Universität Rostock

Traditio et Innovatio

Flotten-Management und dynamische kürzeste Pfade

Name, Vorname:	Al-Nsser, Hadeel
Studiengang:	Informatik
Eingereicht an der:	Fakultät für Informatik und Elektrotechnik
Erstgutachter:	Dr.-Ing. Holger Meyer
Zweitgutachter:	Dr.-Ing. Dennis Marten
Abgabedatum:	28.02.2023

Bachelorarbeit

Zusammenfassung

In einem gerichteten und gewichteten Graphen wird das Single-Pair Shortest Path Problem im Kontext der maritimen Navigation unter Berücksichtigung dynamischer Aspekte untersucht. Dabei können verschiedene Faktoren betrachtet werden, die zu dynamischen Veränderungen im Graphen führen können. Insbesondere wird der Fokus auf Unwetterereignissen gelegt, da diese ein häufiges und schwerwiegendes Ereignis im maritimen Kontext darstellen. Zur Lösung des Problems werden vier verschiedene Ansätze vorgeschlagen und diskutiert. Diese umfassen die Suche nach einem alternativen Pfad, die Manipulation der Geschwindigkeit des Schiffs, die Anpassung der Abfahrtszeit sowie das Warten an einem oder mehreren Knoten. Jeder dieser Ansätze wird im Hinblick auf seine Anwendbarkeit analysiert und im Falle der Umsetzbarkeit werden dafür konkrete Algorithmen vorgestellt und getestet. Im Anschluss werden die resultierenden Algorithmen verglichen. Hierbei werden spezifische Bewertungskriterien wie die Länge der ermittelten kürzesten Pfade und die Wartezeit herangezogen. Ziel ist es nicht, den effizientesten Algorithmus zu finden, sondern denjenigen, der den kürzesten Weg findet. Die Ergebnisse dieser Arbeit leisten einen bedeutenden Beitrag zur Anwendung von Algorithmen im maritimen Kontext und bieten konkrete Handlungsempfehlungen für die Praxis. Insbesondere können die untersuchten Ansätze und Algorithmen dazu beitragen, die Leistungsfähigkeit und Sicherheit des Seeverkehrs zu erhöhen und damit langfristig sowohl ökonomische als auch ökologische Vorteile zu generieren.

Abstract

In a directed and weighted graph, the single-pair shortest path problem is investigated in the context of maritime navigation, taking dynamic aspects into account. Various factors that can lead to dynamic changes in the graph can be considered. In particular, the focus is on severe weather events, as these are a frequent and severe event in the maritime context. Four different approaches are proposed and discussed to solve the problem. These include finding an alternative path, manipulating the speed of the vessel, adjusting the departure time and waiting at one or more nodes. Each of these approaches is analysed in terms of its applicability and, in case of feasibility, concrete algorithms are presented and tested for it. Subsequently, the resulting algorithms are compared. Specific evaluation criteria such as the length of the determined shortest paths and the waiting time are used. The goal is not to find the most efficient algorithm, but the one that finds the shortest path. The results of this work make a significant contribution to the application of algorithms in the maritime context and offer concrete recommendations for action in practice. In particular, the approaches and algorithms investigated can contribute to increasing the efficiency and safety of maritime transport and thus generate both economic and ecological benefits in the long term.

Inhaltsverzeichnis

1 Einleitung

Hunderttausende von Schiffen mit kommerziellen, industriellen, touristischen und anderen Hintergründen fahren jeden Tag. Einige davon fahren kurze Strecken, andere lange oder sogar extrem lange Strecken, die sich über Tage hinziehen. Wenn ein Schiff auf dem weiten Meer unterwegs ist, fährt es mit dem Ziel, an einem bestimmten Punkt oder Hafen anzukommen. Meistens verlaufen jedoch mehrere Routen in Richtung des Ziels, aber woher soll das Schiff wissen, welchen Weg es nehmen soll? Außerdem fährt nicht nur ein Schiff, sondern mehrere. Es kann auch vorkommen, dass mehrere Schiffe das gleiche Ziel ansteuern. Wie kann sichergestellt werden, dass das Schiff sicher ankommt, ohne den Weg mit anderen Schiffen zu kreuzen und mit ihnen zusammenzustoßen?
Dies ist einer der Anwendungsfälle, in denen eine Entscheidung für die Suche nach einem Weg erforderlich ist, jedoch nicht irgendeinem Weg, sondern dem kürzesten Weg. Denn der kürzeste Weg bedeutet weniger Zeit auf See und eine kürzere Fahrtstrecke, was zu schnelleren Lieferzeiten bei Frachtschiffen führt. Ebenso bedeutet er geringere Kosten, da weniger Treibstoff verbraucht wird. Allerdings ist es nicht immer einfach, den kürzesten Weg zu finden, und kann manchmal problematisch sein. Dieses Problem des kürzesten Pfades (Abschnitt 2.2) besteht schon seit langem, wobei versucht wird, diesen schnellsten effizienten Weg zu ermitteln. Hierzu müssen alle notwendigen Informationen vorliegen, wie z. B. die Anzahl der Häfen und die Position der einzelnen Häfen. Je mehr Auswahlmöglichkeiten bzw. Wege es gibt, desto schwieriger ist es, den Richtigen zu finden. Tatsächlich wurden bereits zahlreiche Algorithmen zur Berechnung und Bestimmung des kürzesten Weges entwickelt. Diese Algorithmen wurden erfolgreich eingesetzt, bis sie schließlich nicht mehr ausgereicht haben. Das liegt daran, dass unerwartete Ereignisse diese Informationen jederzeit ändern können, z.B. ein Brand am Hafen oder ein Sturm an einem Kanal im Meer führt zur vorübergehenden Sperrung des Hafens/Kanals. Solche Änderungen müssen dann unbedingt im Blick behalten werden. Da sie sich auf den gefundenen Weg auswirken können, und der Weg ist dann nicht mehr befahrbar und muss entsprechend angepasst oder erneut gesucht werden. Um diese Problematik besser zu begreifen, wird sie zunächst an einem kleinen, vereinfachten Beispiel geschildert.

1.1 Allgemeines Beispiel

Ein Schiff möchte von dem Hafen (A) zu dem Hafen (F) fahren (Siehe Abbildung 1). Auf dem Meer zwischen den beiden Häfen (A) und (F) befinden sich mehrere festgelegte Orte, die das Schiff anfahren und an denen es sich orientieren kann (diese sind B, C, D, E, G, H, I, K und M). Hierbei muss es sich nicht zwangsläufig um Häfen handeln, sondern es können auch Wegpunkte/ Koordinationspunkte sein.
Diese Häfen/Punkte sind miteinander verbunden, und es sind die Zeiten berechnet, die das Schiff zwischen jeweils zwei verbundenen Punkten benötigt, um von einem zum anderen zu gelangen, z. B. das Schiff benötigt für die Fahrt von (A) nach (M) 2 Stunden. Betrachtet man diese Punkte als Knoten und die Verbindungen als Kanten mit Gewichten, die der Zeit entsprechen, kann das Ganze als Graph (Abschnitt 2.1) angesehen werden.

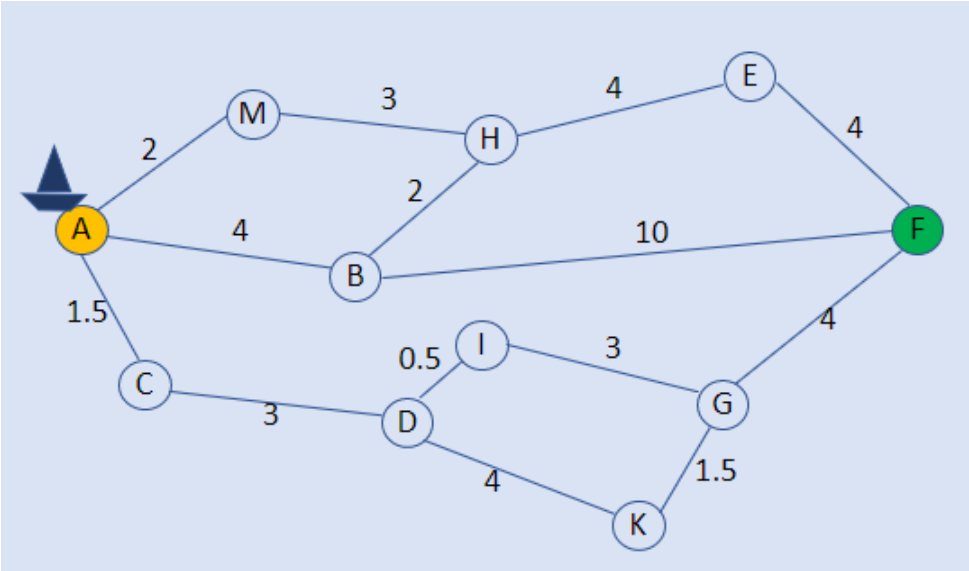


Abbildung 1: Allgemeines Beispiel

Aus den Knoten und Kanten des Graphs entstehen verschiedene Wege von (A) nach (F). Beispielsweise die Wege (A → B → F), (A → C → D → K → G → F) oder (A → M → H → E → F) (Siehe Abbildung 2). Aber welchen Weg soll das Schiff nehmen? Welcher Weg ist der kürzeste?

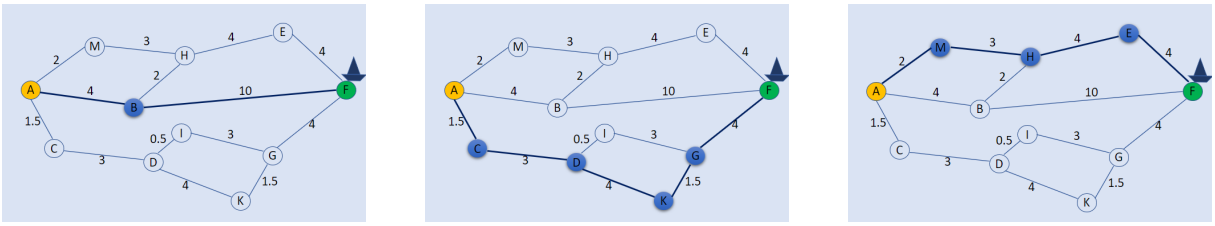


Abbildung 2: Mögliche Wege von (A) nach (F)

Jeder Weg hat seine eigenen Gesamtkosten, die sich aus der Summe der Kantengewichte ergibt, die den Weg bilden. Nun sollen alle möglichen Wege von (A) nach (F) auf ihre Gesamtkosten hin untersucht werden, um den kürzesten Weg zu finden und ihm folgen zu können, wobei der kürzeste Weg derjenige ist, der die geringsten Gesamtkosten aufweist.
Nachstehend werden die Gesamtkosten für alle möglichen Wege berechnet:

1. A → B → F: Gesamtkosten = Gewicht von (A, B) + Gewicht von (B, F) = 4 + 10 = 14
2. A → B → H → E → F: Gesamtkosten = 4 + 2 + 4 + 4 = 14
3. A → M → H → E → F: Gesamtkosten = 2 + 3 + 4 + 4 = 13
4. A → M → H → B → F: Gesamtkosten = 2 + 3 + 2 + 10 = 17
5. A → C → D → I → G → F: Gesamtkosten = 1.5 + 3 + 0.5 + 3 + 4 = 12
6. A → C → D → K → G → F: Gesamtkosten = 1.5 + 3 + 4 + 1.5 + 4 = 14

Der kürzeste Weg ist somit der Weg (A → C → D → I → G → F) mit einen Gesamtkosten von 12 Stunden (Abbildung 3). Diese Berechnung muss heutzutage nicht mehr auf diese Art und Weise durchgeführt werden, es existieren verschiedene Algorithmen, die dies berechnen und schnell den kürzesten Weg liefern. Mehr dazu in den Abschnitten 2.3 und 5.

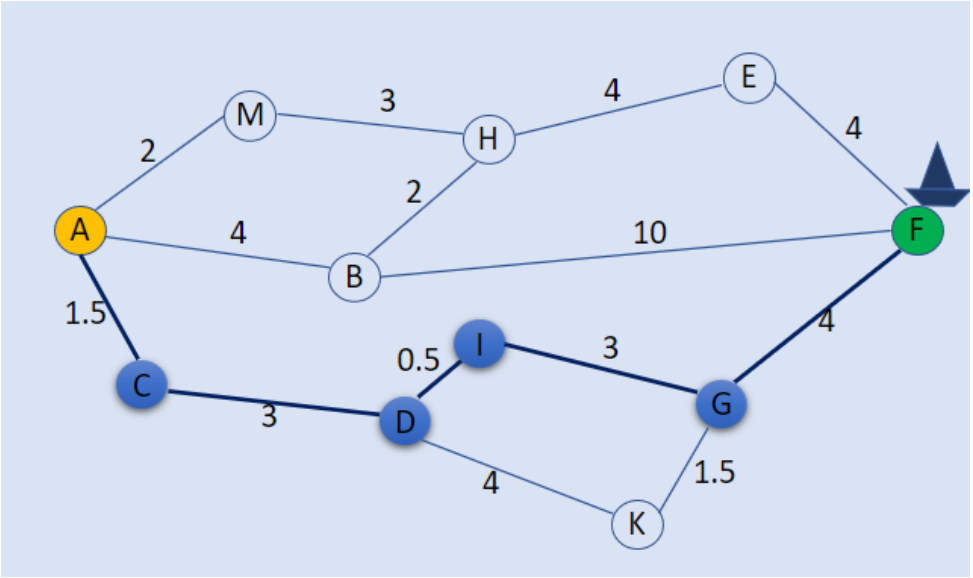


Abbildung 3: Allgemeines Beispiel - Der kürzeste Weg

Nun ist der bestmögliche Weg ermittelt und das Schiff weiß, welcher Spur es folgen soll. Dieser Weg ist jedoch nur so lange befahrbar, soweit sich der Graph nicht verändert. Der Grund dafür ist, dass der Graph als statisch und konstant betrachtet wurde, das heißt, es sind keine Änderungen vorzusehen, weder an den Knoten noch an den Kanten. Was aber, wenn unerwartete Hindernisse auf der Strecke auftauchen und der Weg aufgrund gewisser Umstände nicht mehr befahrbar ist? Wie soll sich das Schiff dann verhalten?

Das größte Hindernis auf See ist das Wetter. Ein Sturm wurde angekündigt, dieser Sturm betrifft die Knoten D, I und G sowie die dazwischen liegenden Kanten, wodurch der geplante Weg nicht mehr befahrbar ist. Das Schiff kann nicht einfach in den Sturm hineinfahren.

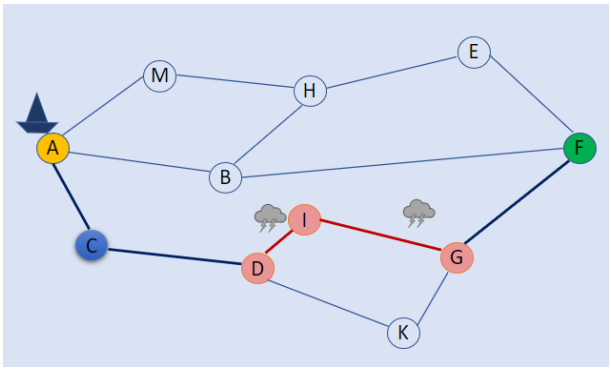


Abbildung 4: Weg - Sturm

Ein solcher Sturm hält sich nicht fest in einem Gebiet, sondern bewegt sich entlang der Windrichtung, und es kann nie genau sichergestellt werden, dass ein Sturm in 2 Stunden abflaut. Daher hat der Sturm den Graphen dynamisch modifiziert (Abschnitt 2.1.1), indem er vorübergehend Knoten und unpassierbare Kanten entfernt hat, was wiederum die Gesamtkosten verändert, da sich die Gewichte ändern.
Deswegen muss zügig eine Lösung und gegebenenfalls ein neuer Weg gefunden werden, was das Ziel der vorliegenden Arbeit ist, das im nächsten Abschnitt bekräftigt wird.

