt, scheint kein Bezug der Ausführzeit auf die Lösungsqualität zu bestehen. Es wird versucht, diesen Umstand in weiteren Tests zu verifizieren.

## 4.4 Dynamisches Routing mit erhöhtem Stauaufkommen

Bei diesem dynamischen Fall treten Verkehrsänderungen vollkommen zufällig auf. Dies gilt für Ort und Ausprägung. Die Staustärke sowie Start- und Zielpunkt sind konstant. Die Staudichte ist zum vorherigen Fall stark erhöht. Die Ausführzeit variiert.

### 4.4.1 Theoretische Überlegung

Das erhöhte Stauaufkommen soll den Fakt kompensieren, dass nicht jeder auftretende Stau einen Einfluss auf die optimale Route besitzt. Die festgelegte und erhöhte Staustärke soll die Quote der für die Routenbestimmung relevanten Verkehrsänderungen verbessern. Aufgrund dessen kommt es durchschnittlich zu häufigeren Änderungen der gerade optimalen Route. Es tritt häufiger die Situation auf, dass die optimale Route vor und nach der Berechnung durch den Dijkstra-Algorithmus eine andere ist. Da der Algorithmus eine gewisse Zeit für die Berechnung benötigt, kann in der Zwischenzeit die nach der letzten Berechnung noch optimale Route sehr schlecht sein. Hingegen beginnt AntNet als dynamisches Verfahren sofort mit Entstehen der Verkehrsänderungen, seine bisherige Lösung anzupassen und kann dadurch möglicherweise einen Vorteil gegenüber Dijkstra erlangen. Ob der Vorteil groß genug ist, um zu kompensieren, dass Dijkstra im Gegensatz zu AntNet zwischenzeitlich das tatsächlich optimale Ergebnis liefert, wird in diesem Fall überprüft.

### 4.4.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei bei jeder Ausführung neue Verkehrsänderungen zufällig erzeugt werden. Die Stauerzeugung folgt den unten zusammengefassten Parametern. Als Endpunkte dienen die in Abbildung 8 dargestellten. Danach werden mehrere Ausführungen mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden wie zuvor beim Dijkstra-Algorithmus. Jede Ausführung läuft 5 Sekunden länger als die davor.

| | |
|---|---|
| *Positive Verkehrsänderungen erlauben:* | Ja |
| *Ungelenkter Zufall:* | Ja |
| *Stauerzeugungsfequenz:* | *100* |
| *Ausgabefrequenz:* | 100 |
| *Grundfaktor:* | 0.25 |
| *Maximalfaktor:* | 1 |

### 4.4.3 Auswertung

Abbildung 11 zeigt die Veränderung der gerade am besten bewerteten Route im Verlauf der Programmausführung an drei Beispielen. Kurven, die eine identische Nummer besitzen, sind das Ergebnis von Ausführungen, die mit denselben Verkehrsänderungen getestet wurden. Aufgrund der wesentlich häufiger erzeugten Verkehrsänderungen schwankt die Lösung des Dijkstra-Algorithmus erheblich stärker als noch in 4.3. Es sind aber weiterhin auch längere konstante Abschnitte vorhanden.

In den Beispielen 1 und 4 gelingt es AntNet kurzzeitig bessere Ergebnisse als Dijkstra zu liefern, was dem Verfahren in den vorherigen Fällen nicht gelingt. Wie bereits in 4.4.1 angenommen, kommt es vermutlich aufgrund der stark erhöhten Häufigkeit und Stärke der Verkehrsänderungen zur Häufung von relevanten Änderungen im Umfeld der besten Route. Die vorliegende Situation vor und nach der Dijkstra-Berechnung kann sich dadurch stärker unterscheiden. Die berechnete Lösung in der neuen Situation würde dann lange genug schlecht bleiben, dass die Näherungslösung von AntNet dichter an der neuen optimalen Route liegt.

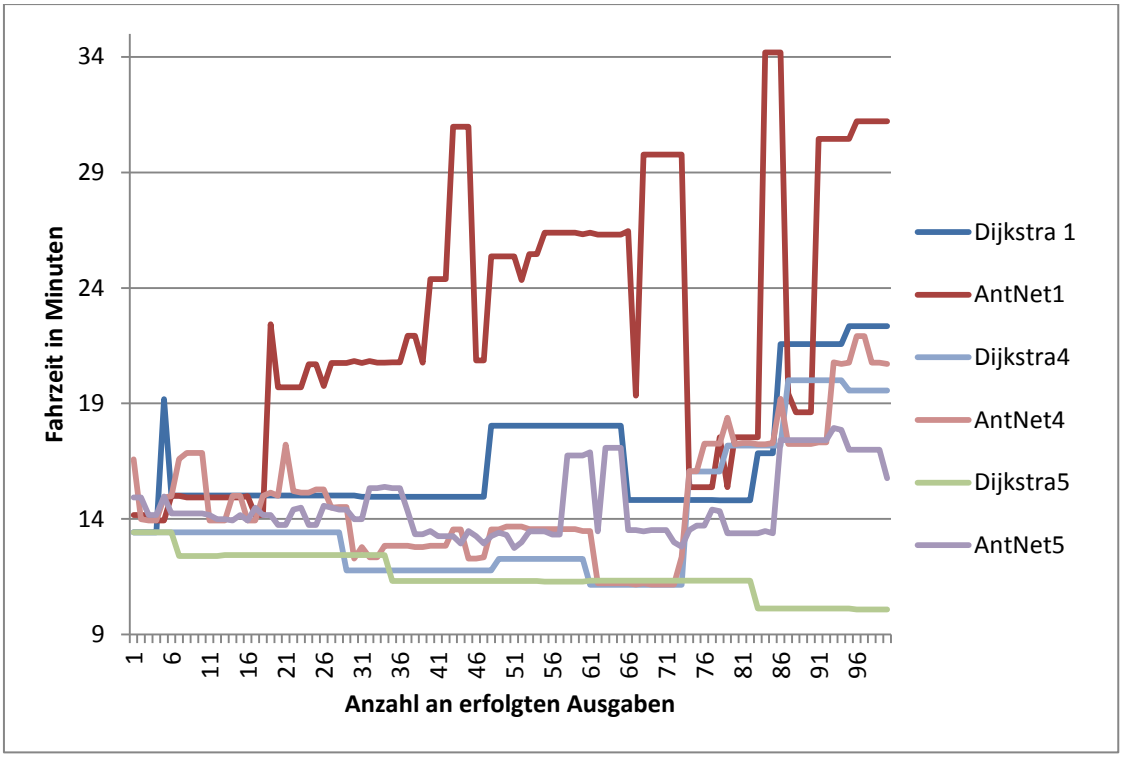


**Abbildung 11: Programmausführungen mit erhöhtem Stauaufkommen**

Durchschnittlich schneidet AntNet aber trotzdem in jeder Ausführung schlechter ab als der Dijkstra-Algorithmus. Die Verschlechterung nimmt im Vergleich zum vorherigen Fall sogar zu. Eine erhöhte Staudichte scheint sich also bei einer reinen Zufallsverteilung durchschnittlich besser auf den Dijkstra-Algorithmus auszuwirken als auf das AntNet-Verfahren.

Wie viel schlechter AntNet durchschnittlich gegenüber Dijkstra abschneidet, ist im Folgenden zusammengefasst:

Test 1: Ameisen sind durchschnittlich 35.04 % schlechter als Dijkstra.

Test 2: Ameisen sind durchschnittlich 35.52 % schlechter als Dijkstra.

Test 3: Ameisen sind durchschnittlich 59.43 % schlechter als Dijkstra.

Test 4: Ameisen sind durchschnittlich 9.68 % schlechter als Dijkstra.

Test 5: Ameisen sind durchschnittlich 47.81 % schlechter als Dijkstra.

Das vergleichsweise überdurchschnittlich gute Ergebnis in Test 4 zeigt, dass der Zufall der Stauerzeugung einen erheblichen Einfluss auf die Lösungsgüte besitzt. Der tatsächliche Einfluss der Staudichte wird aufgrund dessen in weiteren Fällen nochmal genauer untersucht.

Es lässt sich feststellen, dass bei einer sehr hohen Staudichte AntNet tatsächlich zeitweise eine bessere Lösung liefern kann als der Dijkstra-Algorithmus. Die durchschnittliche Lösungsgüte profitiert aber nicht davon. Es müssen also Fälle gefunden werden, in denen der Umstand, der zur temporären Überlegenheit der Lösung des AntNet-Verfahrens gegenüber dem Dijkstra-Algorithmus geführt hat, möglichst häufig erzeugt wird, ohne dabei eine zu große Gesamtstaudichte zu erzeugen. Da im Rahmen dieser Arbeit die Tauglichkeit der beiden Verfahren für reale Verkehrsprobleme verglichen werden soll, orientieren sich die weiteren Fälle noch mehr an speziellen realistischen Verkehrssituationen.

Jede der oben dargestellten Ausführungen hatte eine jeweils 5 Sekunden längere Ausführzeit als die jeweilige Ausführung davor. Da die Differenz der Lösungsqualität der beiden Verfahren in Test 4 erheblich sinkt, aber in Test 5 wieder erheblich steigt, scheint kein Bezug der Laufzeit auf die Lösungsqualität zu bestehen. Dies bestärkt die Annahme, die in 4.3.3 getroffen wurde und wird in 4.8 noch einmal gezielt überprüft. Bis dahin wird auf eine erneute Untersuchung diesbezüglich verzichtet.

## 4.5 Dynamisches Routing mit Beschränkung auf echte Staus

Bei diesem dynamischen Fall treten Verkehrsänderungen eingeschränkt zufällig auf. Der Ort ist weiterhin vollkommen zufällig, die Ausprägung ist auf negative Verkehrsänderungen und damit echte Staus beschränkt. Start- und Zielpunkt sind konstant. Es wird sowohl mit niedriger als auch mit erhöhter Staudichte getestet.

### 4.5.1 Theoretische Überlegung

Traditionelle Navigationsgeräte schlagen zu Beginn der Fahrt eine Route vor. Diese wird anschließend nur noch selten geändert. Dies liegt vor allem daran, dass in natürlichen Verkehrsnetzen Verkehrsänderungen immer als tatsächliche Verkehrsbehinderungen definiert sind, die Durchschnittsfahrgeschwindigkeit eines Abschnitts mit Stau wird also grundsätzlich reduziert. Dieses Verhalten soll in diesem Fall untersucht werden.