1.2 Ziel

In dieser Arbeit wird das Problem des kürzesten Pfades in einem maritimen Kontext unter dynamischen kritischen Effekten auf einem umfangreichen Graphen mit viel mehr Häfen als in dem kleinen Beispiel im vorherigen Abschnitt untersucht und analysiert. Dieser Graph wird nicht als statischer, konstanter Graph behandelt, sondern als dynamischer Graph, der durch verschiedene unerwartete oder unvorhersehbare Faktoren wie schlechtes Wetter, Meeresströmungen, Hafenüberlastung oder andere Schiffe und den Seeverkehr beeinflusst werden kann.
Es werden Möglichkeiten bzw. Lösungskonzepte vorgestellt, wie auf solche dynamischen Faktoren reagiert werden kann und wie sich jedes Schiff verhalten sollte. Solche Lösungen sollten so effizient wie möglich gestaltet werden, da sie für eine Vielzahl von Schiffen erarbeitet werden müssen und darüber hinaus bei jeder dynamischen Veränderung erneut durchgeführt werden müssen. Dies ist ziemlich aufwändig und kostenintensiv. Überdies hängt diese Lösung von vielen Faktoren ab, wie der Größe des Schiffes, dem Schiffstyp oder auch der Hafenkapazität.
Das Hauptanliegen bleibt jedoch, den kürzesten, schnellsten und effizientesten Weg zu finden. Genauer ausgedrückt, es soll ein Weg von einem Start- zu einem Zielhafen gesucht werden, d.h. es wird der Fall des Single-Pair Shortest Path Problems behandelt (Abschnitt 2.2). Ferner wird von allen genannten Faktoren nur das Unwetter berücksichtigt, aber die sich daraus ergebende mögliche Lösung sollte auch auf die anderen Faktoren anwendbar sein. Entscheidend ist nicht nur, welcher Weg kürzer ist, sondern auch, welcher Weg die Sicherheit des Schiffes und der Ladung sowie der Crew und der Passagiere gewährleistet und garantiert.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in mehreren Kapiteln. Zu Beginn werden im Kapitel 2 “Grundlagen“ die theoretischen Grundlagen und Konzepte vermittelt, die für das Verständnis der nachfolgenden Kapiteln erforderlich sind. Danach wird im Kapitel 3 “Flottenmanagement“ die Bedeutung von Flottenmanagement im maritimen Kontext untersucht, sowie die Herausforderungen, die bei der Verwaltung einer Flotte von Schiffen auftreten können. Das darauf folgende Kapitel 4 “Wetterverhältnisse“ beschäftigt sich mit der Rolle des Wetters im Zusammenhang mit dem Seeverkehr und den Auswirkungen von Wettervorhersagen auf die Sicherheit und Effizienz von Schiffen. In einem weiteren Kapitel 5 “Stand der Technik“ werden die neuesten Entwicklungen in Bezug auf Technologien und Methoden im Bereich des dynamischen- / kürzeste Pfades beleuchtet. Daraufhin werden die zur Umsetzung des Konzepts verwendeten Daten im Kapitel 6 “Vorstellung der eingesetzten Daten“ näher aufgezeigt. Daran schließt sich das Kapitel 7 “Konzept“, darin werden verschiedene denkbare Lösungsansätze für das Problem des kürzesten Weges unter dynamischen Aspekten vorgestellt und im Kapitel 8 “Auswertung und Evaluation“ verglichen. Anschließend, im Kapitel 9 “Fazit und Ausblick“ werden die wichtigsten Ergebnisse und Erkenntnisse dieser Arbeit zusammengefasst. Die Implikationen der Ergebnisse werden diskutiert und mögliche zukünftige Forschungsrichtungen werden vorgeschlagen.

2 Grundlagen

Für ein tieferes Verständnis des Themas und der Problematik werden in diesem Kapitel die wesentlichen Grundlagen und Begriffe genau abgegrenzt und erklärt. Im Abschnitt 2.1 wird ein Grundverständnis für den Graphen vermittelt. Es wird verdeutlicht, was ein Graph ist, welche Typen und Arten unterschieden werden und wie der Graph eingesetzt und visualisiert werden kann. Danach wird das Problem des kürzesten Pfades (Abschnitt 2.2) näher vorgestellt, sowie einer der berühmtesten Algorithmen, der Dijkstra-Algorithmus (Abschnitt 2.3), der eine Lösung des Problems liefert, dessen Vorgehensweise anhand eines Beispiels gezeigt wird. Daraufhin wird die Reisezeit zwischen zwei Punkten auf der Erde diskutiert (Abschnitt 2.4), die zur Berechnung der Kantengewichte notwendig sein könnte.

2.1 Graph

Ein Graph [17] ist eine Struktur, die aus einer Menge von Objekten (Knoten und Kanten) besteht, wobei Kanten die Knoten miteinander verbinden und diese Knoten als Endpunkte der jeweiligen Kanten bezeichnet werden. Häufig werden Knoten durch Punkte und Kanten durch Linien dargestellt. Mathematisch ausgedrückt, ist ein Graph G = (V, E) ein endliches geordnetes Paar, wobei V = $\{v_1, v_2, \ldots, v_n\}$ die Menge der Knoten und E $\subseteq \{(v_1, v_2) : (v_1, v_2) \in V^2, v_1 \neq v_2\}$ die Menge der Kanten sind. Diese Kanten können Daten bereitstellen, z. B. die Anzahl der verbundenen Knoten und die Richtung der Kante. Durch diese Daten wurden die Graphen in mehrere Typen unterteilt:

- *ungerichteter Graph*, die Kanten haben keine Richtung. Jede Kante kann in beide Richtungen durchlaufen werden.
- *gerichteter Graph*, die Kanten werden durch Pfeile vom Anfangs- zum Endknoten dargestellt. Dies verdeutlicht, dass jede Kante des Graphen nur in eine Richtung durchlaufen werden kann.
- *gewichteter Graph*, jeder Kante wird eine Zahl (das Gewicht) zugeordnet, die je nach Problemstellung z.B. Kosten, Längen oder Kapazitäten darstellen kann.
- *Hypergraph*, eine Kante verbindet nicht nur zwei, sondern mehrere Knoten gleichzeitig.

Daneben gibt es noch andere Typen, wie z. B. Multigraph und Baum. Bei jedem dieser Typen kann noch zwischen statisch und dynamisch unterschieden werden. Was der Unterschied zwischen den beiden ist, wird im nächsten Abschnitt erklärt.

2.1.1 Statische vs. Dynamische Graphen

Ein Graph kann entweder statisch sein, d.h. die Graph-Entitäten (Knoten, Kanten, Gewichte) bleiben über die Zeit konstant, oder dynamisch, d.h. einige dieser Entitäten ändern sich im Laufe der Zeit [4]. Typische Änderungen an einem Graphen sind die Steigerung bzw. die Verringerung des Kantengewichts und das Einfügen bzw. das Entfernen von Kanten/Knoten. Die letzten beiden Arten von Änderungen modellieren topologische Veränderungen im Graphennetz. Diese dynamischen Veränderungen stellen eine besondere Herausforderung dar, denn es

ist unerlässlich, effizient auf diese Veränderungen zu reagieren und auf jede Anfrage schnell zu antworten. Warum sind Graphen wichtig, obwohl sie auch problematisch sind? Diese Frage wird nachstehend beantwortet.

2.1.2 Einsatz und Darstellung von Graphen

Graphen erlauben es, viele Arten von Beziehungen und Prozessen in physikalischen, biologischen, sozialen und Informationssystemen zu modellieren. Viele praktische Probleme können durch Graphen dargestellt und analysiert werden, z. B. die Untersuchung von Atomen in der Chemie oder die Simulation einer möglichen Ausbreitung von Infektionskrankheiten.
Um einen Graph darzustellen und abzubilden, stehen zahlreiche Optionen zur Verfügung, darunter das NetworkX. NetworkX [11] ist ein Python-Paket zur Erstellung, Bearbeitung und Untersuchung der Struktur, Dynamik und Funktionen komplexer Netzwerke. Es eignet sich für den Einsatz in großen realen Graphen. Zur Visualisierung des Netzes ist Matplotlib [24], eine Python-Bibliothek, eine der besten Varianten. Die Abbildung 5 illustriert ein Beispiel für einen kleinen gerichteten, gewichteten Graphen, der mit NetwerkX erstellt und mit Matplotlib gezeichnet wurde. Die roten Kreise sind die Knoten, die Pfeile sind die gerichteten Kanten und die Nummern darauf sind die Gewichte.

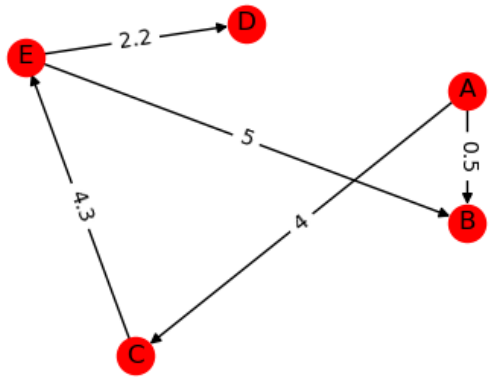


Abbildung 5: Gerichteter-Graph Beispiel

2.2 Problem des kürzesten Pfades

Nachdem das Prinzip der Graphen deutlich geworden ist, kann das Problem des kürzesten Pfades besser beschrieben werden. Das Problem liegt darin, einen Pfad zwischen zwei vorgegebenen Knoten in einem Graphen zu finden, bei dem die Summe der Gewichte der ihn bildenden Kanten minimal ist. Es wird auch als das *Single-Pair Shortest Path Problem* genannt, um es gegenüber von folgenden Varianten abzugrenzen [20] :

- Das *Single-Source Shortest Path Problem*: In einem Graphen soll der kürzeste Pfad von einem gegebenen Anfangsknoten zu jedem Knoten gefunden werden.
- Das *Single-Destination Shortest Path Problem*: Der kürzeste Pfad zu einem gegebenen Zielknoten von jedem Knoten sollte gefunden werden.

- Das *All-Pairs Shortest Path Problem*: Kürzeste Pfade zwischen jedem Paar von Knoten im Graphen sollten gefunden werden.

Für das Problem des Kürzesten Pfades liegen bereits mehrere Lösungen vor. Eine davon bietet der Dijkstra-Algorithmus. Dieser wird im folgenden Abschnitt erörtert.

2.3 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus [15], mit der Zeitkomplexität $O(n^2)$, durchläuft alle Knoten v im Graphen mit dem Ziel, den kürzesten Pfad zwischen einem StartKnoten s und einem Zielknoten z zu finden. Jeder v wird einen Abstand d[v] von Unendlichkeit (das ist der Abstand von s nach v), einen p[v] mit dem Wert von none (das ist der Vorgänger von v) und b[v] als false (das ergibt, ob v besucht wurde oder nicht) haben. Der Algorithmus beginnt bei s, für das gilt d[s] = 0, weil der Abstand von s nach s gleich 0 ist, und untersucht alle Nachbarknoten N(s) auf das geringste Totalgewicht W (es entspricht der Summe der Kantengewichte vom s bis zum aktuellen Knoten), dann wählt er die derzeit schnellsten Pfade aus den Knoten (mit dem geringsten Totalgewicht), die als nächstes erreicht werden können. Dies geht so lange, bis z erreicht ist.

Algorithmus 1 Dijkstra

```
1:  ∀ v ∈ V:
2:      d[v] = ∞; p[v] = null; b[v] = false
3:  d[s] = 0
4:
5:  While l:
6:      k = v', where v' ∈ l and ∀ v ∈ l: d[v'] <= d[v]
7:      b[k] = true
8:
9:      If k = z:
10:         path = invert(p[k])
11:         return path
12:
13:     ∀ n ∈ N(k), where b[n] = false:
14:         W = d[k] + w(k, n)
15:         If W < d[n]:
16:             d[n] = W
17:             p[n] = k
```

Anhand eines Beispiels wird näher demonstriert, wie Dijkstra Schritt für Schritt den Weg sucht.

Beispiel:
Betrachtet man den ungerichteten, gewichteten Graph in der Abbildung 6. Gesucht ist der kürzeste Pfad vom Startknoten A zum Zielknoten D.

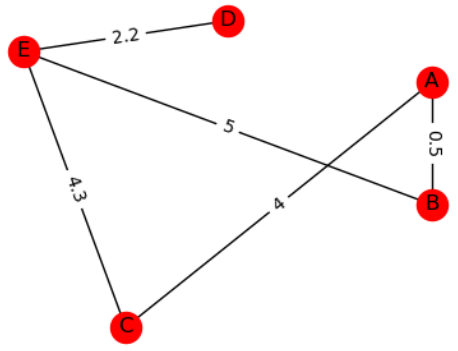


Abbildung 6: Dijkstra Beispiel

Der erste Schritt ist, drei Attribute für alle Knoten im Graphen zu initialisieren: (Abbildung 7)

- besucht = False für alle Knoten: bedeutet, dass Dijkstra den betreffenden Knoten noch nicht untersucht hat.
- vorgänger = None für alle Knoten
- abstand = ∞ für alle Knoten außer dem Startknoten A: abstand ist die Distanz zwischen dem Startknoten und den betroffenen Knoten, daher ist abstand[A] = 0, denn die Distanz von A zu A ist 0.

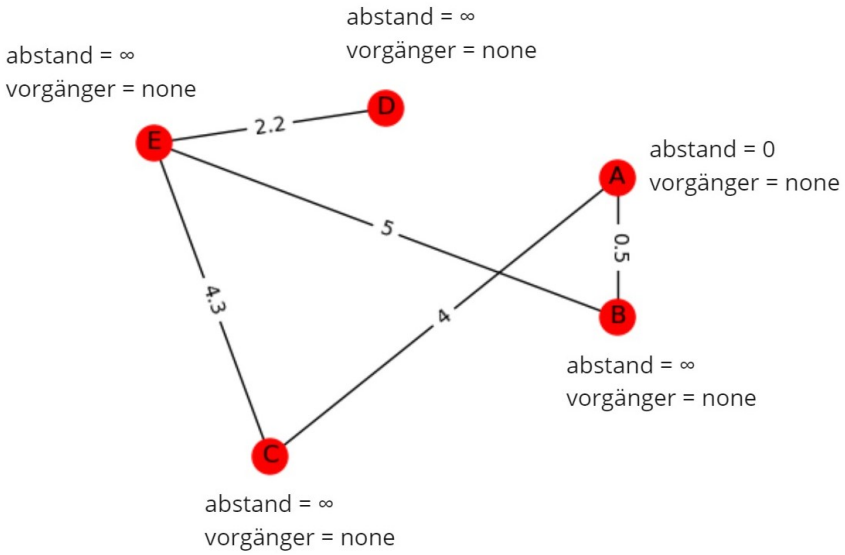


Abbildung 7: Dijkstra Beispiel - Init

Zunächst wird aus der Liste der nicht besuchten Knoten = [A, B, C, D, E] ein Knoten mit dem geringsten Abstand herausgefiltert. Dies ist der Knoten A mit dem Abstand 0. (Abbildung 8) A hat 2 Nachbarn, die nicht besucht wurden, nämlich B und C. Dann wird Dijkstra die Entfernung zu B und C analysieren und gegebenenfalls aktualisieren. Dazu fängt er zufällig mit B an.

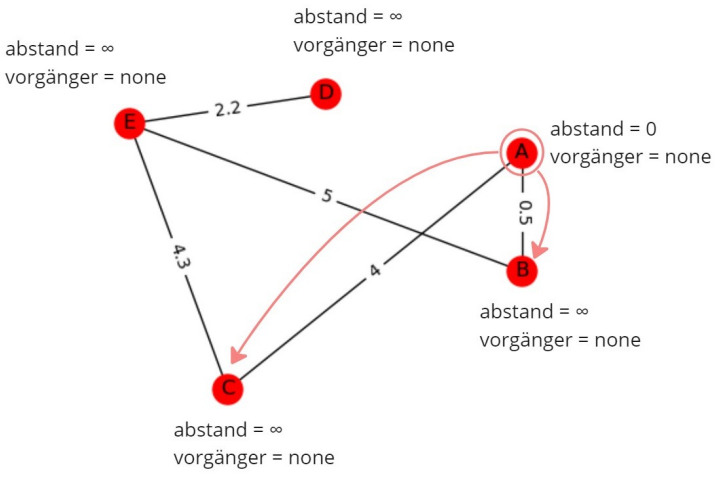


Abbildung 8: Dijkstra Beispiel - A

Es wird die Distanz von A nach B berechnet. Ist das Ergebnis kleiner als abstand[B], erhält abstand[B] dieses Ergebnis als neuen Wert und A wird als Vorgänger von B gespeichert.