Tritt ein Stau auf einem Abschnitt auf, der nicht Teil der aktuell besten Route ist, werden alternative Routen höchstens noch unattraktiver. Die beste Route bleibt also als diese erhalten. Nur wenn ein Stau auf einer Kante auftritt, die zur bisher besten Route gehört, kann diese dadurch so viel schlechter werden, dass eine andere Route zur neuen optimalen Lösung wird. Da die Verkehrsänderungen weiterhin zufallsverteilt bleiben, kann angenommen werden, dass nur selten tatsächlich ein Abschnitt des aktuell besten Weges davon betroffen ist und sich dieser daher nur selten ändert. Wie in vorherigen Fällen bereits beobachtet, profitiert der Dijkstra-Algorithmus erheblich davon, wenn die optimale Lösung konstant bleibt. Dem AntNet-Verfahren hingegen kommt es zu Gute, wenn ein Stau einen Abschnitt verschlechtert, der zu Lösungen gehört, die möglichst nah am Optimum sind, ohne dieses tatsächlich zu sein. Die Ameisen werden dann mehr in Richtung tatsächlicher Lösung gelenkt. Es gibt also einige Kanten, bei denen sich Stau negativ für den Dijkstra-Algorithmus auswirkt und einige Kanten bei denen sich Stau positiv auf das AntNet-Verfahren auswirkt. Beide Verfahren profitieren also theoretisch von den Umständen dieses Szenarios.

In diesem Fall werden zwei Szenarien unterschieden: Einmal mit niedriger und einmal mit hoher Staudichte. Auf diese Weise wird erneut untersucht, in wie weit die Staudichte sich auf die Lösungsqualität auswirkt. Aus 4.4 ist bereits eine Tendenz bekannt, die hier noch einmal überprüft werden soll.

### 4.5.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei bei jeder Ausführung neue Verkehrsänderungen zufällig erzeugt werden. Die Stauerzeugung folgt den unten zusammengefassten Parametern. Die Werte der beiden Zeilen teilen sich auf die Ausführungen auf. Als Endpunkte dienen die in Abbildung 8 dargestellten. Danach werden mehrere Ausführungen mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden wie zuvor beim Dijkstra-Algorithmus.

| | | |
|---|---|---|
| *Positive Verkehrsänderungen erlauben:* | Nein | Nein |
| *Ungelenkter Zufall:* | Ja | Ja |
| *Stauerzeugungsfequenz:* | 500 | 100 |
| *Ausgabefrequenz:* | 250 | 50 |
| *Grundfaktor:* | 0.1 | 0.25 |
| *Maximalfaktor:* | 9 | 1 |

### 4.5.3 Auswertung

Abbildung 12 und Abbildung 13 zeigen die Veränderungen der aktuell als beste angegebene Route im Verlauf der Programmausführung an jeweils drei Beispielen. Kurven, die eine identische Nummer besitzen, sind das Ergebnis von Ausführungen, die mit denselben Verkehrsänderungen getestet wurden. Abbildung 12 stellt die Ausführungen mit wenigen, Abbildung 13 die mit vielen Verkehrsänderungen dar.

Wie in Abbildung 12 zu sehen ist, sind die Ergebnisse bei geringer Staudichte vergleichbar mit denen von 4.3. Der Dijkstra-Algorithmus liefert über längere Phasen konstante Lösungen. Das bedeutet, dass die als jeweils beste angezeigte Route auch die tatsächlich optimale Route ist. AntNet besitzt die üblichen und von der Implementierung vorgesehenen Fluktuationen und schafft es dabei nur in kurzen Momenten, ebenfalls die optimale Lösung zu liefern. Durchschnittlich setzt sich erneut der Dijkstra-Algorithmus im direkten Vergleich durch.

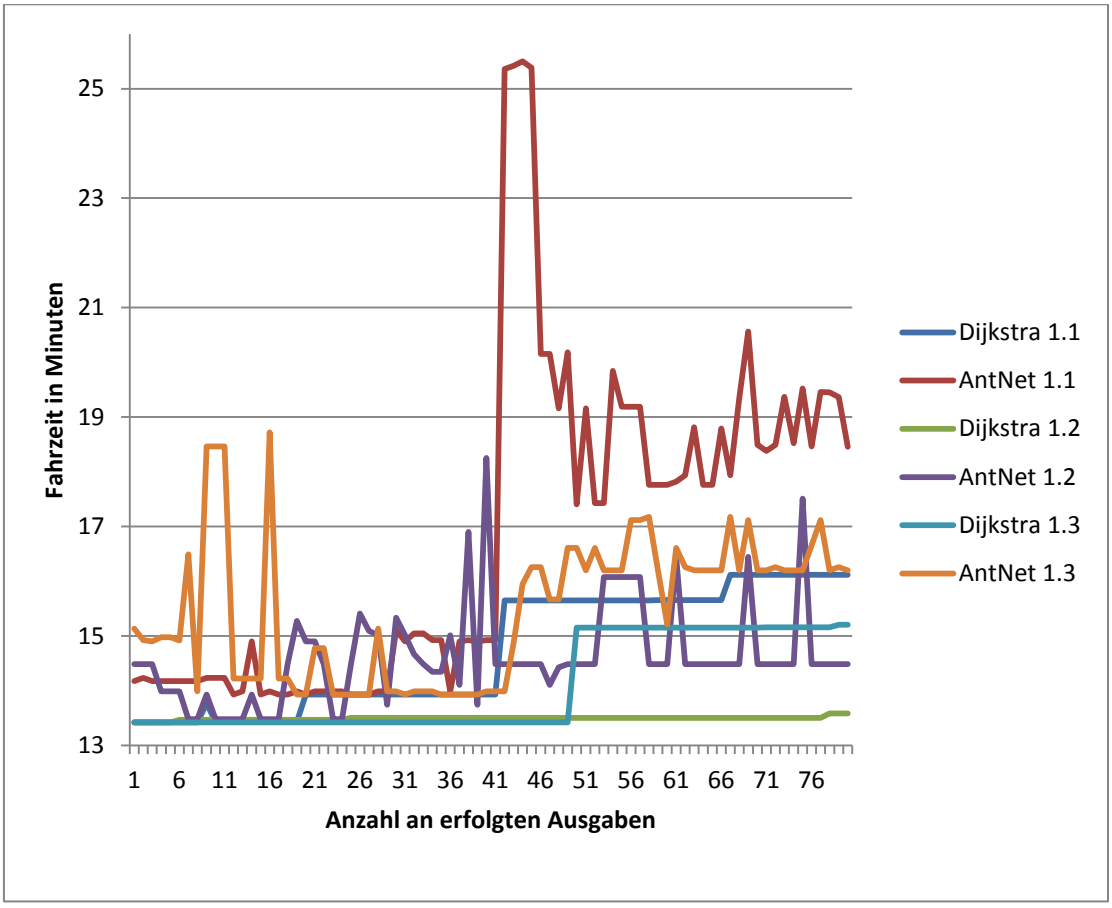


**Abbildung 12: Ausführungen bei wenigen Verkehrsänderungen**

Wie viel schlechter das AntNet-Verfahren im Vergleich mit dem Dijkstra-Algorithmus abschneidet, ist im Folgenden zusammengefasst:

Test 1.1: Ameisen sind durchschnittlich 13.46 % schlechter als Dijkstra.

Test 1.2: Ameisen sind durchschnittlich 8.32 % schlechter als Dijkstra.

Test 1.3: Ameisen sind durchschnittlich 9.83 % schlechter als Dijkstra.

Die durchschnittliche Differenz in der Lösungsgüte der beiden Verfahren ist mit der in 4.3 vergleichbar. Die Beschränkung auf negative Kantenentwicklungen scheint also bei geringer Staudichte wie in 4.5.1 vermutet keines der beiden Verfahren zu bevorteilen.

Wie in Abbildung 13 zu sehen ist, sind die Ergebnisse bei hoher Staudichte vergleichbar mit denen von 4.4. Auch hier kann AntNet für kurze Momente bessere Lösungen als Dijkstra liefern, schlägt sich aber durchschnittlich sehr viel schlechter. Es gelten die gleichen Gründe, die bereits dort diskutiert wurden.

Wie in Abbildung 13 auch zu sehen ist, steigen die gefundenen Lösungen kontinuierlich an. Dies ist eine Folge der Begrenzung auf negative Verkehrsänderungen. Auf die Realität übertragen stellt dies die Situation dar, wenn die dem Stau ausweichenden Fahrzeuge auf der Umgehungsstraße ebenfalls einen Stau erzeugen und die Staus immer länger werden. In Abbildung 12 ist dies wesentlich schwächer ausgeprägt, da die Staudichte dort geringer ist.

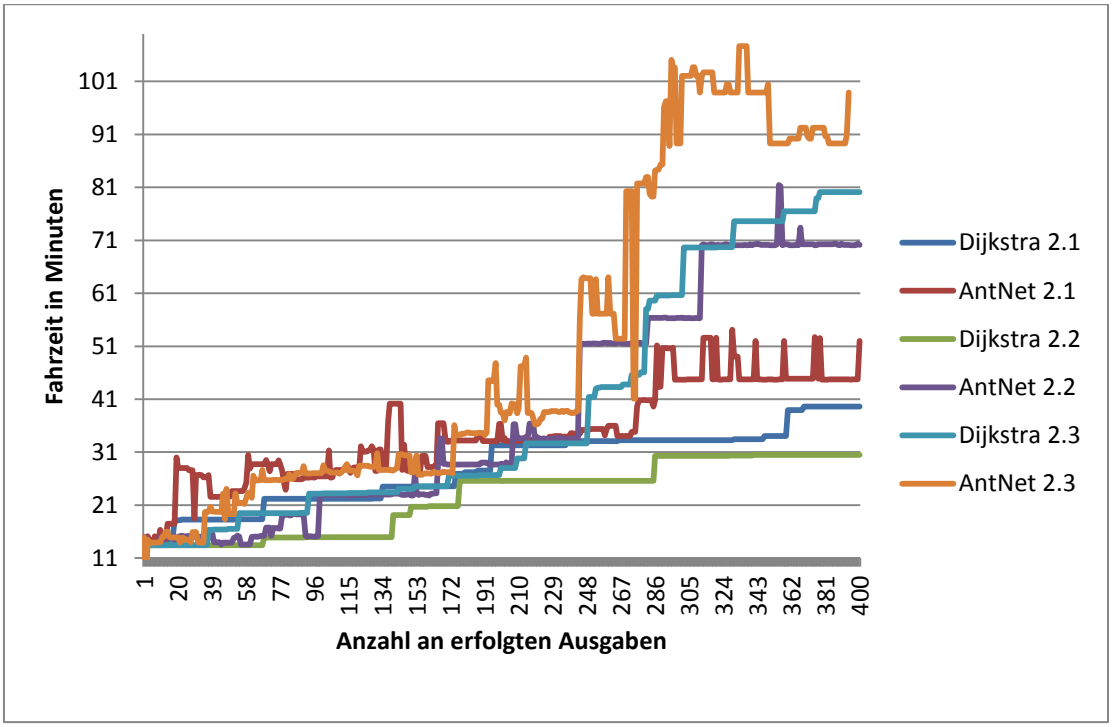


**Abbildung 13: Ausführungen bei vielen Verkehrsänderungen**

Wie viel schlechter das AntNet-Verfahren im Vergleich mit dem Dijkstra-Algorithmus abschneidet, ist im Folgenden zusammengefasst:

Test 2.1: Ameisen sind durchschnittlich 24.14 % schlechter als Dijkstra.

Test 2.2: Ameisen sind durchschnittlich 59.98 % schlechter als Dijkstra.

Test 2.3: Ameisen sind durchschnittlich 27.74 % schlechter als Dijkstra.

Die durchschnittliche Differenz in der Lösungsgüte der beiden Verfahren ist mit der in 4.4 vergleichbar. Die Beschränkung auf negative Kantenentwicklungen scheint also auch bei hoher Staudichte keines der beiden Verfahren zu bevorteilen.

Da sich die Art der auftretenden Verkehrsänderungen unabhängig von der Staudichte offenbar nicht auf die Differenz der Lösungsgüte auswirkt, kann dieser Parameter bei der Analyse von weiteren Fällen mit rein zufällig erzeugten Verkehrsänderungen ignoriert werden. Ob sich dies bei bedingtem Zufall anders verhält, wird in 4.6 noch untersucht.

Der Fall stützt die Vermutung, dass eine hohe Staudichte AntNet durchschnittlich mehr behindert als Dijkstra. Dies wird in 4.7 noch einmal gezielt überprüft. Bis dahin wird auf eine erneute Untersuchung diesbezüglich verzichtet.

## 4.6 Dynamisches Routing mit gezielten Verkehrsänderungen

Bei diesem dynamischen Fall treten Verkehrsänderungen nur begrenzt zufällig auf. Dies gilt für Ort und Ausprägung. Verkehrsänderungen werden nur an Kanten erzeugt, die Teil der besten Route sind. Die Stärke wird zufällig bestimmt. Start- und Zielpunkt sind konstant. Es wird sowohl mit auf negative Verkehrsänderungen beschränkter als auch mit zufälliger Ausprägung gearbeitet.

### 4.6.1 Theoretische Überlegung

Verkehrsänderungen entstehen in der Realität nicht zufällig, sondern verstärkt an besonders stark frequentierten Routen. Diese sind meistens deshalb stark frequentiert, weil sie besonders schnell zum Ziel führen. Navigationsgeräte schlagen diese Routen daher auch bevorzugt vor. Viele Fahrzeuge werden gleichzeitig dort entlang geleitet, das Verkehrsaufkommen steigt und ein Stau bildet sich. Gute Navigationsgeräte erkennen das und schlagen eine Alternativroute vor. Machen das aber alle Navigationsgeräte, verlagert sich der Stau nur auf die neue Route. Um dieses Verhalten zu simulieren, werden in diesem Fall Verkehrsänderungen nur entlang der von dem Dijkstra-Algorithmus aktuell als beste geltenden Route erzeugt.

Dabei werden zwei Szenarien unterschieden: Einmal mit nur negativen Verkehrsänderungen und einmal mit positiven sowie negativen Verkehrsänderungen. Für den allgemeinen Fall wurde in 4.5 bereits gezeigt, dass die Art der Verkehrsänderung keines der beiden Verfahren bevorzugt. Hier liegt allerdings ein Spezialfall vor. Da bei negativen Verkehrsänderungen jedes Mal ein Abschnitt des besten Weges verschlechtert wird, ändert sich dieser häufig. Es wurde bereits gezeigt, dass Dijkstra von einer konstant bleibenden optimalen Route

profitiert. Häufige Wechsel wirken sich hingegen negativ auf die Lösungsgüte des Algorithmus aus. Werden stattdessen positive und negative Verkehrsänderungen erzeugt, sorgt jede positive Veränderung einer Kante dafür, dass dieser Abschnitt als Teil der optimalen Route bestärkt wird. Es kommt also zu weniger Fluktuationen bei der optimalen Route, was sich vermutlich wiederum positiv auf die Lösungsgüte des Dijkstra-Algorithmus auswirkt.

AntNet liefert nur selten die optimale Lösung, daher beeinträchtigen gezielte Änderungen an dieser das Verfahren vermutlich weniger als den Dijkstra-Algorithmus. Verstärkte positive Veränderung der Abschnitte entlang der optimalen Lösung hingegen könnten die Pheromone mehr in Richtung des Optimums fokussieren und die Fluktuationen des Verfahrens reduzieren, was die durchschnittliche Lösungsgüte verbessern würde. In wie weit tatsächlich eines der beiden Verfahren von einem der beiden Szenarien stärker profitiert als das andere, wird hier untersucht.

### 4.6.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei bei jeder Ausführung neue Verkehrsänderungen zufällig erzeugt werden. Die Stauerzeugung folgt den unten zusammengefassten Parametern. Die Werte der beiden Zeilen teilen sich auf die Ausführungen auf. Als Endpunkte dienen die in Abbildung 8 dargestellten. Danach werden mehrere Ausführungen mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden wie zuvor beim Dijkstra-Algorithmus.

| | | |
|---|---|---|
| *Positive Verkehrsänderungen erlauben:* | Nein | Ja |
| *Ungelenkter Zufall:* | Nein | Nein |
| *Stauerzeugungsfequenz:* | 500 | 500 |
| *Ausgabefrequenz:* | 250 | 250 |
| *Grundfaktor:* | 0.1 | 0.1 |
| *Maximalfaktor:* | 9 | 9 |

### 4.6.3 Auswertung

Abbildung 14 und Abbildung 15 zeigen die Veränderungen der gerade am besten bewerteten Route im Verlauf der Programmausführung an jeweils drei Beispielen. Kurven, die eine identische Nummer besitzen, sind das Ergebnis von Ausführungen, die mit denselben Verkehrsänderungen getestet wurden. Abbildung 14 stellt die Ausführungen mit nur echten Staus, Abbildung 15 die mit positiven und negativen Verkehrsänderungen dar.

Wie in Abbildung 14 zu sehen ist, bleiben die vom Dijkstra-Algorithmus berechneten besten Wege durch das Begrenzen auf gezielte negative Verkehrsänderungen wie angenommen nie lange konstant, sondern sind stetigen Fluktuationen unterworfen. Dem AntNet-Verfahren gelingt es mehrmals kurzzeitig bessere Lösungen zu liefern als Dijkstra, was bei den vorherigen Fällen nur bei erhöhter Staudichte vorkommt. Die optimale Route wird durch einen Stau so verschlechtert, dass die neue optimale Route näher an der Näherungslösung des AntNet-Verfahrens als an der alten Lösung liegt. Die Lösung des Dijkstra-Algorithmus, die noch auf der Situation des alten Optimums basiert, ist dadurch schlechter. Dass sich AntNet schneller als Dijkstra an veränderte Optima anpasst, lässt sich anhand der Untersuchungen der vorherigen Fälle mit erhöhtem Stauaufkommen ausschließen. Dass die von AntNet gefundene Näherungslösung also in einigen Situationen die Lösung von Dijkstra schlägt, ist in der heuristischen Natur des Verfahrens begründet und unterliegt dem Zufall.

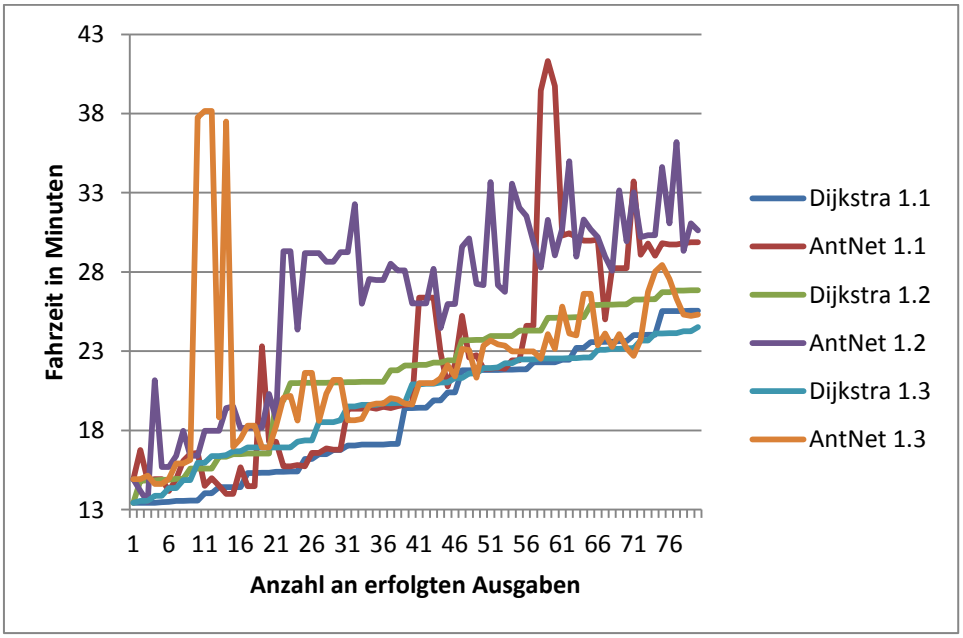


**Abbildung 14: Ausführungen bei reinen Staus**

Durchschnittlich kann sich aber auch in diesem Fall AntNet nicht gegen Dijkstra durchsetzen. Wie viel schlechter das AntNet-Verfahren gegenüber dem Dijkstra-Algorithmus abschneidet, ist im Folgenden zusammengefasst:

Test 1.1: Ameisen sind durchschnittlich 15.31 % schlechter als Dijkstra.