→ abstand[A] + Gewicht der Kante (A, B) = 0 + 0.5 = 0.5

→ 0.5 < abstand[B] = 0.5 < ∞ ? Ja → abstand[B] = 0.5 und vorgänger[B] = A (Abbildung 9)

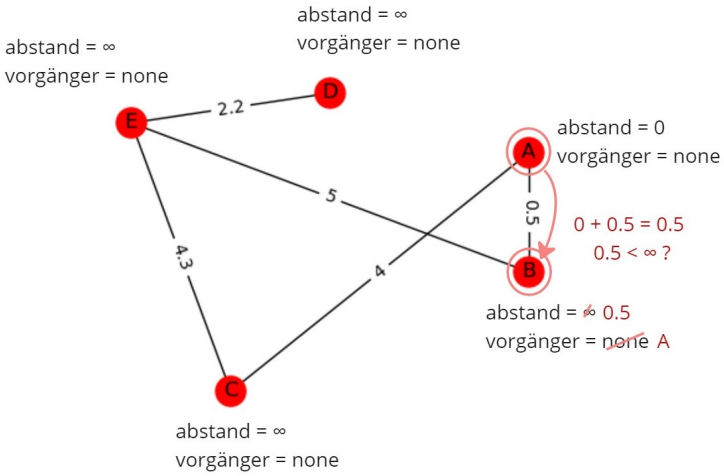


Abbildung 9: Dijkstra Beispiel - A → B

Nachdem die Berechnung zwischen A und B abgeschlossen ist, wird mit dem zweiten Nachbarn C fortgefahren und derselbe Vorgang wird wiederholt. (Abbildung 10)

→ abstand[A] + Gewicht der Kante (A, C) = 0 + 4 = 4

→ 4 < ∞ ? Ja → abstand[C] = 4 und vorgänger[C] = A

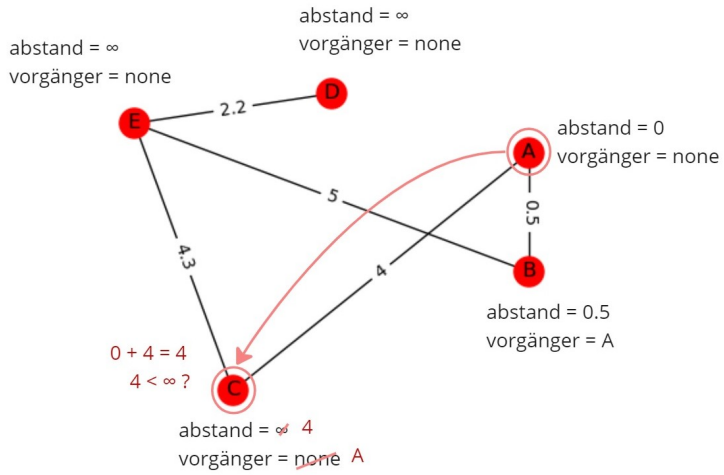


Abbildung 10: Dijkstra Beispiel - A → C

Da der Knoten A und seine Nachbarn untersucht wurden, wird der Knoten A als besucht markiert und wird nicht nochmal besucht, wie in der nachfolgenden Abbildung 11 zu sehen ist. Daher sucht Dijkstra einen neuen Knoten aus der aktualisierten Liste der nicht besuchten Knoten = [B, C, D, E] mit der geringsten Distanz, nämlich den Knoten B, der nur einen nicht besuchten Nachbarn hat, der Knoten E.

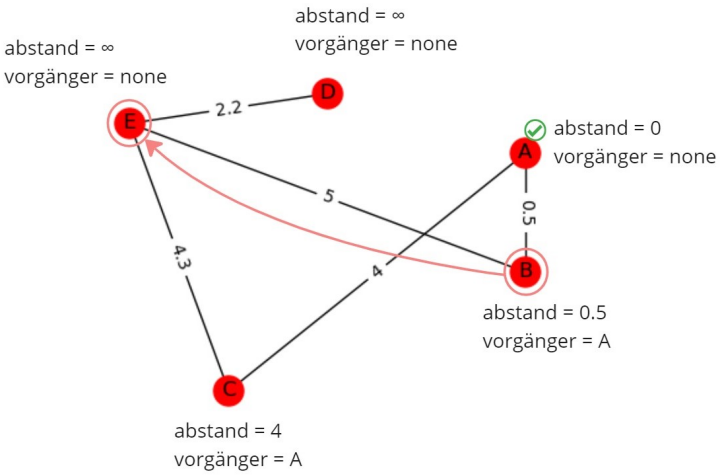


Abbildung 11: Dijkstra Beispiel - B

Jetzt wird die Distanz von B nach E berechnet (Abbildung 12)

→ abstand[B] + Gewicht der Kante (B, E) = 0.5 + 5 = 5.5

→ 5.5 < ∞ ? Ja → abstand[E] = 5.5 und vorgänger[E] = B

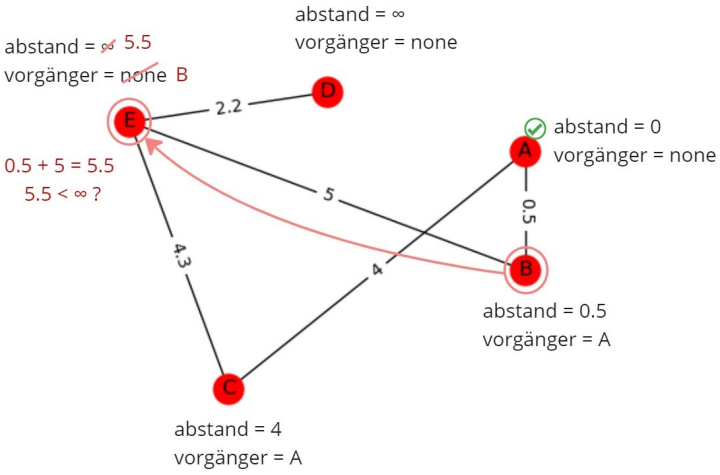


Abbildung 12: Dijkstra Beispiel - B → E

Dadurch wird auch B als besucht markiert und die Liste der nicht besuchten Knoten wird aktualisiert = [C, D, E]. Der nächste ausgewählte Knoten ist C, er hat einen nicht besuchten Nachbarn, den Knoten E. (Abbildung 13)

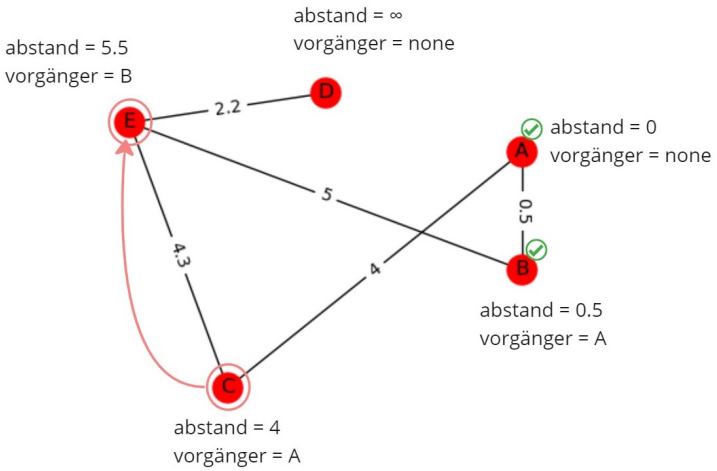


Abbildung 13: Dijkstra Beispiel - C

Dieselbe Vorgehensweise wird sich hier wiederholen. (Abbildung 14)

→ abstand[C] + Gewicht der Kante (C, E) = 4 + 4.3 = 8.3

→ 8.3 < 5.5 ? Nein → es wird nichts verändert.

C wird als besucht markiert und Dijkstra setzt fort.

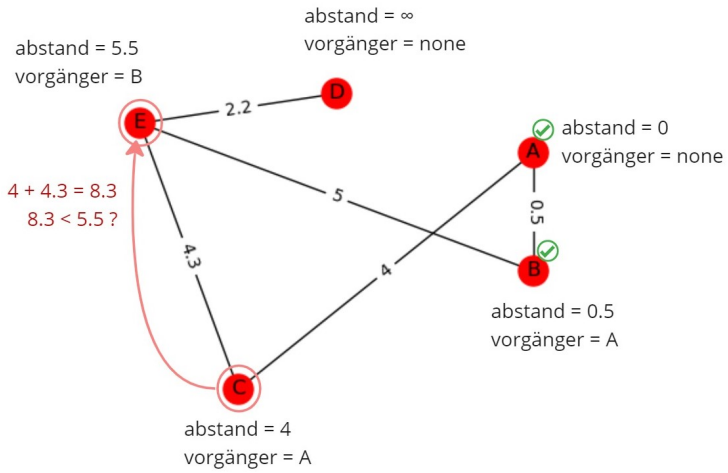


Abbildung 14: Dijkstra Beispiel - C → E

Die Liste der nicht besuchten Knoten wird erneut angepasst = [E, D]. Der neu ausgewählte Knoten ist E, er hat ebenfalls nur einen nicht besuchten Nachbarn, den Knoten D, und die Distanz zwischen den beiden wird berechnet. (Abbildung 15)

→ abstand[E] + Gewicht der Kante (E, D) = 5.5 + 2.2 = 7.7

→ 7.7 < ∞ ? Ja → abstand[D] = 7.7 und vorgänger[D] = E

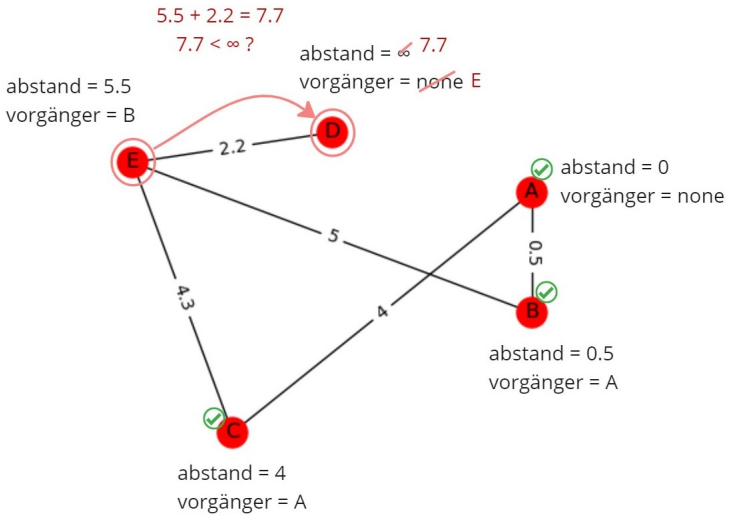


Abbildung 15: Dijkstra Beispiel - E → D

Somit wurde E als besucht markiert und die Liste der nicht besuchten Knoten enthält nur noch einen Knoten = [D]. D ist der gesuchte Zielknoten, er wurde erreicht. Um den Weg zu D zu finden, werden die Vorgänger verfolgt (Abbildung 16), d.h. D ist der Zielknoten, D hat Vorgänger E, E hat Vorgänger B, B hat Vorgänger A und A ist der Startknoten. Folglich ist der kürzeste Weg von A zu D (A → B → E → D) mit einem Gesamtgewicht von 7.7

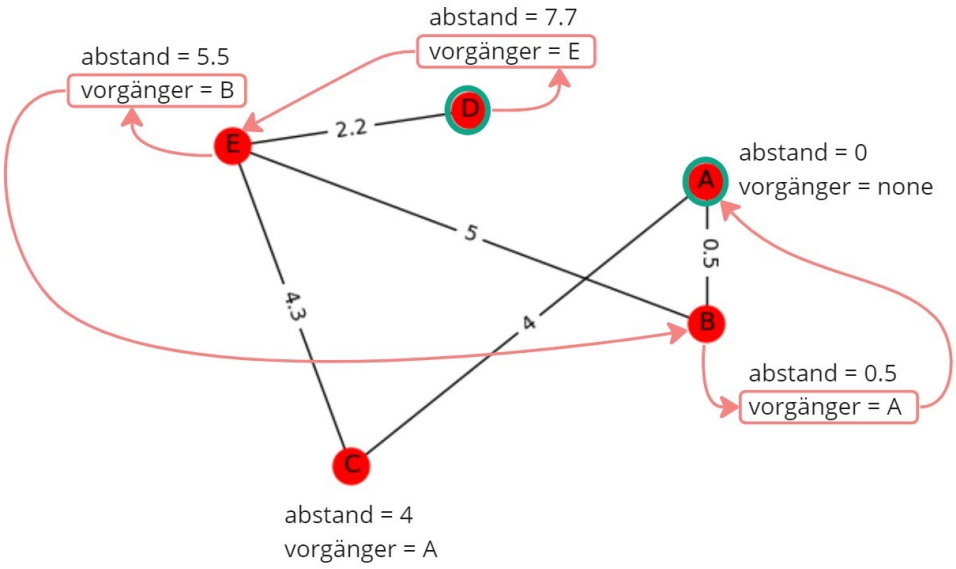


Abbildung 16: Dijkstra Beispiel - D

Dijkstra hat sein Ziel erreicht und der Weg wurde erfolgreich gefunden. Dies war aber ohne die Angaben der Kantengewichte nicht möglich, daher sollte immer gewährleistet sein, dass die Gewichte vorhanden sind. Andernfalls müssen sie auf jeden Fall berechnet werden. Im Folgenden wird beschrieben, wie die Gewichte berechnet werden können, für den Fall, dass die Knotenstandorte/-positionen auf der Erde bekannt sind und das Gewicht jeder Kante der Reisezeit entspricht, die benötigt wird, um die jeweilige Kante zu überqueren.

2.4 Reisezeit zwischen zwei Standorten

In einem Graphen, dessen Knoten auf der Erde echte Standorte darstellen, sind die Kantengewichte zu berechnen. Es wird angenommen, dass das Gewicht jeder Kante der Reisezeit entspricht, die benötigt wird, um vom ersten zum zweiten Standort zu gelangen. Um diese Reisezeit zu berechnen, wird zuerst im Abschnitt 2.4.1 erklärt, welche Attribute für die Lokalisierung eines Standortes auf der Erde von Bedeutung sind. Anhand dieser Attribute könnte die Entfernung zwischen zwei Standorten bestimmt werden (Abschnitt 2.4.2), und im Abschnitt 2.4.3 wird eine Formel vorgeschlagen, mit der die Reisezeit mithilfe dieser Entfernung bestimmt werden kann.

2.4.1 Lokalisierung eines Punktes auf der Erde

Die Position eines Standorts auf der Erde kann nicht einfach durch x und y wiedergegeben werden, da die Erde keine flache Ebene ist und daher nicht durch Koordinatenachsen dargestellt werden kann. Alternativ kann die Erde durch imaginäre Linien geteilt werden (Siehe Abbildung 18), dann können Standorte anhand der geografischen Latitude und Longitude bestimmt und lokalisiert werden.[13]

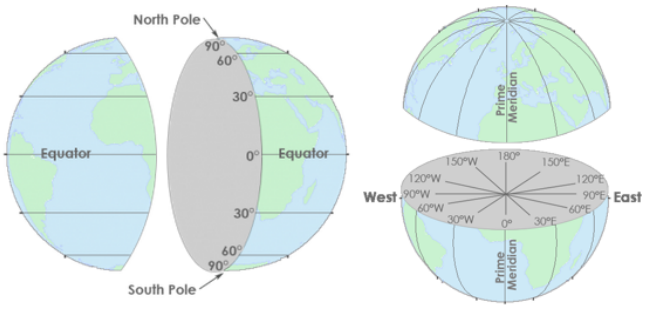


Abbildung 17: latitude-longitude [23]

Latitude ist der Winkelabstand eines Punktes vom Äquator. Der Äquator ist eine imaginäre Linie, die genau in der Mitte zwischen dem Nordpol und dem Südpol liegt und um den Erdumfang verläuft. Latitude-linien sind die horizontalen Linien des Erdnetzes, die parallel zum Äquator verlaufen. Sie werden in Grad von 0 bis 90 Grad nördlich oder südlich des Äquators gemessen.