Test 1.2: Ameisen sind durchschnittlich 21.13. % schlechter als Dijkstra

Test 1.3: Ameisen sind durchschnittlich 12.42 % schlechter als Dijkstra.

Die Differenz der Lösungsgüten der beiden Verfahren fällt geringer aus als bei den Fällen mit erhöhtem Stauaufkommen (siehe 4.4 und 4.5) und ähnelt der beim allgemeinen Fall (siehe 4.3).

Wie in Abbildung 15 zu sehen ist, unterliegen die vom Dijkstra-Algorithmus berechneten besten Wege auch beim Erzeugen von beliebigen gezielten Verkehrsänderungen starken Schwankungen. Anders als bei der Begrenzung auf echte Verkehrsänderungen gelingt es dem AntNet-Verfahren hier aber kaum, Lösungen zu liefern, die jene des Dijkstra-Algorithmus schlagen. Die Lösungsgüte im Vergleich mit Dijkstra ist im Durchschnitt schlechter als bei einer Begrenzung auf nur negativen Verkehrsänderungen. Die Annahme, dass in diesem speziellen Fall die Unterscheidung der Stautypen eine Rolle spielt, bestätigt sich also.

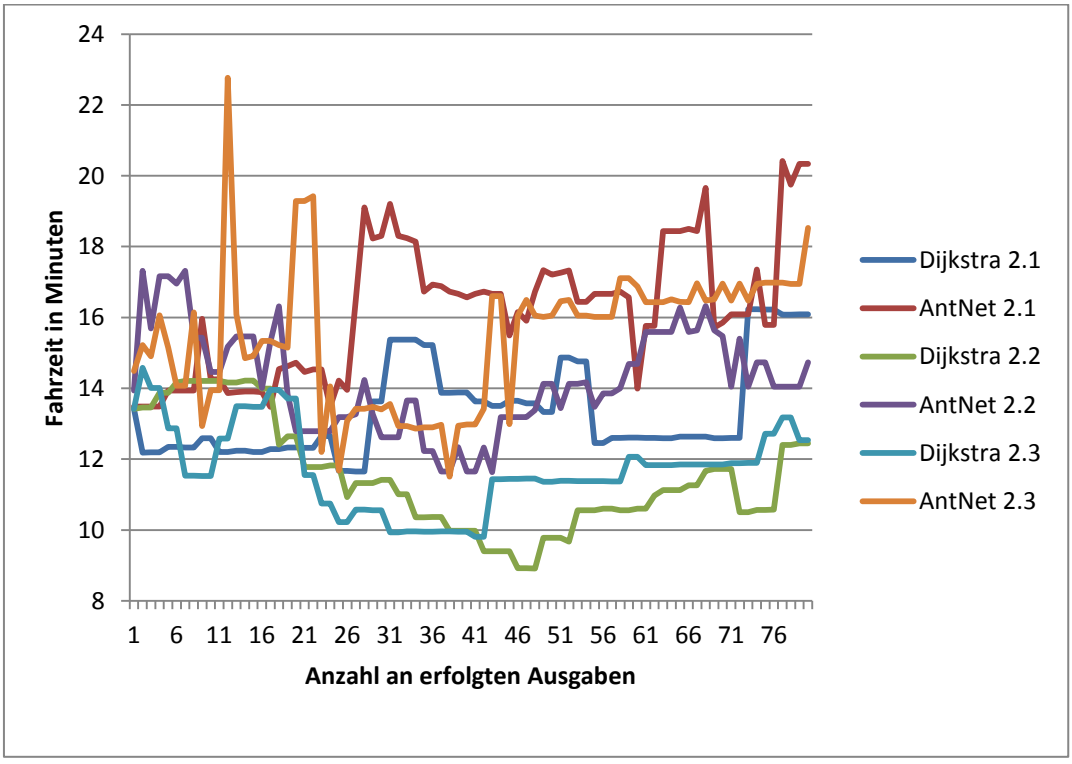


**Abbildung 15: Ausführungen bei positiven und negativen Verkehrsänderungen**

Wie viel Schlechter das AntNet-Verfahren gegenüber dem Dijkstra-Algorithmus abschneidet, ist im Folgenden zusammengefasst:

Test 2.1: Ameisen sind durchschnittlich 21.64 % schlechter als Dijkstra.

Test 2.2: Ameisen sind durchschnittlich 24.65 % schlechter als Dijkstra.

Test 2.3: Ameisen sind durchschnittlich 32.07 % schlechter als Dijkstra.

Auch bei gezielten Verkehrsänderungen insgesamt schlägt sich der Dijkstra-Algorithmus insgesamt besser als das AntNet-Verfahren. Dies ist unabhängig von der Ausprägung der Verkehrsänderungen.

## 4.7 Dynamisches Routing mit variablen Staudichten

Bei diesem dynamischen Fall treten Verkehrsänderungen vollkommen zufällig auf. Dies gilt für Ort und Ausprägung. Die Staustärke sowie Start- und Zielpunkt sind konstant. Die Staudichte variiert.

### 4.7.1 Theoretische Überlegung

In verschiedenen Fällen hat sich der Verdacht erhärtet, dass AntNet bei einer hohen Staudichte größere Probleme besitzt als Dijkstra. Daher wird hier überprüft, wie der genaue Bezug der Staudichte zur Lösungsqualität der beiden Verfahren ist. Beim statischen Routing in 4.2 hat AntNet zwar schlechter abgeschnitten als der Dijkstra-Algorithmus, aber in den meisten dynamischen Fällen ist die Differenz in der Lösungsgüte noch wesentlich größer. Dies lässt vermuten, dass AntNet im Vergleich mit Dijkstra besser arbeitet, je niedriger die Staudichte ist.

### 4.7.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei bei jeder Ausführung neue Verkehrsänderungen zufällig erzeugt werden. Die Stauerzeugung folgt den unten zusammengefassten Parametern. Jede Ausführung wird mit einer anderen Staudichte ausgeführt. Als Endpunkte dienen die in Abbildung 8 dargestellten. Danach werden mehrere Ausführungen mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden, wie zuvor beim Dijkstra-Algorithmus.

| | |
|---|---|
| *Positive Verkehrsänderungen erlauben:* | Nein |
| *Ungelenkter Zufall:* | Ja |
| *Stauerzeugungsfequenz:* | 4000 2000 1000 500 |
| | 250 125 60 30 |
| *Ausgabefrequenz:* | 100 |
| *Grundfaktor:* | 0.25 |
| *Maximalfaktor:* | 1 |

### 4.7.3 Auswertung

Das AntNet-Verfahren kann sich unabhängig von der Staudichte nicht gegen den Dijkstra-Algorithmus durchsetzen. Abbildung 16 zeigt, wie sich die Differenz der Lösungsgüte der beiden Verfahren in Abhängigkeit von dem zeitlichen Abstand zwischen zwei Stauerzeugungen verhält. Je niedriger dieser Abstand, desto größer ist die Staudichte. Es lässt sich am Graphen ein klarer Trend feststellen. Dieser unterstützt die bisherige Vermutung, dass eine erhöhte Staudichte sich negativer auf das AntNet-Verfahren auswirkt als auf den Dijkstra-Algorithmus.

Ein Erklärungsversuch erfordert das genaue Betrachten des Verhaltens der beiden Verfahren. Je häufiger Verkehrsänderungen auftreten und sich die Verkehrssituation ändert, desto schneller sind Pheromonwerte veraltet. Das Ameisensystem benötigt dann eine gewisse Zeit, bis die Ameisen diese angepasst haben. Daher kommt es bei einer zu schnellen Stauerzeugung nicht zeitnah hinterher. Der Dijkstra-Algorithmus, wie bei mehreren vorherigen Fällen bereits diskutiert, leidet unter dem Umstand, dass sich bei erhöhtem Stauaufkommen häufiger die Situation während der Berechnung stark verändert und der berechnete Wert dann weit von der neuen optimalen Route entfernt liegt. Allerdings sind für den Dijkstra-Algorithmus im Gegensatz zu AntNet vorherige Verkehrsänderungen irrelevant, es muss sich also nicht mit rechnerischen Altlasten herumgeschlagen werden. Beide Verfahren leiden unter einer hohen Staudichte, wobei die Nachteile des AntNet-Verfahrens schwerer zu wiegen scheinen. Woran genau das liegt, wird in weiteren Fällen untersucht.

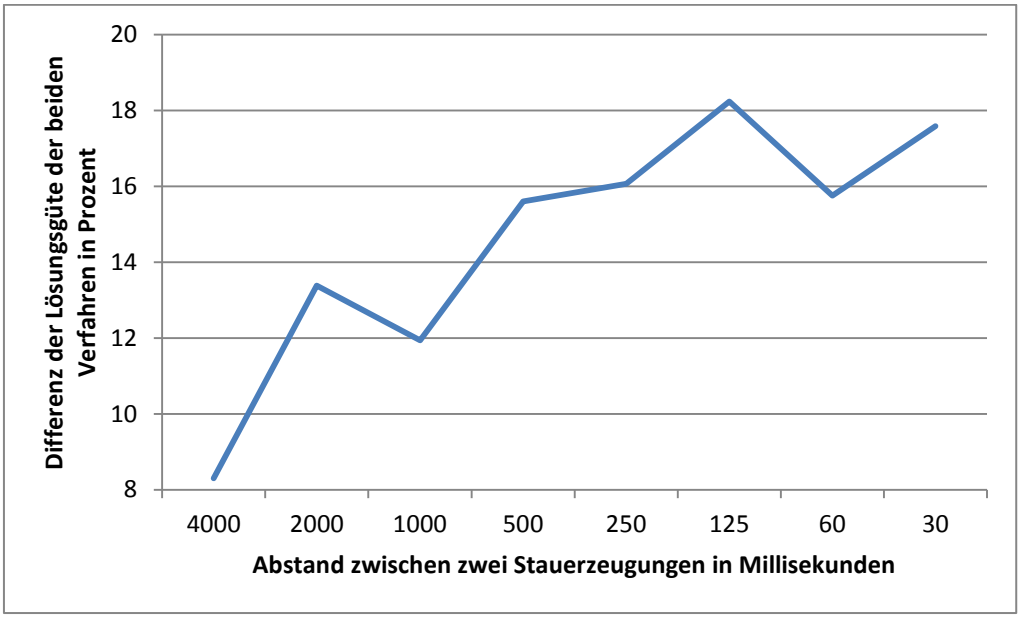


**Abbildung 16: Prozentuale Differenz der Lösungsqualität beider Verfahren**

## 4.8 Dynamisches Routing mit steigender Ausführzeit

Bei diesem dynamischen Fall treten Verkehrsänderungen vollkommen zufällig auf. Dies gilt für Stärke, Ort und Ausprägung. Start- und Zielpunkt sind konstant. Die Ausführzeit variiert.

### 4.8.1 Theoretische Überlegung

Nach den Fällen in 4.3 und 4.4 stellte sich die Vermutung, dass die Ausführzeit von AntScout sich nicht auf die Differenz der Lösungsgüte der beiden Verfahren auswirkt. Diese Vermutung wird hier überprüft. Die Berechnung durch den Dijkstra-Algorithmus betrachtet nur die aktuelle Situation im Verkehrsnetz, daher kann davon ausgegangen werden, dass der Algorithmus unabhängig von der Ausführzeit ist. Beim AntNet-Verfahren werden kontinuierlich neue Ameisen erzeugt und später wieder aufgelöst. Außerdem basieren die Pheromonwerte an den Knoten auf den vergangenen Veränderungen des Graphen. Ob sich dadurch die Ausführzeit auf die Lösungsqualität des AntNet-Verfahrens auswirkt, wird hier untersucht.

### 4.8.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei bei jeder Ausführung neue Verkehrsänderungen zufällig erzeugt werden. Die Stauerzeugung folgt den unten zusammengefassten Parametern. Als Endpunkte dienen die in Abbildung 8 dargestellten. Anschließend werden mehrere Ausführungen mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden, wie zuvor beim Dijkstra-Algorithmus. Es wird eine Ausführzeit von einer Minute verwendet. Dann werden die Lösungsgüten alle 10 Sekunden verglichen.

| | |
|---|---|
| *Positive Verkehrsänderungen erlauben:* | Nein |
| *Ungelenkter Zufall:* | Ja |
| *Stauerzeugungsfequenz:* | 100 |
| *Ausgabefrequenz:* | 50 |
| *Grundfaktor:* | 0.1 |
| *Maximalfaktor:* | 9 |

### 4.8.3 Auswertung

Abbildung 17 zeigt die Entwicklung der Differenz der Lösungsgüte der beiden Verfahren im Verlauf der Ausführzeit in Prozent. Die x-Achse stellt die Anzahl an Ausgaben dar. Es lässt sich erkennen, dass die Lösungsgütedifferenz mit steigender Ausführzeit ebenfalls anwächst und erst kurz vor Ende wieder leicht sinkt. Der rasche Anstieg lässt entgegen der Beobachtungen in 4.3 und 4.4 vermuten, dass die Ausführzeit sich doch negativ auf das AntNet-Verfahren auswirkt. Dem entgegen steht das Absinken der Differenz zum Ende der Ausführzeit. In allen drei Beispielen liegt diese vor, daher kann der Zufall als Begründung faktisch ausgeschlossen werden. Es steht damit im Widerspruch mit der gerade formulierten Vermutung.

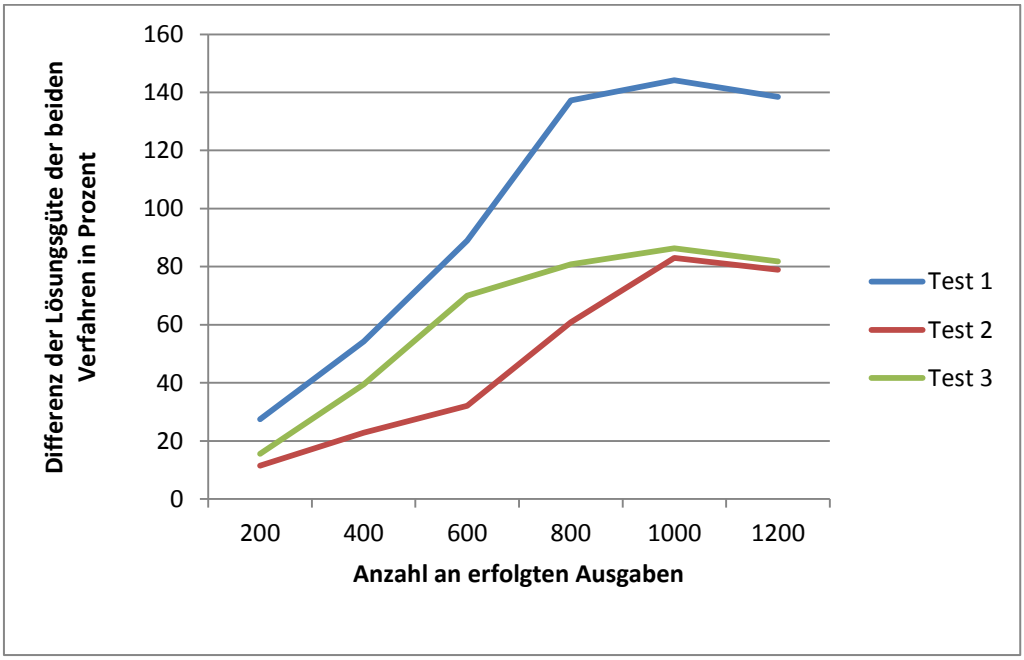


**Abbildung 17: Beste Wege im Verlauf der Ausführzeit**

Abbildung 18 zeigt die von den beiden Verfahren gelieferte Lösung in Abhängigkeit der vergangenen Ausführzeit an zwei Beispielen. Wie zu sehen ist, steigen die gefundenen Lösungen kontinuierlich an. Dies wurde bei allen anderen Fällen mit einer Begrenzung auf negative Verkehrsänderungen ebenfalls beobachtet. Die durchschnittliche Lösungsgüte des AntNet-Verfahrens steigt schneller an als die des Dijkstra-Algorithmus. Zum Ende hin stagniert die Lösung von AntNet und bewegt sich nur noch in einem festen Bereich. Dies ist eine Folge davon, dass inzwischen die Durchschnittsfahrgeschwindigkeit an fast allen Abschnitten in Lösungsnähe so gering ist, dass die Pheromonbewertung nahe 0 liegt. Die Bereitschaft des Ameisensystems, neue Wege auszuprobieren, ist so gut wie nicht mehr vorhanden. Nur das Untersuchen aller Wege führt weiterhin zu einer Anpassung der besten Route. Die Lösung des Dijkstra-Algorithmus steigt daher weiter an und die Differenz der Lösungsqualität der beiden Verfahren sinkt. Dieser Umstand erklärt den Knick am Ende jedes Graphen in Abbildung 17 und löst den dadurch verursachten Widerspruch auf.

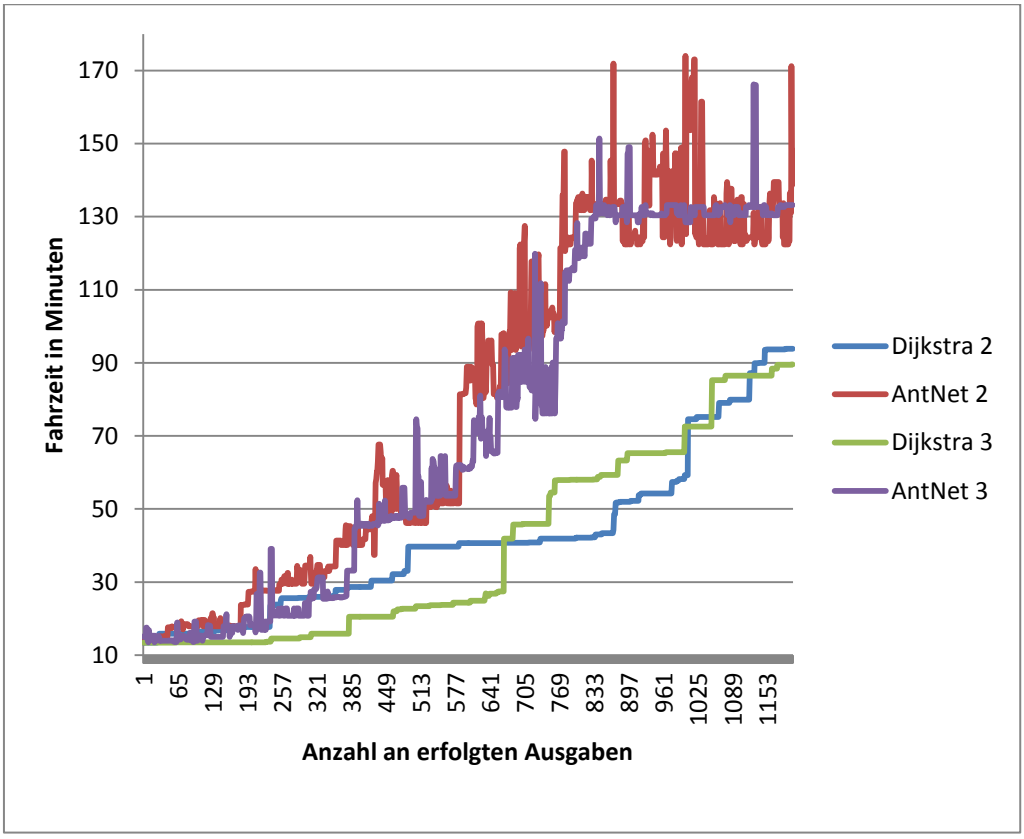


**Abbildung 18: Ausführungen bei steigender Ausführzeit**

Sich immer weiter verdichtende Verkehrsänderungen sind also tatsächlich ein Problem für das AntNet-Verfahren. Dies tritt nur bei der Begrenzung auf negative Verkehrsänderungen auf. Diese liegt bei den Fällen, die in 4.3 und 4.4 untersucht werden, nicht vor. Das erklärt den sich unterscheidenden Einfluss der Ausführzeit auf die Ergebnisse. Abschließend lässt sich daher zusammenfassen: Wird die Stauausprägung zufällig bestimmt, ist die Ausführzeit irrelevant. Wird sie auf negative Verkehrsänderungen beschränkt, leidet AntNet unter einem Anstieg der Ausführzeit. Der rasche Anstieg des Graphen in 4.7 passt zu dieser Erkenntnis, da eine höhere Staudichte bei einer Begrenzung auf negative Verkehrsänderungen bereits bei geringerer Laufzeit zu stark verdichteten Verkehrsänderungen führt. Die Gesamtzahl an erzeugten Verkehrsänderungen ist also der entscheidende Faktor beim Vergleich der beiden Verfahren.