Longitude ist der Winkelabstand eines Punktes vom Nullmeridian, wobei der Nullmeridian eine imaginäre Linie ist, die senkrecht zum Äquator vom Nordpol zum Südpol verläuft. Longitudelinien sind Halbkreise mit einem Radius, die vom Nordpol zum Südpol verlaufen. Sie werden in Grad von 0 bis 180 Grad östlich oder westlich des Nullmeridians gemessen.

Abbildung 18: Konrad-Zuse-Haus *(Google Earth)*

Beispiel:
Die Position der Universität Rostock, Institut für Informatik, Konrad-Zuse-Haus ist (54.077704 N, 12.106689 E) Dies gibt an, dass dieses Gebäude 54.077704 Grad nördlich des Äquators und 12.106689 Grad östlich des Nullmeridians liegt.

So werden zur Lokalisierung eines Standorts die beiden Attribute logitude und latitude verwendet. Hat man zwei Standorte lokalisiert, kann man die Distanz zwischen den beiden Standorten ausrechnen. Eine Formel dafür wird im nächsten Abschnitt vorgeschlagen.

2.4.2 Distanz zwischen zwei Punkten auf der Erde

Eine mögliche Variante zur Berechnung der Distanz zwischen zwei Standorten/Punkten auf der Erde ist die Haversinus-Formel [18], sie bestimmt die Großkreisdistanz zwischen den beiden Punkten. Diese Großkreisdistanz entspricht der kürzesten Entfernung zwischen zwei Punkten auf der Oberfläche einer Kugel, gemessen entlang der Kugeloberfläche. Die Formel gründet sich auf die geografische Latitude und Longitude beider Punkte und dient als Grundlage für die Navigation. Für die Punkte (ϕ_1, λ_1) und (ϕ_2, λ_2) beträgt die Distanz:

$$d = 2 * r * arcsin\sqrt{\sin^2(\frac{\phi_2 - \phi_1}{2}) + \cos(\phi_1) * \cos(\phi_2) * \sin(\frac{\lambda_2 - \lambda_1}{2})^2}$$

wobei: d = die Distanz zwischen den zwei Punkten (in km)
ϕ_1 und ϕ_2 = Latitude die Punkte (im Bogenmaß)
λ_1 und λ_2 = Longitude die Punkte (im Bogenmaß)
r = Radius der Erde = 6371 km
beschreibt.

Das Ziel war nicht, die Distanz zwischen zwei Punkten zu berechnen, sondern die Reisezeit. Diese Distanzberechnung ist jedoch der erste Schritt zur Ermittlung der Reisezeit. Distanz allein ist allerdings nicht ausreichend, es gibt noch einen weiteren Faktor, der berücksichtigt werden muss, und zwar die Geschwindigkeit.

2.4.3 Geschwindigkeit-Distanz-Zeit

Die Geschwindigkeit ist der Faktor, der die zu berechnende Reisezeit stark beeinflussen kann. Wenn z.B. ein Auto 100 Kilometer mit einer Geschwindigkeit von 50 km/h zurücklegen soll, dauert es für diese Strecke 2 Stunden. Wenn es die gleiche Strecke mit nur 30 km/h zurücklegt, dauert es 3 Stunden und 20 Minuten, was deutlich länger ist.

Die Beziehung zwischen der Geschwindigkeit, der Länge der Strecke und der zum Zurücklegen der Strecke benötigten Zeit wird durch die folgende Formel veranschaulicht:

$$\text{Geschwindigkeit} = \frac{\text{Distanz}}{\text{Zeit}}$$

Das ist die Geschwindigkeit-Distanz-Zeit Formel [6], eine äußerst einfache Formel, die den Zusammenhang, die Abhängigkeit und das Verhältnis zwischen den drei Faktoren ausdrückt.
Aus dieser Formel lassen sich zwei weitere Gleichungen ableiten, eine zur Berechnung der Distanz bei gegebener Geschwindigkeit und Zeit und andere zur Berechnung der Zeit bei gegebener Distanz und Geschwindigkeit:

$$\text{Distanz} = \text{Geschwindigkeit} * \text{Zeit} \quad \& \quad \text{Zeit} = \frac{\text{Distanz}}{\text{Geschwindigkeit}}$$

Die letztgenannte Form der Formel ist genau die Form, die für die Berechnung der Reisezeit zwischen zwei Standorten verwendet werden kann. Dazu sind die Eingaben von Distanz und Geschwindigkeit erforderlich, wobei die Distanz nach Kenntnis der lokalisierten Standorte und der festgelegten Geschwindigkeit ermittelt werden kann.

3 Flottenmanagement

Das Thema “Suche nach dem kürzesten Weg“ ist eng mit dem Thema “Flottenmanagement“ verbunden. Unter Flottenmanagement [16] versteht man die Verwaltung, Planung, Steuerung und Überwachung von Fahrzeugflotten. Dabei werden die Fahrten von Fahrzeugen (LKW, PKW, Schiffe, Bahnen, Busse) unter Berücksichtigung gewisser Einflussparameter abgestimmt, ausgewertet und optimiert. Im folgenden Kapitel wird das Flottenmanagement im maritimen Kontext und sein Verhältnis zu den Shortest-Path-Algorithmen behandelt.

3.1 Flottenmanagement im maritimen Kontext

Im maritimen Kontext befasst sich das Flottenmanagement mit einer Flotte von Schiffen, wie z. B. Containerschiffen, Tankern, Reiseschiffen und vielen anderen, die für den Transport von Fracht und Personen über Ozeane und Wasserstraßen eingesetzt werden. Dabei wird diese Flotte überwacht, verwaltet und ihre Performance optimiert. Meistens geschieht dies in Verbindung mit Algorithmen für den kürzesten Weg, um die Schiffsbewegungen zu steuern und die rechtzeitige und kostengünstige Lieferung von Waren zu gewährleisten. Beispielsweise kann ein Flottenmanagementsystem die effizienteste Route für ein Schiff mit Hilfe von Dijkstra berechnen und dann den Fahrplan des Schiffes entsprechend anpassen. Dieses Management ist effizient und hochleistungsfähig, weil es das Verfolgungssystem, das Autmatic Identification System, verwendet.

3.2 Automatic Identification System (AIS)

Das Automatic Identification System (AIS) [9] ist ein Verfolgungssystem, das im Seesverkehr und in der maritimen Industrie zur Identifizierung und Lokalisierung von Schiffen eingesetzt wird. Es wurde in den 1990er Jahren mit dem vorrangigen Ziel entwickelt, Schiffskollisionen zu verhindern und die Navigationssicherheit zu erhöhen. Heutzutage liefert es Echtzeit-Positionsdaten und Fahrpläne für fast die gesamte Handelsflotte der Welt und ermöglicht es Schiffen und Küstenbehörden, über lange Strecken hinweg miteinander zu kommunizieren. AIS-Daten dienen vielfältigen Zwecken, wie z. B. zur:

- Bewertung der Schiffsleistung. Dazu gehören Faktoren wie Schiffsauslastung, Geschwindigkeit und Kosten der Reise.
- Analyse der Hafenleistung. Die von einem AIS erhaltenen Schiffspositionen lassen sich zur Ableitung des Schiffsverkehrs, der Liegeplatzauslastung und der Terminalproduktivität von Containerhäfen nutzen.
- Gewinnung eines umfassenderen Verständnisses der Reiseleistung von einzelnen Schiffen und Flotten, z. B. Geschwindigkeitsoptimierung, Wetterrouting.

AIS ist ein mächtiges System, das das Flottenmanagement unterstützt und optimiert hat. Darüber hinaus ermöglicht es, schlechte Wetterverhältnisse wie Stürme oder Hurrikane leicht und gefahrlos nach vermeiden, was ebenfalls die Schiffsicherheit optimiert.

4 Wetterverhältnisse

Die Zielsetzung besteht darin, einen kürzesten Weg zwischen zwei Punkten auf See unter dynamischen Veränderungen zu finden. Unter allen möglichen Arten von Veränderungen werden in dieser Arbeit nur dynamische Wetterveränderungen berücksichtigt. Das Wetter ist ein wichtiger Faktor bei der Bestimmung des kürzesten Weges für ein Schiff, da es die Sicherheit, Effizienz und Geschwindigkeit der Fahrt beeinflussen kann. Daher werden in diesem Kapitel im Abschnitt 4.1 die diversen Wetterereignisse hervorgehoben und wie sie sich negativ auf Schiffe, Seeverkehr und Häfen auswirken können (Abschnitt 4.2). Schließlich wird im Abschnitt 4.3 die Wettervorhersage erwähnt und wie sie einen großen Vorteil für die Suche nach dem Weg bringt.

4.1 Wetterbedingungen auf See

Die Hauptantriebskräfte des Wetters auf See sind die **Winde** [21]. Die Winde erzeugen Meeresströmungen an der Oberfläche, indem sie über das Wasser ziehen. Es gibt auch kleinere Strömungen, die sich entlang der Ränder der Hauptströmungen bewegen und als **Wirbel** bezeichnet werden. Diese erzeugen und beeinflussen einen Großteil des Seewetters, das auf allen Ozeanen der Welt zu beobachten ist. Darüber hinaus erzeugt das Wetter **Wellen und hohen Wellengang**, der einen starken Einfluss auf Schiffe hat. Die Größe der Wellen hängt von der Stärke und Dauer des Windes ab und davon, wie weit der Wind ohne Unterbrechung weht. **Swells** sind Gruppen von großen Wellen, die größer sind als der Wind oder der Sturm, der sie erzeugt hat. Zudem können Schiffe auf See auf sogenannte **“Rogue Waves“** treffen, d. h. ungewöhnlich große Wellen inmitten kleinerer Wellen. Diese können großen Schaden anrichten. Besonders gefährlich sind Wellen und Winde, die während eines Sturms zusammen auftreten, wie z. B. **Hurrikane, auch Wirbelstürme** genannt. Dabei handelt es sich um äußerst wütende und schädliche Stürme, die Schiffe jeder Größe gefährden.
Schiffe, die in hohen geographischen Breitengraden verkehren, sind neben den altbekannten Bedrohungen durch **Eisberge und Meereis** auch der Gefahr durch **gefrierende Gischt** ausgesetzt [25]. Allerdings sind es nicht nur große Wetterereignisse, die Probleme verursachen können. Auch **Nebel** kann die weltweiten Schifffahrtsrouten beeinträchtigen [26]. Aufgrund des Nebels werden die Schiffe verlangsamt und ihr Fahrtziel wird später als geplant angefahren. Sobald sich das Wetter aufklärt, können sie wieder beschleunigen, was wiederum einen höheren Treibstoffverbrauch und höhere Treibstoffkosten zur Folge hat.

4.2 Einfluss der Wetterverhältnisse auf Schiffsverkehr

Wie bereits zu Beginn dieses Kapitels erwähnt, wird nur das Wetter als Auslöser der dynamischen Veränderungen berücksichtigt. Aber warum ist dies besonders bedeutsam zu betrachten? Die Wetterbedingungen müssen wahrgenommen werden, denn jedes ungünstige Wetterereignis kann dazu führen, dass Schiffe und Boote kentern, mit anderen Schiffen oder Objekten wie einer Brücke oder einem Eisberg kollidieren, schwer belastet werden und sogar Crewmitglieder in Gefahr bringen.
Nicht nur Schiffe, die sich auf dem Meer befinden, können betroffen sein, sondern auch Häfen [22]. Störungen in einem Containerhafen haben Auswirkungen auf die gesamte Lieferkette. Wird

ein Hafen beispielsweise von starkem Wind heimgesucht, können die Kräne nicht in Betrieb genommen werden und müssen abgeschaltet werden. Die Ladung muss dann über einen längeren Zeitraum zurückgehalten werden (sofern sie dabei nicht zerstört wird) oder wird komplett umgeleitet, was zu Lieferverzögerungen führt. Infolgedessen kann es vorkommen, dass die Behörden bei extrem ungünstigen Wetterbedingungen gewisse Häfen als Schutzmaßnahme schließen.
Ferner müssen Containerschiffe mit gewaltigen Wetterereignissen auf dem Wasser zurechtkommen und entscheiden, ob sie ihre Fahrtroute ändern oder warten, bis das Ereignis vorüber ist. Dagegen müssen Frachtschiffe, die immer nach einem engen Zeitplan fahren, entscheiden, ob sie dem Sturm trotzen können, eine Ausweichroute nehmen, warten, bis das schlechte Wetter vorbei ist, oder die Fahrt ganz absagen.

4.3 Wettervorhersage

Das geschilderte Wetter macht die maritime Einsätze und Aktivitäten risikoreich und gefährlich, doch eine Wettervorhersage kann helfen, Unglücke zu vermeiden. Die Vorhersage erfolgt durch Beobachtung und Überwachung verschiedener Parameter des Wetters und der Atmosphäre. Wichtig sind hierbei Windgeschwindigkeit und -richtung, denn die Windgeschwindigkeit sagt aus, ob es ein ruhiger oder stürmischer Tag wird, und die Windrichtung ist entscheidend, denn die Richtung, aus der der Wind kommt, kann an einem Ort große Zerstörungen anrichten, ohne eine ganze Gegend zu beeinträchtigen.
Durch eine präzise Wettervorhersage kann man einen großen Vorteil gewinnen, nämlich die Zeit. Normalerweise wird das Wetter für 10 bis 14 Tage im Voraus prognostiziert. Wenn ein Schiff auf seiner Route einen Sturm erwartet, wird es bereits 10 bis 14 Tage im Voraus darüber informiert sein. Das gibt ausreichend Zeit, um eine Lösung zu finden oder eine Entscheidung zu treffen. Allerdings lässt sich das Wetter nie hundertprozentig vorhersagen, da es sich im Laufe der Zeit ständig ändert, was wiederum eine kontinuierliche Anpassung der Fahrtroute voraussetzt, um eine hohe Sicherheit zu garantieren.

5 Stand der Technik

In der Einleitung wurde bereits betont, dass zwar zahlreiche Algorithmen entwickelt wurden, um das Problem der Suche nach dem kürzesten Weg in einem statischen Graphen zu lösen, diese Algorithmen jedoch nicht ausreichen, um das Problem in einem dynamischen Graphen ebenfalls zu bearbeiten. Hier folgt zunächst im Abschnitt 5.1 einen kurzen groben Überblick über die bekanntesten Forscher und ihre Errungenschaften zu diesem Thema. Danach werden zwei Algorithmen beschrieben, nämlich der Dijkstra-Algorithmus (Abschnitt 5.2) und der A*-Algorithmus (Abschnitt 5.3), und zwar aus zwei Gründen: Zum einen sind sie die berühmtesten und zum anderen wurden beide zu einer dynamischen Version weiterentwickelt, die für diese Arbeit von Interesse ist. Ein bemerkenswerter Ansatz zur Lösung des dynamischen Kürzeste-Wege-Problems wurde von Orda und Rom vorgeschlagen (Abschnitt 5.2.2). Ihre Methode wird ausführlich besprochen und als Vergleichspunkt für den in dieser Arbeit vorgeschlagenen Algorithmus benutzt. Schließlich wird festgelegt, welcher Algorithmus das Kernstück der Arbeit bilden wird.