## 4.9 Dynamisches Routing mit verschiedenen Start- und Endpunkten

Bei diesem dynamischen Fall variieren Start- und Zielpunkt. Verkehrsänderungen treten zufällig auf. Stärke und Ausprägung sind konstant.

### 4.9.1 Theoretische Überlegung

In allen vorherigen Fällen werden dieselben Start- und Endpunkte verwendet. Die optimale Route ändert sich bei hoher Staudichte sehr häufig und die bisherigen Endpunkte decken den Graphen fast in seiner kompletten Breite ab. Es ist daher davon auszugehen, dass ein Großteil aller möglichen Routen von den bisherigen Fällen abgedeckt wird. Die bei der Wahl von anderen Endpunkten entstehenden Routen verhalten sich bei ausreichend langer Ausführzeit daher vermutlich nicht anders als diese. Ob diese Vermutung stimmt, wird in diesem Fall überprüft. Dafür werden zwei verschiedene Endpunktkombinationen betrachtet. Beide unterscheiden sich von der bisher genutzten. Die erste Kombination besteht aus zwei dicht beieinander liegenden, die zweite aus zwei weit entfernt von einander liegenden Endpunkten.

### 4.9.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei eine Hälfte mit den Endpunkten aus Abbildung 19 und eine Hälfte mit den Endpunkten aus Abbildung 20 durchgeführt wird. Verkehrsänderungen werden zufällig erzeugt. Die Stauerzeugung folgt den unten zusammengefassten Parametern. Mehrere Ausführungen werden anschließend mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden, wie zuvor beim Dijkstra-Algorithmus.

| | |
|---|---|
| *Positive Verkehrsänderungen erlauben:* | nein |
| *Ungelenkter Zufall:* | ja |
| *Stauerzeugungsfequenz:* | *100* |
| *Ausgabefrequenz:* | 100 |
| *Grundfaktor:* | 0.25 |
| *Maximalfaktor:* | 1 |


**Abbildung 19: Nah beieinanderliegende Endpunkte**

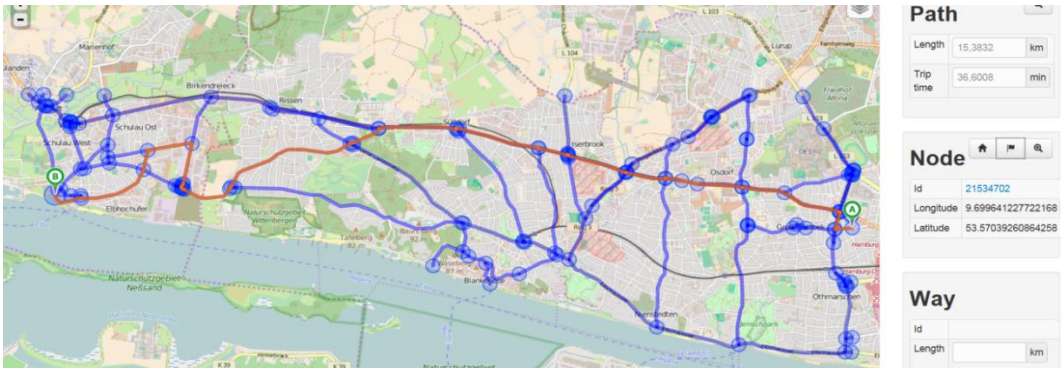


**Abbildung 20: Weit voneinander entfernt liegende Endpunkte**

### 4.9.3 Auswertung

In Abbildung 19 ist zu sehen, dass die Entfernung vom Start- zum Zielpunkt in dem Szenario verhältnismäßig klein ist. Die Anzahl an möglichen Routen vom Start zum Ziel ist aufgrund dessen recht begrenzt. Abbildung 21 zeigt, dass AntNet bei jedem Test zu jeweils einem Zeitpunkt eine Lösung findet, um die es ab dann nur noch sehr geringfügige Fluktuationen gibt. Dieses Verhalten ist vergleichbar mit dem in 4.8 beobachteten. Es ist daher zu vermuten, dass auch die Begründung eine ähnliche ist. Durch die geringe Anzahl an möglichen Routen, erreicht die Staudichte in dem Bereich der optimalen Lösung besonders schnell ein Niveau, bei dem die meisten Kanten keine Option mehr darstellen. Aus diesem Grund nähern sich die Lösungen des Dijkstra-Algorithmus auch immer weiter der durch AntNet gefundenen Lösung an. Wie aber auch bereits bei mehreren vorherigen Fällen, wie z.B. in 4.7 festgestellt, leidet das Ergebnis von AntNet stärker unter einer sehr hohen Staudichte als Dijkstra. Das zeigt sich auch hier.

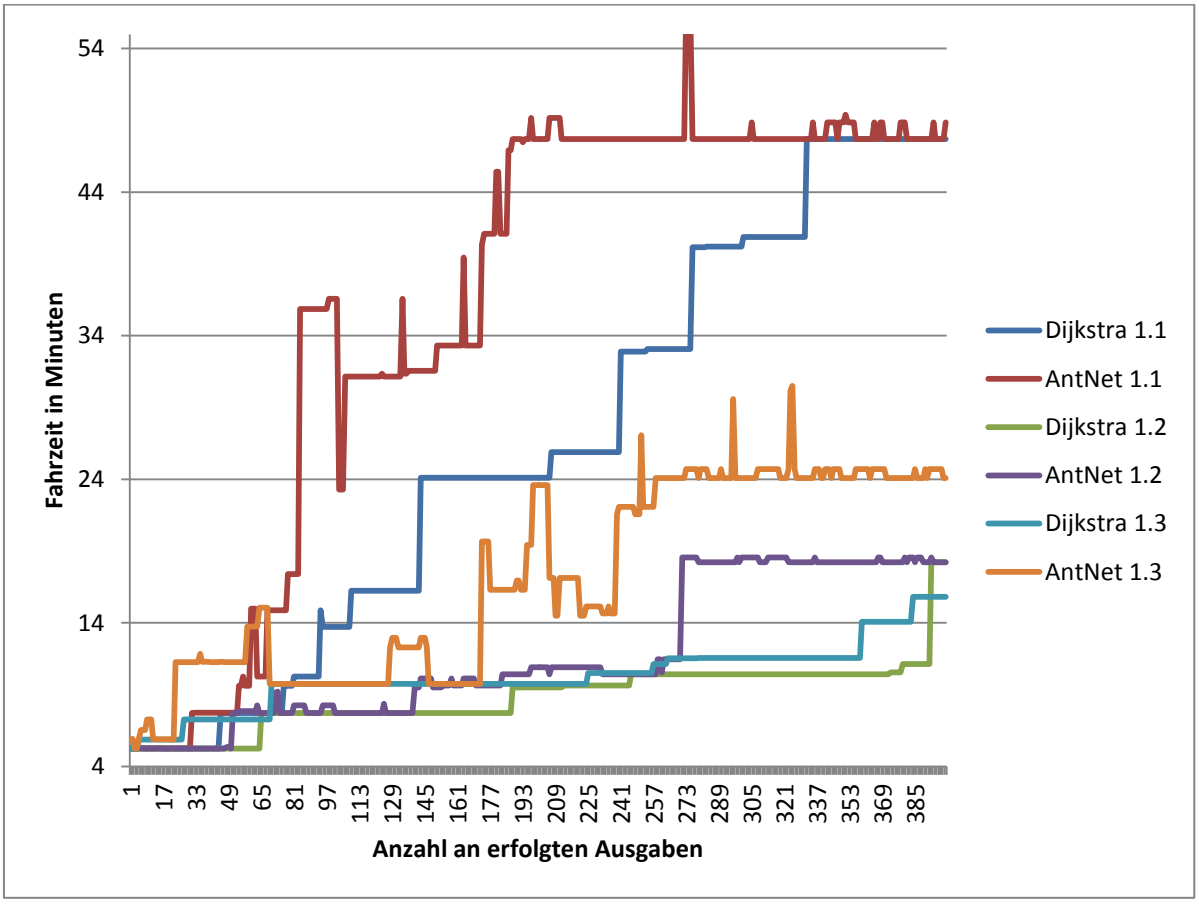


**Abbildung 21: Ausführungen bei nah beieinander liegenden Endpunkten**

Wie genau das AntNet-Verfahren im Vergleich mit dem Dijkstra-Algorithmus abschneidet, ist im Folgenden zusammengefasst.

Test 1.1: Ameisen sind durchschnittlich 48.70 % schlechter als Dijkstra.

Test 1.2: Ameisen sind durchschnittlich 30.07 % schlechter als Dijkstra.

Test 1.3: Ameisen sind durchschnittlich 62.58 % schlechter als Dijkstra.

Das AntNet-Verfahren fällt in diesem Fall prozentual besonders schlecht aus. Grund ist auch dafür die geringe Entfernung der Endpunkte. Da wesentlich weniger Kanten für eine durchschnittliche Lösung befahren werden als bei den Endpunkten in den anderen Fällen, wirkt sich eine einzelne Kante, die durch eine schlechtere Kante ersetzt wird, prozentual wesentlich stärker auf die Gesamtfahrzeit aus.

In Abbildung 20 ist zu sehen, dass die Entfernung der Endpunkte in diesem Szenario wieder größer ist als im eben betrachteten und vergleichbar mit der in dem Szenario, auf dem die anderen Fälle basieren. In Abbildung 22 zeigt sich ein Verhalten der beiden Algorithmen, das zu dem der bisherigen Fälle passt. Dies entspricht der Vermutung, die in 4.9.1 aufgestellt wurde.

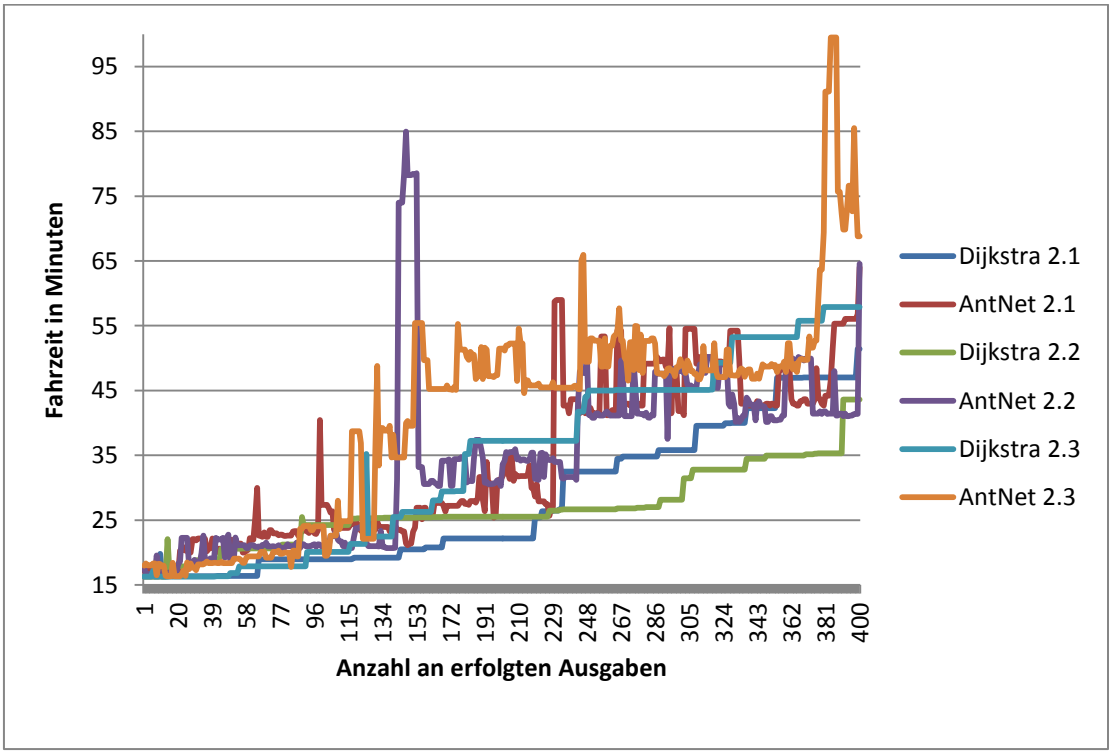


**Abbildung 22: Ausführungen bei weit voneinander entfernt liegenden Endpunkten**

Wie viel schlechter das AntNet-Verfahren in diesem Szenario gegenüber dem Dijkstra-Algorithmus abschneidet, ist im Folgenden zusammengefasst:

Test 2.1: Ameisen sind durchschnittlich 24.53 % schlechter als Dijkstra.

Test 2.2: Ameisen sind durchschnittlich 26.81 % schlechter als Dijkstra.

Test 2.3: Ameisen sind durchschnittlich 20.59 % schlechter als Dijkstra.

Es lässt sich feststellen, dass die Entfernung der Endpunkte, bzw. wie viele Kanten dazwischen liegen, das Ergebnis stark beeinflusst. Ähnlich weit entfernte Endpunkte führen zu vergleichbaren Ergebnissen. Die Annahme aus 4.9.1 ist also teilweise bestätigt. Je geringer die Entfernung der Endpunkte im Verhältnis zur Größe des Gesamtgraphen ist, desto schlechter schlägt sich das AntNet-Verfahren im Vergleich mit dem Dijkstra-Algorithmus.

## 4.10 Laufzeittest

Bei diesem Spezialfall soll die Laufzeit der beiden Algorithmen überprüft werden. Es treten Verkehrsänderungen eingeschränkt zufällig auf. Die Staudichte sowie der Start- und Zielpunkt sind konstant. Alle anderen Parameter variieren.

### 4.10.1 Theoretische Überlegung

Dem AntNet-Verfahren ist in keinem Fall gelungen sich gegenüber dem Dijkstra-Algorithmus durchzusetzen, obwohl einige der Szenarien in der Theorie sogar explizit AntNet bevorzugen. Ein möglicher Grund dafür kann eine sehr kurze Laufzeit des Dijkstra-Algorithmus sein. Findet der eine Lösung äußerst schnell, kommt es selbst bei speziellen Szenarien nur sehr selten vor, dass während der Berechnung ein Stau auftritt oder ein veralteter bester Pfad ausgegeben wird. Dadurch würde Dijkstra fast immer die optimale Lösung liefern und könnte von AntNet gar nicht geschlagen werden. Die theoretische Laufzeit des Dijkstra-Algorithmus wird in 2.2 diskutiert und beträgt $O(n \log n)$, wobei n die Knotenanzahl des Graphen ist. Das AntNet-Verfahren besitzt eine natürliche initiale Laufzeit von $O(z \cdot n^3)$, wobei n die Knotenanzahl des Graphen und z die Anzahl an benötigten Ameisenzyklen ist. AntScout verwendet für die Initialisierung allerdings den Floyd-Warshall-Algorithmus mit einer Laufzeit von $O(n^3)$, was besser als die natürliche Initialisierung, aber schlechter als die Laufzeit des Dijkstra-Algorithmus ist. Dijkstra berechnet den ersten Pfad also schneller als AntNet. Theoretisch ergibt sich danach für AntNet eine Laufzeit von $O(n^2)$, da die Ameisen aber parallel implementiert sind, ist die tatsächliche Laufzeit in Anhängigkeit von der vorliegenden Hardware wesentlich

geringer. Wie in 2.4 erläutert wird und in (Bertram, 2012) zu lesen ist, erzeugt die in AntScout implementierte Variante des AntNet-Verfahrens Overhead, der sich in einer relativ hohen Konstante in der Laufzeitkomplexität des Verfahrens wiederspiegelt. In der Komplexitätstheorie sind Konstanten uninteressant, in der Praxis können sie aber von Relevanz sein.

Die Laufzeit beider Algorithmen in der Praxis wird hier untersucht.

### 4.10.2 Versuchsbeschreibung

Es werden zuerst mehrere Ausführungen mit dem Dijkstra-Algorithmus durchgeführt, wobei bei jeder Ausführung neue Verkehrsänderungen zufällig erzeugt werden. Die Stauerzeugung besitzt bei jeder Ausführung andere Parameterbelegungen. Als Endpunkte dienen die in Abbildung 8 dargestellten. Danach werden mehrere Ausführungen mit dem AntNet-Verfahren durchgeführt, wobei dieselben Verkehrsänderungen und Endpunkte verwendet werden wie zuvor beim Dijkstra-Algorithmus. Es wird die Laufzeit der Algorithmen anhand der Systemzeit gemessen.

| | |
|---|---|
| *Positive Verkehrsänderungen erlauben:* | variabel |
| *Ungelenkter Zufall:* | variabel |
| *Stauerzeugungsfequenz:* | 100 |
| *Ausgabefrequenz:* | 100 |
| *Grundfaktor:* | variabel |
| *Maximalfaktor:* | variabel |

### 4.10.3 Auswertung

Die festgestellte Laufzeit des Dijkstra-Algorithmus liegt fast immer im Bereich von 20 bis 30 Millisekunden. Das ist verhältnismäßig schnell. Tests mit Stauerzeugungsfrequenzen, die noch unterhalb dieses Bereiches liegen, um eine Stauerzeugung während der Rechenzeit zu erzwingen, sind weit entfernt von realistischen Szenarien. Grund für die sehr niedrige Laufzeit ist die eingeschränkte Größe des Verkehrsnetzes. Die oben beschriebene große Konstante in der Laufzeit von AntNet spielt bei der verhältnismäßig geringen Anzahl an Knoten eine größere Rolle als der variable Teil. In (Bertram, 2012) wurde experimentell festgestellt, dass die Ameisen in AntScout teilweise mehrere Sekunden benötigen, um sich an einzelne neue Verkehrsänderungen anzupassen. Die Anpassungszeit ist also gravierend schlechter als beim Dijkstra-Algorithmus. Es ist möglich, dass mit steigender Größe des Graphen der beobachtete Vorteil des Dijkstra-Algorithmus gegenüber dem AntNet-Verfahren sinkt und sich irgendwann sogar in einen Nachteil wandelt, da die Konstante bei größeren Problemstellungen an Bedeutung verliert. Die sehr geringe Laufzeit des Dijkstra-Algorithmus und die sehr deutlichen Unterschiede bei der Lösungsgüte der beiden Verfahren, die sich in den verschiedenen vorherigen Fällen gezeigt haben, legen die Vermutung nahe, dass das Verkehrsnetz erheblich vergrößert werden muss, um AntNet wirklich bessere Ergebnisse liefern zu lassen als Dijkstra. Die vorliegende Implementierung des AntNet-Verfahrens kann aber nicht mit so großen Netzen getestet werden, daher ist es nicht möglich, diese Vermutung zu überprüfen.

# 5 Abschlussbetrachtung

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und ein Fazit gezogen. Im Anschluss wird ein Ausblick auf mögliche zukünftige Entwicklungen und weiterführende Untersuchungen gegeben.

## 5.1 Zusammenfassung

In der Einführung wurde die aktuelle Situation im Bereich Verkehrsnavigation erläutert und daran begründet, warum ein Vergleich des Dijkstra-Algorithmus mit einem Ameisensystem bezüglich ihrer Einsatzmöglichkeit bei dynamischen Routingproblemen sinnvoll ist. Dieser Vergleich wurde als zentrales Thema dieser Arbeit vorgestellt.

In den Grundlagen wurde die real vorhandene Verkehrsproblematik abstrakt anhand des Shortes Path Problem dargestellt und die beiden zu vergleichenden Verfahren Dijkstra-Algorithmus und Ameisensystem ausführlich erklärt. Beim Ameisensystem wurde AntNet als Spezialversion besonders herausgearbeitet und gezeigt, dass diese bereits in dem im Rahmen einer vorherigen Masterarbeit entstandenen Programms AntScout Verwendung gefunden hat. Dieses Programm wiederrum diente als Ausgangsbasis für alle Konzeptionen und Implementierungen, die im Rahmen dieser Arbeit entstanden sind.