5.1 Kurzer Überblick

Die Suche nach dem kürzesten Weg in einem Graphen ist zeitabhängig und dynamisch, wenn die Reisezeit zwischen den Knoten im Graphen zeitabhängig ist, wie z. B. in einem Verkehrsnetz, in dem sich die Verkehrsbedingungen mit der Zeit ändern. Das Problem der dynamischen zeitabhängigen Suche wurde zum ersten Mal von Cooke und Halsey [2] gestellt, die die von Bellman [1] veröffentlichte Arbeit erweitert haben, um den kürzesten Weg zwischen zwei beliebigen Knoten in einer endlichen Anzahl von Iterationen zu erzielen.
Eine Klassifizierung des Problems des dynamischen kürzesten Weges ermöglicht ein besseres Verständnis der verschiedenen Varianten des Problems und bietet einen Rahmen für die Entwicklung und den Vergleich von Algorithmen, die das Problem unter verschiedenen Annahmen und Beschränkungen effizient lösen können. Hierzu hat Chabini mehrere Kriterien aufgestellt [8]. Dazu gehören die Darstellung der Zeit, die diskret oder kontinuierlich sein kann, die Art des Netzes, die First-in-First-out (FIFO) oder Non-FIFO sein kann und ob es erlaubt ist, an Knoten zu warten oder nicht. Beispielsweise ist der Dijkstra-Algorithmus, der im nächsten Abschnitt vorgestellt wird, ein diskreter Algorithmus, der dem Non-FIFO-Prinzip folgt und das Warten ist nur dann erlaubt, wenn immer an festen Knoten für eine feste Zeit gewartet wird.

5.2 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus sucht nach dem kürzesten Weg in einem gewichtetem Graphen und existiert in mehreren Varianten [15]:

- Eine Variante behandelt das *Single-Source Shortest Path Problem* (Abschnitt 2.2). Sie verwendet eine Prioritätswarteschlange, um die Knoten entsprechend ihrer Entfernung zum Startknoten zu besuchen und zu bearbeiten, wonach der Knoten mit der kürzesten Entfernung die höchste Priorität hat. Außerdem speichert der Algorithmus die aktuelle kürzeste Entfernung zu jedem Knoten und merkt sich jeden besuchten Knoten.

- Eine andere Variante löst das *Single-Pair Shortest Path Problem*, sodass der Algorithmus anhält, wenn er den Zielknoten erreicht.
- Der Bidirektionale Dijkstra-Algorithmus verwendet zwei Instanzen des Dijkstra-Algorithmus, die von beiden Enden des Graphs aus laufen und stoppt, wenn sich die beiden Instanzen in der Mitte des Graphs treffen. Diese Variante ist wesentlich schneller [7], da sie die Suche sowohl vom Start- als auch vom Zielknoten aus simultan ausweitet, wodurch der Suchraum und die Anzahl der besuchten Knoten reduziert wird.

5.2.1 Dynamische Dijkstra-Algorithmus

Der Dijkstra-Algorithmus bietet eine Lösung für das Problem des kürzesten Weges in statischen Graphen und kann Veränderungen im Graphen im Laufe der Zeit nicht angemessen berücksichtigen. Um diese Einschränkung zu beheben, haben Sunita und Garg [10] eine dynamische Variante des Dijkstra-Algorithmus entwickelt, die eine retroaktive Prioritätsschlange verwendet und eine effiziente Lösung für dynamische Kürzeste-Wege-Probleme bietet.
Bei einer Änderung des Graphen berechnet der Dijkstra-Algorithmus den gesamten Pfad neu, während die vorgeschlagene dynamische Variante dies nicht tut. Die retroaktive Prioritätsschlange kann Änderungen im Graphen gut bewältigen, ohne dass der gesamte Pfad vom Startknoten aus neu berechnet werden muss. Um dies zu erreichen, prüft der dynamische Algorithmus, ob sich die Änderung auf zuvor durchgeführte Operationen auswirkt (Hierbei handelt es sich um die Operationen zur Berechnung der Kantengewichte und des Gesamtgewichts sowie zur Bestimmung des Pfades), und wenn dies der Fall ist, kehrt er zum Zeitpunkt der Änderung zurück und führt alle Operationen ab diesem Punkt erneut durch.
Der Algorithmus wurde mit Rot-Schwarz Bäumen implementiert, die einen schnellen Zugriff auf bereits erkundete Pfade gewähren. Er erweist sich als effizienter als andere Algorithmen wie Dijkstra und A*, sowohl in Bezug auf die Rechenzeit als auch auf den Speicherbedarf. Allerdings wurde der Algorithmus nur für gerichtete Graphen angepasst und kann nur mit Änderungen umgehen, die die Kantengewichte verändern, nicht aber mit Änderungen, die Knoten oder Kanten eliminieren.

5.2.2 Orda und Rom

Orda und Rom haben in ihrer Arbeit [3] den Dijkstra-Algorithmus eingesetzt und einen Ansatz zur Lösung des Problems des kürzesten Pfades von einem gegebenen Startknoten zu allen Netzknoten bei dynamischen Änderungen der Kantengewichte präsentiert. Ihre Strategie lautet, an einem Knoten solange zu warten, bis der Störfall beendet und der Weg wieder befahrbar ist, dabei haben sie zwischen den folgenden drei Möglichkeiten unterschieden:

1. Unbeschränktes Warten (UW - engl. Unrestricted waiting): Das Warten ist an jedem Knoten für eine beliebige Zeit erlaubt.
2. Verbotenes Warten (FW - engl. Forbidden waiting): Das Warten ist nirgendwo erlaubt.
3. Start-Warten (SW - engl. Source waiting): Das Warten ist nur am Startknoten erlaubt.

In Bezug auf die verschiedenen Wartearten wurde jeweils ein Algorithmus entwickelt, um den kürzesten Weg mit minimaler Verzögerung zu finden. Jeder Algorithmus hat seine eigenen Vor- und Nachteile. Der Algorithmus für Unbeschränktes Warten ist in der Lage, den kürzesten Weg zu finden, jedoch kann die Berechnung möglicherweise viel Zeit in Anspruch nehmen. Im Vergleich dazu ist der Algorithmus für Start-Warten schneller als der Algorithmus für Unbeschränktes Warten, aber es besteht die Möglichkeit, dass er den kürzesten Weg nicht findet. Der Algorithmus für Verbotenes Warten ist der schnellste, hat jedoch das Risiko, nicht den kürzesten Pfad zu entdecken.

Da der Ansatz von Orda und Rom eventuell auch eine Lösung für das in dieser Arbeit behandelte Problem bietet, nämlich das Finden des kürzesten Weges unter dynamischen Aspekten, wird er implementiert und zu Vergleichszwecken verwendet. Der Algorithmus, der dafür verwendet wird, ist der Algorithmus für Unbeschränktes Warten, da es wichtig ist, den kürzesten Weg ermitteln zu können. Das Unbeschränktes Warten wurde für zwei Fälle umgesetzt: einen, bei dem eine bestimmte Startzeit vorgegeben ist, und den anderen für alle Startzeiten, der eine Verallgemeinerung des ersten Falles darstellt. Für die Zwecke dieser Arbeit wird der erste Fall betrachtet. Daher wird dieser Form zunächst näher vorgestellt.
Er ist eine direkte Erweiterung von Dijkstra und hat die Zeitkomplexität $O(n^2)$. Er sucht für einen gegebenen Startknoten s und eine Startzeit t_s kürzeste Pfade und minimale Verzögerungen zwischen s und allen anderen Knoten für diese Startzeit. Jeder Knoten $v \in V$ wird durch die frühestmögliche Ankunftszeit an diesem Knoten gekennzeichnet. X ist die permanente Kennzeichnung des Knotens (NULL zeigt an, dass der Knoten nicht permanent gekennzeichnet ist), und Y ist seine temporäre Kennzeichnung. N(j) entspricht die Nachbarn von j. Der Algorithmus erstellt auch den kürzesten topologischen Spannbaum mit der Wurzel s, der durch die vom Algorithmus berechneten Werte von f konstruiert werden kann. f ist die Identität des Vaters von Knoten v in diesem Baum. Außerdem ist die Fuktion $D_{ik}(t)$, die die Summe aus der Wartezeit und dem Gewicht der Kante (i, k) ergibt, ein Teil der Eingabe des Algorithmus.

5.3 A*-Algorithmus

Der A*-Algorithmus [14] wird zur Ermittlung des kürzesten Weges zwischen zwei Knoten in einem Graphen mit positiven Kantengewichten angewendet und kann als eine Erweiterung von dem Dijkstra-Algorithmus betrachtet werden. Er verwendet eine heuristische Funktion, um die Entfernung von jedem Knoten zum Zielknoten zu schätzen, sowie eine Prioritätsschlange. In dieser Schlange werden die Knoten nach der Summe des bisher zurückgelegten Weges und der Schätzung des noch zu durchlaufenden Weges zum Zielknoten geordnet. Der Algorithmus wählt dann den Knoten mit der niedrigsten Summe bzw. der höchsten Priorität aus.
Im Gegensatz zum Dijkstra-Algorithmus arbeitet der A*-Algorithmus mit Heuristiken, d. h. während Dijkstra den Knoten mit der kürzesten Entfernung zum Startknoten wählt, wählt A* die Knoten aus, die wahrscheinlich schnell zum Ziel führen. Der A*-Algorithmus kombiniert somit eine Breitensuche mit einer heuristischen Schätzung, um den Suchraum effizient zu durchsuchen.

Algorithmus 2 UW - Unrestricted Waiting

```
1: ∀ v ∈ V:
2:       Y_v = ∞
3:       X_v = null
4:       f_v = none
5: X_s = ts
6: f_s = none
7: j = s
8: ∀ n ∈ N(j), If X_n = null:
9:       Y_n = min (Y_n, X_n + D_jn(X_j))
10:      If Y_n changed:
11:           f_n = j
12:
13: If ∀ v ∈ V: X_v ≠ null: Stop
14: Else:
15:      l = v', where X_v' = null and Y_v' ≤ Y_v ∀ v ∈ V, with X_v = null
16:      X_l = null
17:      j = l
18:      Go to line 8
```

5.3.1 Dynamische A*-Algorithmus

Chabini und Lan [5] haben effiziente Anpassungen des A*-Algorithmus zur Berechnung der schnellsten Pfade in dynamischen Graphen vorgestellt, sowohl für eine als auch für mehrere Abfahrtszeiten am Startknoten. Der Hauptunterschied zwischen dem vorgestellten dynamischen A*-Algorithmus und dem ursprünglichen A*-Algorithmus liegt darin, dass der dynamische A*-Algorithmus Reisezeitänderungen in einem zeitabhängigen Graphen berücksichtigt, während der ursprüngliche A*-Algorithmus von einem statischen Graphen mit festen Kantengewichten ausgeht.

Welcher Algorithmus wird in dieser Arbeit eingesetzt?

Der A*-Algorithmus ist, wie bereits erläutert, schneller als der Dijkstra-Algorithmus, wodurch er in dieser Arbeit von Vorteil ist, da der Graph, in dem der kürzeste Weg gesucht wird, recht groß und umfangreich ist. Auf der anderen Seite muss für den A*-Algorithmus eine heuristische Funktion implementiert werden, die relativ kompliziert sein kann, und für den Fall, dass die verwendete Heuristik die Kosten überschätzt, findet der A*-Algorithmus keinen Weg [14].
Nicht nur aus diesen Gründen wird im weiteren Verlauf der Arbeit der Dijkstra-Algorithmus eingesetzt, sondern auch deshalb, weil das Ziel der Arbeit nicht darin besteht, den besten und effektivsten Algorithmus auszuwählen, sondern verschiedene Methoden auf einen Algorithmus hin getestet und verglichen werden. Dabei macht es keinen Unterschied, ob es sich um den Dijkstra, den Bidirektionalen oder den A*-Algorithmus handelt. Hierfür reicht die einfachste Variante von Dijkstra, die den kürzesten Weg suchen kann, aus.

6 Vorstellung der eingesetzten Daten

Im Folgenden werden die Daten vorgestellt, die im Rahmen dieser Arbeit berücksichtigt werden, um einen Graphen zu erstellen. Dieser Prozess erfolgt in zwei Schritten. Zunächst werden die Knoten und Kanten beschrieben (Abschnitt 6.1), einschließlich der bereits vorhandenen Attribute. Dann wird im Abschnitt 6.2 der Graph aus diesen Knoten und Kanten erstellt und visualisiert. Im Abschnitt 6.3 wird eine Übersicht vermittelt, die das zu lösende Problem zusammenfasst.

6.1 Die Knoten und Kanten

Für diese Arbeit wurden die Daten von FleetMon [12] bereitgestellt. Sie enthalten echte Koordinatendaten von Häfen und Zwischenpunkten auf dem Meer und sind in zwei CSV-Dateien aufgeteilt - eine für Knoten und eine für Kanten. CSV-Dateien bieten den Vorteil, dass sie umfangreiche Datensätze einfach speichern und in vielen verschiedenen Formaten lesen können. Zur Verarbeitung dieser Daten wird das Python-Paket “pandas“ verwendet. Pandas [19] ist eine Programmbibliothek für Python, die Datenstrukturen und Operatoren zur Verfügung stellt, um numerische Tabellen und Zeitreihen zu verarbeiten, zu analysieren und darzustellen.

Die Abbildung 19 zeigt einen Ausschnitt aus der Datei “nodes.csv“, der nur die ersten drei Zeilen enthält. Insgesamt enthält die Datei 2335 Zeilen (:= Knoten), die den Häfen und Zwischenpunkten entsprechen. Jeder Knoten in der Datei verfügt über eine eindeutige ID, die zur Identifikation des Knotens verwendet wird. Darüber hinaus enthält jeder Knoten folgende Informationen:

1. Tiefgang (engl. draught): An vielen Knoten ist die Einfahrt nur mit gewissen vorgegebenen Tiefgängen erlaubt. Die Datei enthält Tiefgangswerte zwischen 2,0 und 30,0. Einige Knoten besitzen den Wert NaN, was bedeutet, dass es keinen festen Tiefgang gibt und alle Schiffe einfahren dürfen.

2. is_port: Ein Feld, das angibt, ob es sich bei dem Knoten um einen Hafen oder einen Zwischenpunkt handelt.

3. Longitude und 4. Latitude: Die geografischen Koordinaten des Knotens (Abschnitt 2.4.1)

	id	draught	is_port	lon	lat
0	6781	NaN	f	18.521576	59.519693
1	5072	NaN	f	19.009094	60.501878
2	6780	NaN	f	19.105225	59.717638

Abbildung 19: Knoten Exemplar

Um die Kanten zwischen den Knoten zu modellieren, wurde die Datei “edges.csv“ auf die gleiche Art und Weise wie die Datei “nodes.csv“ gelesen. Die Abbildung 20 zeigt ebenfalls den Inhalt der allerersten drei Zeilen der Datei. Jede Kante wird durch eine eindeutige ID identifiziert und verbindet zwei Knoten durch ihre jeweiligen IDs. Insgesamt gibt es 3562 Kanten in den Daten.

	edge_id	from_id	to_id
0	22919	9724	15010
1	6032	8255	8253
2	128	5091	5059

Abbildung 20: Kanten Exemplar

6.2 Der Graph

Zusammen mit den CSV-Dateien wurde auch die unten stehende Abbildung 21 vorgelegt, die einen Graphen (Abschnitt 2.1) visualisiert und einen Überblick darüber gibt, wie die verschiedenen Häfen bzw. Zwischenpunkten miteinander verbunden sind. In diesem Graphen soll der kürzeste Weg zwischen zwei Knoten gesucht werden.

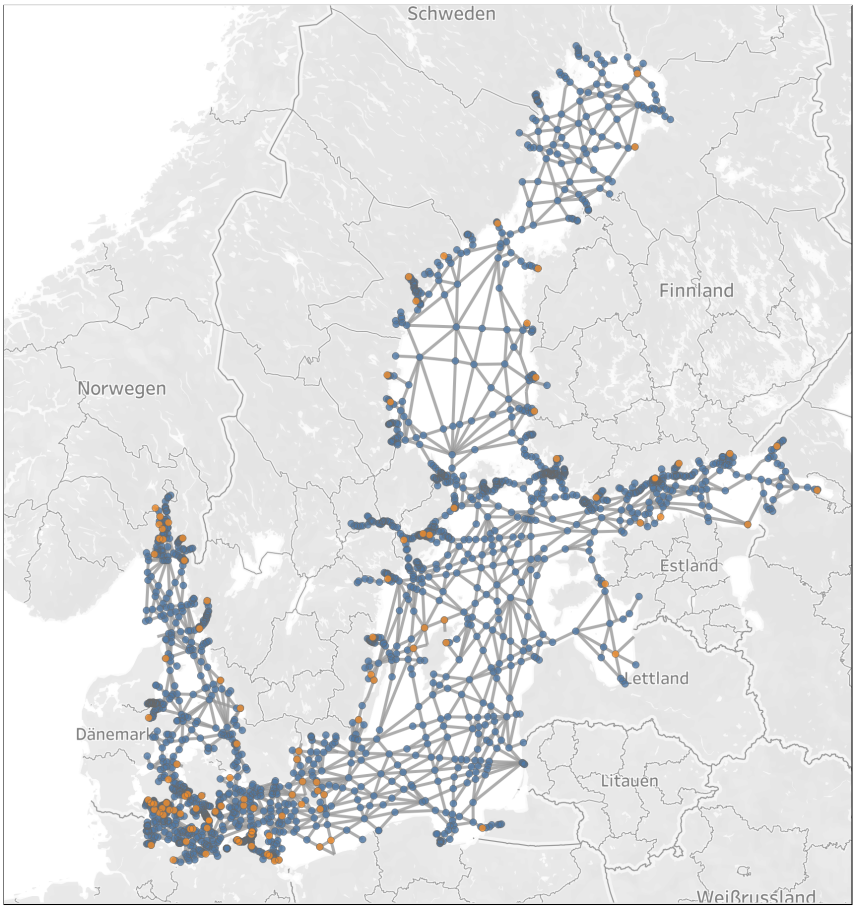


Abbildung 21: Graph - Initialzustand - FleetMon

Um eine Lösung für dieses Problem finden und evaluieren zu können, soll ein Graph gebildet werden, der genau diese Struktur in der Abbildung 21 aufweist. Zur Formulierung des Graphen werden Knoten und Kanten benötigt, welche bereits vorliegen. Der Graph kann somit erstellt werden, wofür ein leerer Graph ohne Knoten oder Kanten mittels NetworkX generiert wurde (Abschnitt 2.1.2), woraufhin die Knoten und Kanten aus den importierten CSV-Dateien mit all ihren Attributen zu dem Graphen hinzugefügt wurden. Die Abbildung 22 veranschaulicht diesen entworfenen Graphen, wobei die roten Knoten Häfen und die blauen Knoten Zwischenpunkte sind. Um zu verhindern, dass der Graph jedes Mal eine andere Form annimmt, und um den gleichen Graphen wie den von FleetMon in der Abbildung 21 zu erhalten, muss die Position der Knoten angegeben werden. Das ist leicht zu bewerkstelligen, da jeder Knoten mit Longitude und Latitude versehen ist. Dadurch hat der Graph 22 wie beabsichtigt in etwa die gleiche Struktur und die gleiche Form wie der Graph in der Abbildung 21.

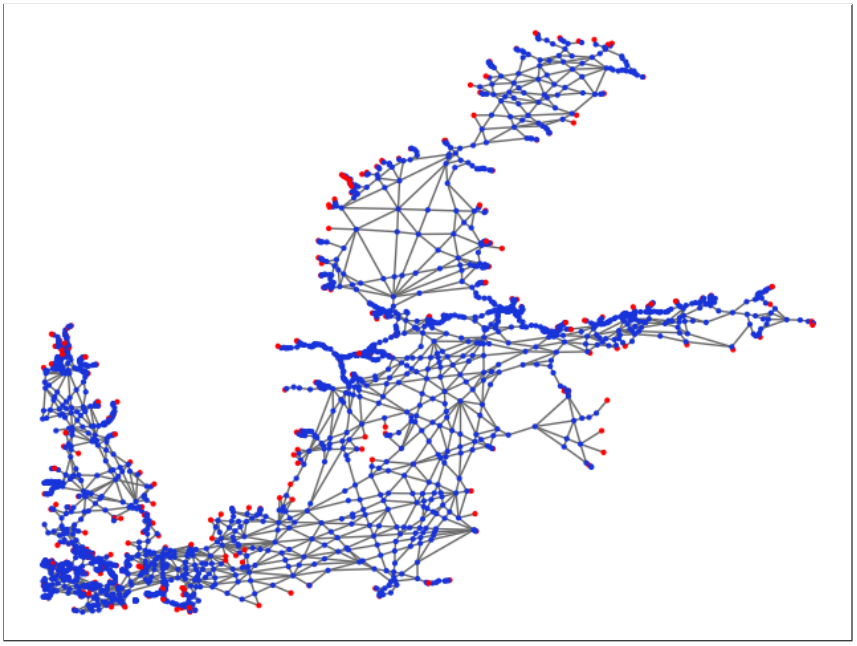


Abbildung 22: Graph - Initialzustand - Generiert

Am Ende des vorigen Kapitels 5 wurde beschlossen, dass der Dijkstra-Algorithmus im weiteren Verlauf der Arbeit verwendet wird. Der Dijkstra-Algorithmus benötigt jedoch Kantengewichte, um den Pfad berechnen zu können. Daher ist es erforderlich, Kantengewichte zu berechnen, dabei richtet sich das Gewicht einer kante nach der Zeit, die für das Überqueren der kante benötigt wird. Diese Berechnung erfolgt, indem die Longitude und Latitude des Knotens abgerufen, von Grad in Bogenmaß umgewandelt danach in die Haversinus-Gleichung eingesetzt werden, wie im Abschnitt 2.4.2 beschrieben, was die Distanz ergibt. Soweit die Distanz berechnet wurde, lässt sich die Zeit unter Angabe der Geschwindigkeit ermitteln (Abschnitt 2.4.3).
Die Geschwindigkeit ist ein Faktor, der jeder Kante nach folgendem Prinzip zugeordnet wird:

- Ist einer der durch die Kante verbundenen Knoten ein Hafen, wird die Geschwindigkeit auf 7 kn angesetzt. (1 kn = 1.852 km/h)
- Ansonsten wird die Geschwindigkeit auf 12 kn angesetzt.

Sowohl Distanz als auch Gewicht werden den Kanten als zusätzliche Attribute zugefügt.

6.3 Problemstellung

An dieser Stelle wird das zu lösende Problem im Rahmen dieser Arbeit kurz zusammengefasst und beschrieben. Dabei werden die relevanten Aspekte des Problems erläutert und die Ziele der Arbeit herausgestellt, um eine klare Übersicht über das Thema zu haben:

Sei G = (V, E) ein ungerichteter, gewichteter Graph mit der Knotenmenge V = $\{v_1, v_2, ..., v_n\}$ und der Kantenmenge E = $\{(v_i, v_j) \mid (v_i, v_j) \in V \times V, i \neq j\}$. Der Graph besteht aus n = 2335 Knoten und m = 3562 Kanten, wobei jede Kante (v_i, v_j) ein zeitabhängiges Gewicht $w(v_i, v_j)$ trägt, das der benötigten Zeit von Knoten v_i zu Knoten v_j entspricht.
Untersucht werden mehrere Lösungsansätze zur Behandlung des *Single-Pair Shortest Path Problems* im maritimen Kontext, wo die Knoten Häfen oder Zwischenpunkte auf dem Meer darstellen. Ziel ist es, den kürzesten Weg für Schiffe zwischen dem Startknoten $s \in V$ und dem Zielknoten $z \in V$ zu finden. Jeder Weg zwischen zwei Knoten v_1 und v_2 besteht aus einer Folge von Kanten $\{e_1, e_2, ..., e_n\}$ und hat ein Gesamtgewicht W, das sich aus der Summe der ihn bildenden Kanten ergibt, sodass $W = e_1 + e_2 + ... + e_n$. Der kürzeste Pfad hat somit das kleinste Gesamtgewicht.
Der Graph wird nicht statisch, sondern dynamisch behandelt, und es werden nur wetterbedingte Änderungen berücksichtigt, die zur vorübergehenden Beseitigung von einer oder mehreren Kanten bzw. Knoten führen. Diese Änderungen werden als topologische Änderungen bezeichnet und können zu Gewichtsveränderungen führen.
Zur Lösung des Problems wird der Dijkstra-Algorithmus eingesetzt, der das Kernstück jedes vorgestellten Lösungsansatz darstellt. Eine Priority Queue wird genutzt, um die Knoten nach ihrem kürzesten Abstand zum Startknoten s zu behandeln, ohne Berücksichtigung des First-In-First-Out-Prinzips.

Die verschiedenen Ansätze werden im folgenden Kapitel präsentiert und beschrieben.

7 Konzept

Nachdem das Problem formuliert und das Ziel gesetzt wurde, kann dieses Kapitel mit der Vorstellung verschiedener Lösungsansätze beginnen, die darauf abzielen, den kürzesten Weg zu finden. Bevor diese Ansätze vorgestellt werden, wird ein kleines Testszenario präsentiert, in dem die verschiedenen Lösungsansätze angewendet werden. Zu diesem Zweck wird das Szenario zuerst beschrieben und eingeführt.

7.1 Testszenario

Ein Szenario ist eine gute Möglichkeit, das Problem und die Lösung zu veranschaulichen. Auf dieses Szenario werden die Ansätze angewendet und ihre Ergebnisse visualisiert.
Zu diesem Zweck werden zufällig zwei Knoten aus dem im Abschnitt 6.2 definierten Graphen ausgewählt, nämlich ein Startknoten s und ein Zielknoten z. Angenommen, ein Schiff mit einem Tiefgang von 2.0 möchte vom Hafen s (mit der ID 7702) zum Hafen z (mit der ID 23070) fahren. Die geplante Abfahrtszeit ist der 23.03.2023 um 11:30 Uhr, und das Schiff möchte sein Ziel mit möglichst minimaler Fahrzeit bzw. minimalen Gesamtgewicht W erreichen. Hierzu wird der Dijkstra-Algorithmus den schnellsten und kürzesten Weg suchen. Die Vorgehensweise von Dijkstra ist im Abschnitt 2.3 ausführlich erläutert.
Die darauffolgende Abbildung 23 zeigt den kürzesten Pfad, der mit dem Algorithmus von Dijkstra gefunden wurde. Start- und Zielknoten sind farblich gekennzeichnet, der orangefarbene Knoten ist der s und der grünfarbene Knoten ist der z. Dieser Pfad verläuft über 48 Knoten und es dauert ungefähr 3692.61 Minuten ≈ 61.54 Stunden ≈ 2.56 Tage, um die Strecke zurückzulegen, was W entspricht, das sich aus der Summe der entstehenden Kanten ergeben hat.

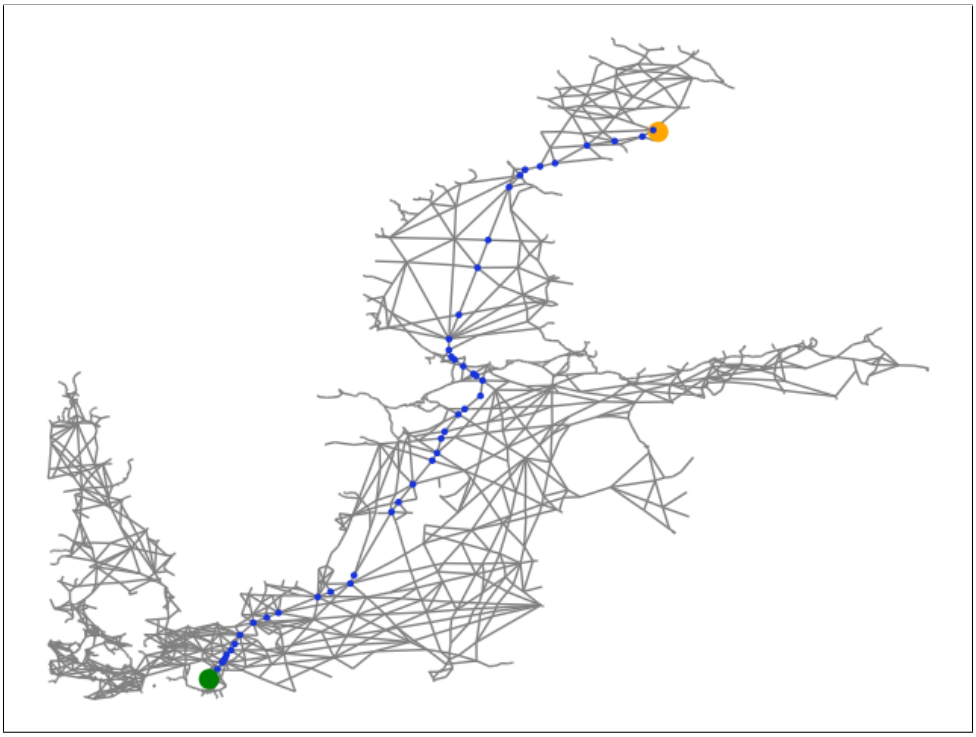


Abbildung 23: Dijkstra - Kürzester Pfad

In diesem Testszenario wurde bislang der Graph statisch behandelt und es wurden keine dynamischen Änderungen vorgenommen. Um dies zu ändern und den Graphen dynamisch zu modifizieren, wird nun eine Sturmmeldung hinzugefügt. Angenommen, es wird ein Sturm erwartet. Dazu wurden bestimmte Informationen zur Verfügung gestellt, die besagen, dass der Sturm am 23.03.2023 um 13:00 Uhr auf bestimmte Knoten aufziehen wird und bis zum nächsten Tag, dem 24.03.2023 um 11:30 Uhr, andauern wird. Außerdem wurde mitgeteilt, dass insgesamt 7 Knoten und alle dazwischen liegenden Kanten vom Sturm betroffen sein werden. Diese Knoten werden als Sturmknoten und die Kanten dazwischen als bezeichnet.

Solange der gefundene Pfad nicht vom Sturm betroffen ist, muss keine Maßnahmen ergriffen werden. Um daher sicherzustellen, dass das Schiff nicht vom Sturm beeinträchtigt wird, prüft ein ganz einfacher Algorithmus 3 jeden Sturmknoten V_s daraufhin, ob er zu einem der Knoten gehört, die zum gefundenen Pfad r zählen. Wenn dies der Fall ist und ein Knoten v gefunden wurde, der zu V_s und zu r gehört, wird die Ankunftszeit des Schiffes an diesem Knoten berechnet t. Wenn das Schiff voraussichtlich während des Sturms an diesem Knoten ankommen wird, bedeutet dies, dass der ursprüngliche Pfad vom Sturm beeinflusst wird.
Der Algorithmus erhält als Eingabe Daten zum Schiff und zum Sturm. Vom Schiff erhält er die Abfahrtszeit t_o, die Fahrtstrecke r, die die IDs der Knoten enthält, die den gefundenen Weg bilden, und den Abstand vom Startknoten s zu jedem Knoten in der Fahrtstrecke d. Vom Sturm erhält er die Sturmknoten V_s und die Zeiten, wann der Sturm aufzieht t_s und endet t_e. Der Algorithmus gibt als Ausgabe True zurück, wenn das Schiff vom Sturm betroffen ist, andernfalls gibt er False zurück.

Algorithmus 3 Ship affected by Storm

```
1: ∀ v ∈ V_s:
2:     If v ∈ r:
3:         t = t_0 + d[v]
4:         If t ∈ [t_s, t_e[:                    ▷ ignore second deviations
5:             return True
6: return False
```

Für das Testszenario und den definierten Sturm wird der Fahrtstrecke an einem oder mehreren Knoten und Kanten durch den Sturm beeinflusst. Um die Auswirkungen des Sturms auf die Fahrt zu minimieren bzw. zu beseitigen, soll etwas dagegen unternommen werden. Ein erster Lösungsansatz ist, einen alternativen neuen Pfad zu suchen.