Im nachfolgenden Kapitel wurde ausführlich dargestellt, wie AntScout aufgebaut ist. Insbesondere wurde dargelegt, was OpenStreetMap ist und wie es von AntScout verwendet wird. Alle Klassen wurden vorgestellt und dabei unterschieden, welche bereits vorher Teil von AntScout waren, um AntNet auf OpenStreetMap umzusetzen, und welche im Rahmen dieser Arbeit neu erstellt worden sind. Dabei wurde vor allem das neu entwickelte alternative Graphenkonzept erklärt, das für die Nutzung durch den Dijkstra-Algorithmus optimiert wurde und auch die Implementierung dieses Algorithmus enthält. Außerdem wurde die Aufgabe und Funktionsweise des neu erstellten Staugenerators JamGen dargelegt, der durch das automatische Erzeugen von Verkehrsänderungen die späteren Vergleichstests ermöglichte. Es wurde gezeigt, dass die überarbeitete Version von AntScout mit verschiedenen Dateitypen für die Kommunikation und Persistenz arbeitet. Diese Dateitypen wurden anhand von einzelnen Beispielen erklärt. Zum Schluss wurden anhand der Konfigurationsdatei die Parameter erklärt, die für die Vergleichstests wichtig waren.

Im Vergleichskapitel wurden die eigentlichen Erkenntnisse gewonnen. Es stellt den Hauptteil dieser Arbeit dar. Nach dem Beschreiben der Testumgebung, wurde in mehreren unterschiedlichen Testszenarien anhand von diversen Aspekten der Dijkstra-Algorithmus mit dem AntNet-Verfahren verglichen, wobei Dijkstra sich jedes

Mal durchsetzen konnte. Durch immer gezieltere, auf realen Verkehrsproblemen basierenden und aufeinander aufbauenden Testfällen, wurde das Verhalten der beiden Algorithmen immer detaillierter untersucht. Als Ergebnis ergab sich, dass die vorliegende Implementierung von AntNet bei der bei AntScout maximal möglichen Szenariogröße die im Rahmen dieser Arbeit entwickelte Version des Dijkstra-Algorithmus durchschnittlich nicht schlagen konnte.

## 5.2 Fazit

In der Einleitung wurde als Ziel dieser Arbeit der theoretische und praktische Vergleich des Dijkstra-Algorithmus mit einem Ameisensystem vorgestellt. Die beiden Verfahren mit AntNet als Beispiel eines Ameisensystems wurden anhand der Testfälle in 4 ausführlich untersucht. Die Tests haben gezeigt, dass sich AntNet im Rahmen der vorliegenden Szenariogröße nicht gegenüber dem Dijkstra-Algorithmus durchsetzen kann. Besonders schlechte Ergebnisse liefert AntNet, wenn die Entfernung von Start- und Zielpunkt besonders klein, die Ausführzeit besonders lang oder die Frequenz, mit der Staus erzeugt wurden, besonders hoch war. Alle drei Umstände resultieren unter anderem in dem gleichen Ergebnis, nämlich einer stark erhöhten Staudichte zwischen Start und Ziel. Der Dijkstra-Algorithmus bestimmt die optimale Lösung für eine konkrete Verkehrssituation unabhängig von vorherigen Berechnungen. Daher liegt immer, wenn bei Berechnungsstart des Algorithmus die optimale Route die gleiche ist wie bei Berechnungsende, danach die tatsächlich optimale Route garantiert vor. Wie in 4.10.3 erläutert, beträgt die Laufzeit für die verwendete Szenariogröße nur 20 bis 30 Millisekunden, womit sie weit unterhalb realistischer Stauerzeugungsfrequenzen liegt. Diese kurzen Zeitintervalle sind also die einzigen Momente, während denen der Dijkstra-Algorithmus nicht garantiert die optimale Route ausgibt. AntNet als heuristisches Verfahren liefert grundsätzlich nur Näherungswerte, die den direkten Vergleich mit dem tatsächlich optimalen Ergebnis durchschnittlich nur verlieren können. Anders als beim Dijkstra-Algorithmus wirkt sich bei dem AntNet-Verfahren jede Verkehrsänderung auch auf zukünftige Berechnungen aus. Eine steigende Staudichte erhöht daher die Zeit, die AntNet zum Anpassen benötigt. Da das Verfahren dafür bereits bei einem einzigen Stau mehrere Sekunden benötigen kann, schlägt sich der Dijkstra-Algorithmus bei hoher Staudichte sehr viel besser. Dies ließ sich auch experimentell bestätigen.

In Abschnitt 2.3 wurde die Stärke von Ameisenalgorithmen und heuristischen Verfahren im Allgemeinen beim Bearbeiten von NP-vollständigen Problemen hervorgehoben. Das dynamische Routingproblem in seiner hier auftretenden Ausprägung gehört allerdings nicht zu dieser Problemklasse. Ein Grund dafür ist die

quasi konstante Anzahl an Kanten pro Knoten. Die sehr effiziente Lösbarkeit durch den Dijkstra-Algorithmus, die im Rahmen der Tests gezeigt wurde, beweist dies auch. Die besondere Stärke von heuristischen Verfahren kommt in diesem Fall daher gar nicht zur Anwendung.

Die Laufzeit des Dijkstra-Algorithmus steigt bei einem Anwachsen der Szenariogröße mehr als linear an. Wenn die Laufzeit dadurch übliche Stauerzeugungsfrequenzen weit übersteigt, reduziert sich der Zeitraum, in dem der Algorithmus das tatsächlich optimale Ergebnis liefert und die durchschnittliche Lösungsgüte sinkt. Die Anpassungszeit der Ameisen steigt theoretisch langsamer an. Die vorliegende Implementierung von AntNet ist aber aufgrund von sehr schnell wachsendem Ressourcenverbrauch im Gegensatz zum Dijkstra-Algorithmus nicht geeignet, in der vorliegenden Testumgebung in größeren Szenarios eingesetzt zu werden.

Das abschließende Fazit lautet, dass für das Lösen von realen dynamischen Routingproblemen bei einer Wahl zwischen der im Rahmen dieser Arbeit entwickelte Dijkstra-Algorithmus und dem in AntScout verwendeten AntNet-Verfahren, immer der Dijkstra-Algorithmus gewählt werden sollte.

## 5.3 Ausblick

Dieser Abschnitt soll einige mögliche Projekte und Forschungsarbeiten beschreiben, die sich aus dieser Arbeit ergeben können. Die Reihenfolge der vorgestellten Ideen ist beliebig und stellt keinerlei Rangliste dar.

- Aktorensysteme erlauben eine verteilte Ausführung auf mehreren Rechnern, da die Kommunikation der einzelnen Klassen primär über Nachrichten erfolgt. Durch die Verteilung von AntScout auf mehrere Rechner, kann dem schnell steigenden Ressourcenverbrauch der Ameisen entgegengewirkt werden. Dadurch könnte es möglich werden, auch wesentlich größere Szenarien mit Hilfe von AntNet bearbeiten zu können. Bei einem entsprechend großen Szenario und ausreichend Rechnern könnte AntNet einen Vorteil gegenüber Dijkstra erlangen, da die Laufzeit von letzterem steigt und der Algorithmus nicht parallel umgesetzt werden kann. Dies müsste aber entsprechend untersucht werden.

- Moderne Navigationsgeräte verwenden teilweise Varianten des Dijkstra-Algorithmus, die effizienter sind als die hier verwendete. Eine Möglichkeit ist z.B. von Start- und Endpunkt gleichzeitig den besten Weg zum jeweils anderen zu berechnen. Auf diese Weise kann der Algorithmus verteilt und der Suchraum in kürzerer Zeit erschlossen werden. Durch Verwendung einer

solchen noch effizienteren Version des Dijkstra-Algorithmus können noch größere Szenarios damit bearbeitet werden, ohne dass die Laufzeit eine realistische Stauerzeugungsfrequenz übersteigt.

- Der im Rahmen dieser Arbeit implementierte Dijkstra-Algorithmus arbeitet auf einem speziellen, sehr effizienten Graphen. Würde eine Möglichkeit gefunden werden, auch AntNet auf diesem Graphen operieren zu lassen, würde ein Teil des Overheads wegfallen und die Ameisen könnten sich schneller an neue Staus anpassen. In wie weit dies sinnvoll möglich ist und in wie weit dies mit der vorher diskutierten Idee mit einem verteilen AntNet-Verfahren kombinierbar ist, müsste untersucht werden.

- Die aktuelle Java-Version Java 8 besitzt im Gegensatz zum Vorgänger Java 7, der bei Entwicklung der ursprünglichen Version von AntScout aktuell war, Unterstützung für das Aktorensystem Akka. Daher könnte AntScout aktuell auch mit Java umgesetzt werden, ohne auf die Unterstützung von Akka verzichten zu müssen. Sollte der mit andauernder Ausführzeit steigende Ressourcenverbrauch eine Folge der Implementierung in Scala sein, würde eine Implementierung in Java dieses Problem lösen.

- Die in Kapitel 4 erstellten Diagramme stellen nur einen Teil der Möglichkeiten dar, die bei den durchgeführten Tests erzeugten Rohdaten zu verarbeiten. Es ist möglich, noch weitergehende Vergleiche anzustellen, um Aspekte zu beleuchten, die im Rahmen dieser Arbeit vernachlässigt wurden. Die Rohdaten finden sich auf der beiliegenden CD.

# Literaturverzeichnis

Bertram, A. (2012). AntScout - Dynamisches Routing auf OpenStreetMap. Wedel.

Diestel, R. (2010). *Graphentheorie.* Berlin: Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik. 1*, S. 269-271.

Dorigo, M., & Gambardella, L. M. (1997). Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *Transactions on Evolutionary Computation*, S. 53-66.

Dorigo, M., & Stützle, T. (2004). *Ant Colony Optimization.* MIT.

Joksch, H. (1966). The shortest route problem with constraints. *Journal of Mathematical Analysis and Applications*, S. 191-197.

# Abbildungsverzeichnis

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Pinneberg, den 15. März 2016 Timo Jürgens