7.2 Einen neuen Pfad suchen

Um mit den Auswirkungen des Sturms auf die geplante Fahrstrecke umzugehen, bietet es sich an, einen neuen Pfad zu suchen. Dies ist wahrscheinlich die einfachste Lösung. Dazu werden in dem Graphen G = (V, E) alle Knoten $V_s \in V$, die vom Sturm betroffen sind, sowie alle Kanten $E_s \in E$, die V_s miteinander verbinden und ebenfalls vom Sturm betroffen sind, einfach ignoriert. Sie werden bei der Suche nach dem neuen Pfad nicht berücksichtigt, als ob sie nicht existierten. Die Distanz vom Startknoten s zu jedem Knoten $v_s \in V_s$ wird nicht berechnet. Der neu gefundene Pfad könnte eventuell als zweitschnellster Pfad betrachtet werden. Es ist jedoch wichtig, sicherzustellen, dass dieser neue Pfad weder durch die Knoten V_s noch durch die Kanten E_s auf seiner Route führt, um erneute Verzögerungen oder Probleme zu vermeiden.

Der vorliegende Algorithmus 4 ist eine Erweiterung des Dijkstra-Algorithmus mit einigen Anpassungen, um die Auswirkungen des Sturms auf die Fahrtstrecke zu berücksichtigen und hat die Zeitkomplexität $O(n^2)$. Zu Beginn wird jeder Knoten $v \in V$ in G initialisiert, indem v ein Abstand d[v], ein Vorgänger p[v] und ein Besuchsstatus b[v] zugewiesen werden. Der Abstand d[v] entspricht dem Abstand von s nach v und wird auf Unendlichkeit initialisiert, da zu Beginn nicht bekannt ist, wie weit s von v entfernt ist. Der Vorgänger p[v] wird allen Knoten mit dem Wert none zugewiesen und der Besuchsstatus b[v] unterscheidet die Knoten darin, ob sie bereits besucht wurden oder nicht. Jeder Knoten v wird nur einmal besucht. Der Startknoten s erhält d[s] gleich 0, da der Abstand von s nach s 0 beträgt.
Der Algorithmus arbeitet in Schleifen. In jeder Schleife wird eine Liste l bestimmt, die jeden Knoten v enthält, für den b[v] = false gilt, d.h. der noch nicht besucht wurde. Aus l wird dann ein Knoten k ausgewählt, der den minimalen Abstand d[k] besitzt, und dieser wird bearbeitet. Dazu wird k zuerst als besucht markiert. Wenn k nicht Teil von V_s ist, werden alle Nachbarn N(k) von k gesucht, die ebenfalls nicht besucht sind. Dann wird für jeden Nachbarn n geprüft, ob er nicht Teil von V_s ist, ob die Kante zwischen k und n, d.h. E(k, n), nicht Teil von E_s ist, und ob der Tiefgang des Schiffs T dem Tiefgang von n T(n) entspricht. Wenn all dies erfüllt ist, wird der Abstand von s nach n berechnet. Dabei wird der Gesamtgewicht W berechnet, der sich aus der Summe der Gewichte der Kanten, die den Pfad von s nach n bilden, ergibt, d.h. W = d[k] (Abstand von s nach k) + w(k, n) (Gewicht der Kante E(k, n)). Ist W kleiner als d[n], wird d[n] = W zugewiesen und k als Vorgänger von n gespeichert.
Dieser Vorgang wiederholt sich, bis alle Knoten v untersucht wurden. Wenn $v \in V_s$ aus l mit minimalem d[v] ausgewählt wird, wird es als besucht markiert, obwohl v nicht bearbeitet wird, um zu verhindern, dass v nicht erneut besucht wird. Andernfalls könnte l niemals leer werden und der Algorithmus würde in einer Endlosschleife laufen. Allerdings wenn der Zielknoten z angetroffen wird, wird die Schleife doch abgebrochen und der Pfad von s nach z zurückgegeben.

Algorithmus 4 Alternativen Pfad suchen

```
1:  ∀ v ∈ V:
2:      d[v] = ∞
3:      p[v] = null
4:      b[v] = false
5:  d[s] = 0
6:
7:  While l:
8:      k = v', where v' ∈ l and ∀ v ∈ l: d[v'] <= d[v]
9:      b[k] = true
10:
11:     If k = z:
12:         path = invert(p[k])
13:         return path
14:
15:     If k ∉ V_s:
16:         ∀ n ∈ N(k), where b[n] = false:
17:             If n ∉ V_s and E(k, n) ∉ E_s and T(n) ≥ T:
18:                 W = d[k] + w(k, n)
19:                 If W < d[n]:
20:                     d[n] = W
21:                     p[n] = k
```

Dieser Algorithmus wird im nächsten Schritt auf das im vorigen Abschnitt 7.1 vorgestellte Testszenario angewandt, um einen neuen Pfad zu finden.

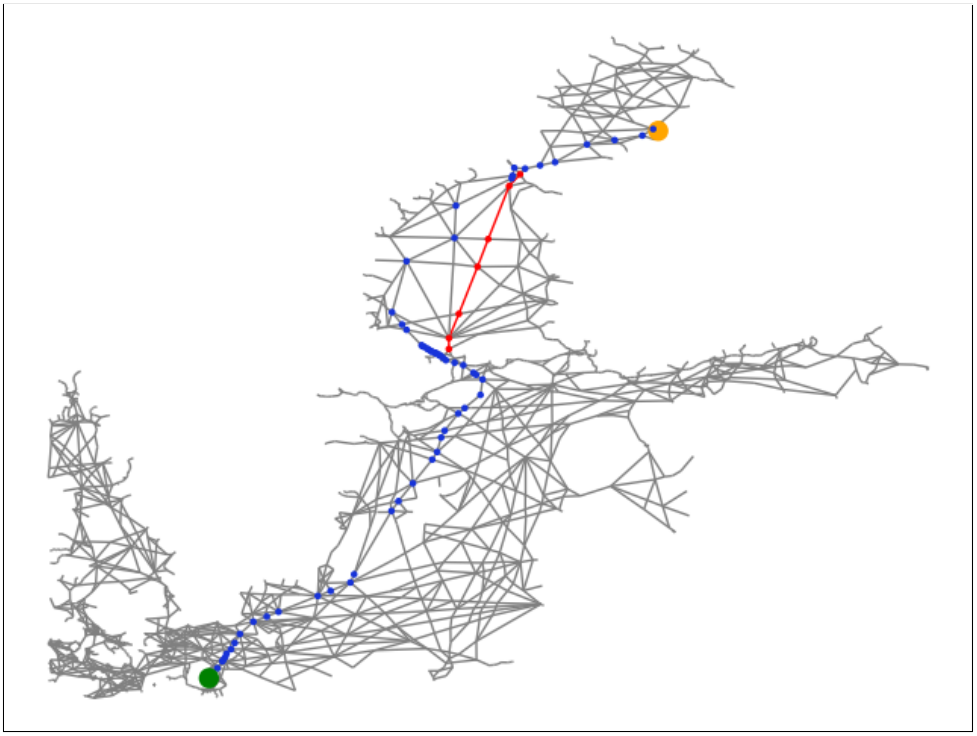


Abbildung 24: Suche nach neuen Pfad

Der Algorithmus hat einen neuen kürzesten Weg gefunden, wie in der obigen Abbildung 24 zu sehen ist. Die V_s und E_s sind rot markiert. Vergleicht man diese Abbildung mit der ursprünglichen Dijkstra-Abbildung in 23, stellt man fest, dass diese rot markierten Knoten und Kanten tatsächlich Teil der ursprünglichen Route sind. Der neue Pfad ist blau markiert und verläuft nicht durch V_s. Vom orangefarbenen Startknoten s bis zum grünfarbenen Zielknoten z erstreckt sich der Pfad über 64 Knoten mit einem Gesamtgewicht W ≈ 3933.27 Minuten ≈ 65.55 Stunden ≈ 2.7 Tagen.

Dieser Lösungsansatz hat im vorherigen Szenario erfolgreich gezeigt, dass er in der Lage ist, sein Ziel zu erreichen, indem er einen neuen Pfad identifiziert. Dabei hilft er, Schiffe vor Schäden durch schlechtes Wetter zu schützen.

7.3 Geschwindigkeit manipulieren

Eine andere Idee zur Lösung des Problems besteht darin, den gefundenen Dijkstra-Pfad beizubehalten und keinen neuen Weg zu berechnen. Dies könnte auf eine andere mögliche Lösung hinweisen, nämlich die Manipulation der Geschwindigkeit, um je nach Bedarf schneller oder langsamer für bestimmte Strecken zu fahren. Wenn das Schiff kurz vor dem V_s ist und die Sturmzone vor dem Eintreffen des Sturms überqueren möchte, aber nicht genügend Zeit dafür hat, sollte es beschleunigen, um diese Strecke -vor Beginn des Sturms- zu überqueren. Die genaue Berechnung der Zeit ist ein wichtiger Faktor, da kein Schiff plötzlich in der Sturmzone stecken bleiben möchte.
Wenn dies aufgrund von Zeitmangel nicht möglich ist, kann das Schiff von der Startknoten s bis zum V_s langsamer fahren, um am V_s anzukommen, nachdem der Sturm vorüber ist. Auf diese Weise entsteht keine Gefahr und das Schiff kann später auf der Route wieder beschleunigen, um die verlorene Zeit aufzuholen. In beiden Fällen müssen jedoch die maximal und minimale zulässige Geschwindigkeit berücksichtigt werden.

Obwohl diese Idee theoretisch eine angemessene Lösung darstellen könnte, ist sie in der Praxis leider abgelehnt. Die meisten Schiffe bevorzugen es, länger auf See zu bleiben, anstatt zu beschleunigen, da eine höhere Geschwindigkeit mehr Treibstoff verbraucht, was wiederum höhere Kosten verursacht. Zudem erhöht eine höhere Geschwindigkeit das Risiko von Kollisionen und erfordert von jedem Kapitän höchste Aufmerksamkeit und Wachsamkeit.
Allerdings könnte diese Lösung in Betracht gezogen werden, wenn die Kantengewichte nicht die Zeit, sondern das Geld berücksichtigen. In diesem Fall würde statt des schnellsten Weges der kostengünstige Weg gesucht werden. Jedoch ist es viel komplizierter, Geld als Gewichtungsfaktor zu betrachten, da die Kosten nicht nur die Treibstoffkosten, sondern auch Personalkosten, Reparaturkosten und andere Ausgaben umfassen.
Solange diese Lösung nicht in der Praxis umgesetzt wird, bleibt ihre Effizienz und Wirksamkeit unbewiesen. Trotz der Unanwendbarkeit dieser Idee lässt sich noch eine andere Idee daraus ableiten. Die zuvor abgelehnte Lösung basiert auf der Steigerung der Kosten durch die beschleunigte Geschwindigkeit. Daher könnte ein alternativer Ansatz darin bestehen, nicht zu beschleunigen, sondern nur zu verlangsamen. Das wird im nächsten Abschnitt behandelt.

7.3.1 Langsamer fahren

Eine Alternative zur beschleunigten Fahrweise könnte das Fahren mit einer geringeren Geschwindigkeit sein, was eine effiziente Lösung darstellen könnte. Hierbei bleibt der bereits gefundene Weg erhalten und es muss sogar kein neuer berechnet werden. Der Algorithmus 5 mit der Zeitkomplexität O(n) berechnet lediglich die neue Geschwindigkeit f, mit der das Schiff fahren soll. Dazu wird zunächst der Knoten v auf der bereits gefundenen Route gesucht, der als erster auf V_s trifft und vom Sturm betroffen ist. Durch diese Suche wird die Liste l = $\{e_1, e_2, .., e_n\}$ bestimmt, die alle Kanten beinhaltet, die den Weg von s nach v bilden. Zur Berechnung von f wird die Geschwindigkeit-Distanz-Zeit Formel (Abschnitt 2.4.3) verwendet, wobei unbedingt auf die Einheiten von Zeit und Geschwindigkeit geachtet werden muss und diese so konvertiert werden müssen, dass am Ende f in der Einheit "Knoten" angegeben wird, da diese Einheit im Seeverkehr gilt. Um den Distanz d zu berechnen, wird l durchlaufen und d wird sich aus der Summe der l bildenden Kanten ergeben, d.h. d = $w(e_1)$ +$w(e_2)$ + .. + $w(e_n)$. Danach wird die Zeit t_2 berechnet, wobei t_0 die Abfahrtszeit des Schiffes ist, t_1 die benötigte bzw. fehlende Zeit zum Überqueren des Weges von s nach v, sodass das Schiff an v ankommt, erst nachdem der Sturm vorüber ist und t_2 die neue Zeit, um die Strecke l mit der Entfernung d zurückzulegen.
Somit gibt der Algorithmus f an, mit der das Schiff von s nach v fahren sollte, um dem Sturm auszuweichen. Falls f nicht der zulässigen Mindestgeschwindigkeit f_m entspricht, bedeutet das, dass das Schiff viel zu langsam fährt, was für andere Verkehrsteilnehmer gefährlich sein kann. Deshalb wird das Schiff nach f_m fahren und die Zeitdifferenz, die durch das Beschleunigung auf f_m entsteht, wird an einem beliebigen Knoten vor dem Sturmgebiet gewisse Zeit τ gewartet werden. Daher wurde am Anfang τ mit 0 initialisiert.
Am Ende bedeutet dies, dass das Schiff auf der Strecke l mit der Geschwindigkeit f fahren soll, und wenn $\tau > 0$ ist, soll an einem beliebigen Knoten vor v gewartet werden.

Algorithmus 5 Neue Geschwindigkeit berechnen

```
1:  τ = 0
2:  d = 0
3:
4:  ∀ E(a, b) ∈ l:
5:      d + = w(a, b)
6:
7:  t1 = te − (t0 + d[v])
8:  t2 = t1 + d[v]
9:  f = d / t2
10:
11: If f < fm:
12:     f = fm
13:     τ = (d / f) + t2
14:
15: return f, l, τ
```

Da dieser Algorithmus die Geschwindigkeit nur bis zum ersten betroffenen Knoten v anpasst, ist es wichtig sicherzustellen, dass das Schiff auf der gesamten Strecke vor Stürmen geschützt ist. Dazu muss die Route so lange überprüft werden, bis sichergestellt ist, dass das Schiff an keinem weiteren Knoten in einen Sturm gerät. Dies kann mit Hilfe des Algorithmus 3 erfolgen. Anschließend wird dem Schiff mitgeteilt, an welchen Kanten es mit welcher Geschwindigkeit fahren soll und wie lange es ggf. warten muss.

Durch den selektiven Ansatz des Algorithmus, der nicht alle Knoten und den gesamten Graphen durchläuft, ist die Berechnung der neuen Geschwindigkeit extrem schnell. Vorrausgesetzt ist eine vorherige Berechnung des Pfades.

Zunächst wird diese Lösung an dem Testszenario ausprobiert, bei dem kein Pfad visualisiert wird, da der Knoten und die Route nicht geändert werden, sondern nur die Geschwindigkeit und die Zeit. Das Ergebnis besagt, dass sich das Schiff mit einer Geschwindigkeit von etwa 4,17 Knoten auf 8 Kanten fahren soll. Dies ergibt ein Gesamtgewicht von ungefähr 4619.17 Minuten ≈ 76.98 Stunden ≈ 3.2 Tage und erfordert keine Wartezeit. Und somit hat der Algorithmus gezeigt, dass er eine effektive Entscheidung treffen und eine geeignete Lösung für das Problem liefern kann.

7.4 Abfahrtszeit ändern

Der erste Lösungsansatz war darauf ausgerichtet, die Fahrtstrecke zu ändern, der zweite die Geschwindigkeit des Schiffs. Ein dritter Ansatz könnte die Abfahrtszeit als Maßstab betrachten und versuchen, durch eine Änderung der Abfahrtszeit eine bessere Lösung zu erzielen. Es könnten zwei Möglichkeiten in Betracht gezogen werden: entweder vorzeitig oder verpätet abfahren.

Das vorzeitige Abfahren ist im Wesentlichen ausgeschlossen. Das Schifffahren ist anders als das Autofahren. Man setzt sich ins Auto, startet den Motor und fährt los. Das Schifffahren ist komplizierter, unabhängig davon, um welche Art von Schiff es sich handelt. Ist es ein Passagierschiff, so wird die Abfahrtszeit in der Regel Monate oder Wochen vor der Reise bestätigt, so dass die Reise erst ab diesem Zeitpunkt erfolgen kann, da alle Passagiere wahrscheinlich ihre Berufstätigkeiten und Aktivitäten auf die Reise ausgerichtet haben und keine Änderung ihrer Pläne vorgesehen ist. Handelt es sich bei dem Schiff um ein Containerschiff, müssen entsprechende Vorbereitungen getroffen werden, das Schiff beladen und die Ladung sicher verpackt und an Bord gesichert werden, was viel Zeit in Anspruch nimmt. Je früher das Schiff abfährt, desto früher müsste alles erledigt werden, was meistens nicht gelingt.
Auch das verspätete Abfahren ist nicht angemessen, denn es ist wichtig, bei der Entscheidung für eine Verschiebung der Abfahrtszeit oder eine Verspätung der Abfahrt auch die Auswirkungen auf den Zeitplan der gesamten Reise zu berücksichtigen. Eine Verzögerung am Startknoten kann sich auf die Ankunftszeit am Zielknoten auswirken und möglicherweise zusätzliche Wartezeiten an anderen Knoten verursachen.

Insgesamt ist eine Änderung der Abfahrtszeit keine angemessene Idee, da sie das Problem nicht lösen, sondern sogar weitere Probleme verursachen könnte.

7.5 Warten

Das Prinzip des Wartens ist ein letzter Ansatz, der vorgeschlagen wird. Es besagt, dass ein Schiff an einem sicheren Ankerplatz verweilt, bis der Sturm abgeflaut ist, bevor es seine Fahrt fortsetzt. Es ist jedoch nicht ratsam, an einem zufälligen Knoten zu warten, da eine solche Einstellung sich als ungünstig erweisen kann. Um das Prinzip des Wartens effektiv umsetzen zu können, sollte das Schiff im Voraus wissen, wo und wie lange es warten soll. Idealerweise sollte es an dem Knoten verweilen, an dem die Wartezeit am geringsten ist. Zusätzlich ist zu beachten, dass nicht jeder Knoten für das Warten geeignet ist. Es ist vorzuziehen, an einem Knoten zu warten, der einen Hafen darstellt, da ein Aufenthalt an Land dem Schiff viele Möglichkeiten bieten kann, wie z. B. eine Durchführung notwendiger Wartungs- und Reparaturarbeiten und eine Aufnehme von Versorgungsgütern. Im Notfall, beispielsweise bei mehreren aufeinanderfolgenden Stürmen, suchen alle gefährdeten Schiffe Zuflucht in einem sicheren Hafen. Dies kann jedoch zu einer raschen Überfüllung aller Häfen in der Umgebung führen. In diesem Fall muss das Schiff an einem Zwischenpunkt in der Mitte des Meeres warten, da es keine bessere Alternative gibt. Jedoch darf nicht vergessen werden, dass jeder Hafen über eine begrenzte Kapazität verfügt. Sobald ein Hafen überfüllt ist, kann das Schiff nicht dorthin fahren und verweilen, da kein Platz mehr vorhanden ist.
In Kürze sollte das Schiff an dem geeigneten Knoten verweilen, an dem die Wartezeit am geringsten ist, vorzugsweise an einem Hafen, der nicht überfüllt ist. Obwohl die Kapazität des Hafens ein wichtiger Faktor bei dieser Entscheidung ist, wird in der Arbeit nicht berücksichtigt bzw. vorausgesetzt, dass alle Häfen freie Ankerplätze für das Schiff bieten. Und es wird untersucht, ob es in diesem Idealfall, in dem das Schiff in jedem Hafen Zuflucht nehmen kann, sinnvoll ist, ausschließlich nur an Häfen zu verweilen oder ob es besser wäre, an jedem beliebigen Knoten, auch wenn es sich um einen Zwischenpunkt handelt, zu warten.

Um zu untersuchen, ob es sinnvoll ist, an einem bestimmten Hafen zu warten, müssen zunächst die Wartezeiten berechnet werden, die von verschiedenen Faktoren abhängen. In diesem Zusammenhang werden drei Fälle betrachtet:

1. Wenn ein Schiff am Knoten A ankommt und zum Knoten B fahren möchte, wird zunächst geprüft, ob $B \in V_s$ ist. Falls das nicht der Fall ist, kann das Schiff ohne Risiko von A nach B fahren, ohne warten zu müssen. In diesem Fall beträgt die Wartezeit τ

$$\tau = 0$$

2. Wenn $B \in V_s$ ist, kann es sein, dass das Schiff an A warten muss. Wenn an A zum Zeitpunkt der Ankunft an A ein Sturm herrscht und das Schiff an A warten muss, wird auch die Kante (A, B) überprüft, um festzustellen, ob sie $\in E_s$ bzw. vom Sturm betroffen ist. Wenn die Kante betroffen ist, kann das Schiff diese Kante nicht passieren. Um τ zu berechnen, müssen verschiedene Zeitangaben berücksichtigt werden: t_e ist der Zeitpunkt, zu dem der Sturm vorüber ist, t_1 ist die Zeit, zu der das Schiff an A ankommt, sie ergibt sich aus der Summe der Abfahrtszeit des Schiffes t_0 und der Entfernung vom Startknoten s nach A

(d[A]). Die Wartezeit ist dann definiert als:

$$t_1 = t_0 + d[A]$$

$$\tau = t_e - t_1$$

3. Wenn B $\in$ V_s und die Kante (A, B) $\notin$ E_s, kann das Schiff diese Kante überqueren und die Zeit, die für die Überquerung benötigt wird, nutzen, um die Wartezeit zu reduzieren. Dann müssen für die Berechnung der Wartezeit die Zeitangaben t_2, t_1 und t_2 betrachtet werden, wobei t_2 die Zeit bezeichnet, zu der das Schiff am Knoten B ankommt, ohne warten zu müssen, wobei zu t_1 noch das Gewicht w(A, B) addiert wird. Die Wartezeit wird dann definiert als:

$$t_2 = t_1 + w(A, B)$$

$$\tau = t_e - t_2$$

Die Einfügung von τ an der geeigneten Stelle im Algorithmus ist von entscheidender Bedeutung für eine erfolgreiche Implementierung der vorgeschlagenen Lösung. Da sich diese Lösung auf den Dijkstra-Algorithmus bezieht, wird τ in den Dijkstra-Algorithmus eingebettet und zur berechneten Distanz addiert. In Abschnitt 2 2.3 wurde die Berechnung des Dijkstra-Algorithmus im Hinblick auf die Distanz Schritt für Schritt erläutert. Im Allgemeinen untersucht Dijkstra an einem ausgewählten Knoten k jeden Nachbarknoten n auf sein Gesamtgewicht W, das sich aus der d[k] + w(k, n) ergibt. Das entspricht der Ankunftszeit am Knoten k plus der Zeit, die benötigt wird, um die Kante (k, n) zu durchqueren. An dieser Stelle wird die berechnete τ hinzugefügt, um das neue W zu erhalten, das aus d[k] + w(k, n) + τ besteht. Anders ausgedrückt ergibt sich die Distanz vom Startknoten s bis zum Nachbarknoten n aus der Zeit, zu der der Knoten k erreicht wurde, plus der Zeit, die am Knoten k gewartet werden soll, plus der Zeit, die benötigt wird, um die Kante (k, n) zu überqueren.

Nach der Definition der Berechnungsmethode für die Wartezeit kann nun mit der Untersuchung der Knotenauswahl begonnen werden, an dem das Schiff auf seine Weiterfahrt warten soll. Die Präferenz liegt hierbei in der Auswahl von Knoten, die einen Hafen repräsentieren. Allerdings ist nicht unbedingt gesagt, dass dies eine bessere Option darstellt als das Warten an einem beliebigen Knoten, auch wenn dieser sich mitten auf dem Meer befindet. Um dies herauszufinden, werden zunächst zwei Algorithmen vorgestellt, einer, der das Warten nur an Häfen erlaubt, und einer, der das Warten an jedem beliebigen Knoten erlaubt.

7.5.1 Warten an jeden Knoten

Zuerst wird der Algorithmus 6 vorgestellt, die das Warten an jedem beliebigen Knoten oder auch an mehreren Knoten zulässt. Dieser Algorithmus ist auf dem Dijkstra-Algorithmus aufgebaut und hat die Zeitkomplexität $O(n^2)$. Im Vergleich zu Dijkstra wurden die folgenden Änderungen vorgenommen:

Zeile 4: Nach der Initialisierung, bei der jedem v $\in$ V ein d[v], p[v] und b[v] zugeordnet wird, wird eine Liste l_w initialisiert. l_w ist eine Warteliste, in der die Wartezeiten sowie die Wartekno-

ten gespeichert werden. Soll das Schiff z.B. 30 Minuten am Knoten v warten, bevor es zum Nachbarknoten n weiterfährt, wird dies der Warteliste als Liste = [v, n, 30] hinzugefügt. Soll das Schiff auch 20 Minuten an Knoten v warten, bevor es zu anderen Nachbarknoten n' weiterfährt, wird dies ebenfalls als neue Liste = [v, n', 20] in die Warteliste aufgenommen. Dies dient dazu, Fälle zu vermeiden, in denen der Knoten v mehrere Nachbarn hat und das Anfahren jedes Nachbarn zu unterschiedlichen Wartezeiten führt. In diesem Fall ist es wichtig, zwischen verschiedenen Knoten und Wartezeiten zu unterscheiden.

Zeile 12: Es wird der Tiefgang des Nachbarn T(n) geprüft, ob das Schiff einen gemessenen Tiefgang T besitzt, T(n) ≥ T, und ob es zum Nachbarknoten fahren darf.

Zeilen 13-45: In diesen Zeilen wird die Wartezeit τ berechnet. Für jeden Knoten n ∈ N(k), der nicht besucht wurde, wird zunächst überprüft, ob n ∈ V_s ist. Wenn das nicht der Fall ist, gibt es keinen Sturm an n und das Schiff kann ohne Wartezeit fahren, d.h. τ = 0. Wenn n ∈ V_s ist, werden t_1 und t_2 wie im vorherigen Abschnitt berechnet. Falls t_1 und t_2 entweder vor dem voraussichtlichen Start des Sturms t_s liegen oder nach dem Ende des Sturms t_e, d.h. das Schiff erreicht den Knoten k und n entweder bevor der Sturm beginnt oder nachdem der Sturm vorüber ist, dann besteht keine Gefahr und es ist keine Wartezeit erforderlich. Somit ist τ = 0.

Andernfalls, wenn das Schiff während des Sturms an n ankommt, wird die Wartezeit τ berechnet. Wie bereits erwähnt, wird diese Berechnung je nachdem, ob E(k, n) überquerbar ist oder nicht bzw. ob E(k, n) ∈ E_s ist, unterschiedlich sein. Sobald τ berechnet wurde, wird überprüft, ob k (der Knoten, an dem gewartet werden soll) ∈ V_s ist. Falls das der Fall ist, muss ein alternativer Knoten gesucht werden, da das Warten an einem Knoten, der später von einem Sturm beeinflusst wird, nicht erlaubt ist. Denn das könnte später das Schiff in Gefahr bringen. Um einen alternativen Knoten zu finden, werden die Vorgänger von k in Richtung n durchlaufen, bis der Knoten v_1 erreicht wird, der nicht in V_s liegt und dessen Vorgänger von v_2 (p[v_2]) v_1 ist. An v_1 wird dann in Richtung v_2 gewartet, was möglicherweise dazu führen kann, dass einige der bereits berechneten W nicht mehr gültig sind. Um dies zu vermeiden, wird eine Liste L erstellt, die Knoten enthält, die durch das Einfügen von τ beeinflusst werden können. Diese Liste wird so lange durchlaufen, bis sie leer ist. Zu Beginn enthält sie die Nachbarknoten v' von v_1, für die p[v'] = v_1 gilt, d.h. v_1 ist der Vorgänger von v'. Für jeden Nachbarknoten v' wird ebenfalls nach Knoten gesucht, die von v' aus geändert werden, und zu L hinzugefügt. Durch die Iteration werden die gefundenen v' auf ihren ursprünglichen Zustand zurückgesetzt und d[v'] auf unendlich und b[v'] auf falsch gesetzt, damit sie erneut durch den Algorithmus bearbeitet werden können. Sofern alle diese Knoten in L bearbeitet wurden, werden k und v_2 als nicht besucht markiert und d[v_2] wird aktualisiert, da auf v_1 gewartet wird und die Liste [v_1, v_2, τ], die besagt, dass von v_1 bis v_2 an Knoten v_1 für τ Minuten gewartet wird, zu l_w hinzugefügt. Danach wird die Schleife abgebrochen und der Algorithmus sucht erneut aus l, der Liste der nicht besuchten Knoten, einen Knoten k mit minimalem d[k].

Zeilen 46, 50: Es wird τ zum W addiert, wodurch W von τ beeinflusst wird. Falls gewartet werden soll, wird [k, n, τ] in Zeile 50 zur l_w hinzugefügt.

Algorithmus 6 Warten überall - Dijkstra

```
1: ∀ v ∈ V:
2:       d[v] = ∞; p[v] = null; b[v] = false;
3: d[s] = 0;
4: l_w = [ ];
5:
6: While l:
7:       k = v', where v' ∈ l and ∀ v ∈ l: d[v'] <= d[v]
8:       b[k] = true
9:       If k = z: return path
10:
11:      ∀ n ∈ N(k), where b[n] = false:
12:          If T(n) ≥ T:
13:              τ = 0
14:              If n ∈ V_s:
15:                  t_1 = t_0 + d[k]                          ▷ time arriving at k
16:                  t_2 = t_1 + w(k, n)                       ▷ time arriving at n
17:                  If (t_1, t_2 < t_s) or (t_1, t_2 > t_e): do nothing
18:                  Else If t_2 ∈ [t_s, t_e[:
19:                      If E(k, n) ∈ E_s: τ = t_e − t_1
20:                      Else: τ = t_e − t_2
21:
22:                      If τ > 0 and k ∈ V_s:                 ▷ k, n ∈ V_s
23:                          v_1 = none; v_2 = none; tmp = k;
24:                          While not v_1:
25:                              v_p = p[tmp]
26:                              If v_p ∉ V_s:
27:                                  v_1 = v_p;
28:                                  v_2 = tmp;
29:                              Else:
30:                                  tmp = v_p;
31:
32:                          L = [v' | v' ∈ N(v_1)], p[v'] = v_1]
33:                          While L:
34:                              l = []
35:                              ∀ v' ∈ L:
36:                                  add each v ∈ [v | v ∈ N(v')], p[v] = v'] to l
37:                                  If v' ≠ v_1 and v' ≠ v_2:
38:                                      d[v'] = ∞; b[v'] = false
39:                              L = l
40:                          b[k] = false;
41:                          d[v_2] + = τ;
42:                          b[v_2] = false
43:                          add [v_1, v_2, τ] to l_w
44:                          break
45:
46:              W = d[k] + w(k, n) + τ
47:              If W < d[n]:
48:                  d[n] = W
49:                  p[n] = k
50:                  If τ > 0: add [k, n, τ] to l_w
```

Dieser Ansatz wird ebenfalls an dem Testszenario angewendet, um zu überprüfen, ob er eine Lösung finden und einen kürzesten Pfad mit Informationen zur Wartezeit und zum Warteort ermitteln kann. In der nachstehenden Abbildung 25 ist der neu ermittelte Pfad zu sehen, der von den Sturmknoten und -kanten abweicht. Der Algorithmus hat einen neuen Weg gefunden, der über 51 Knoten verfügt und ein Gesamtgewicht von etwa 3900.43 Minuten ≈ 65 Stunden ≈ 2.7 Tage hat. Außerdem hat der Algorithmus mitgeteilt, dass das Shiff für 84 Minuten warten soll. Der schwarz markierte Knoten ist der Knoten, an dem gewartet werden soll. Es handelt sich nicht um einen Hafen, sondern um einen Zwischenpunkt.

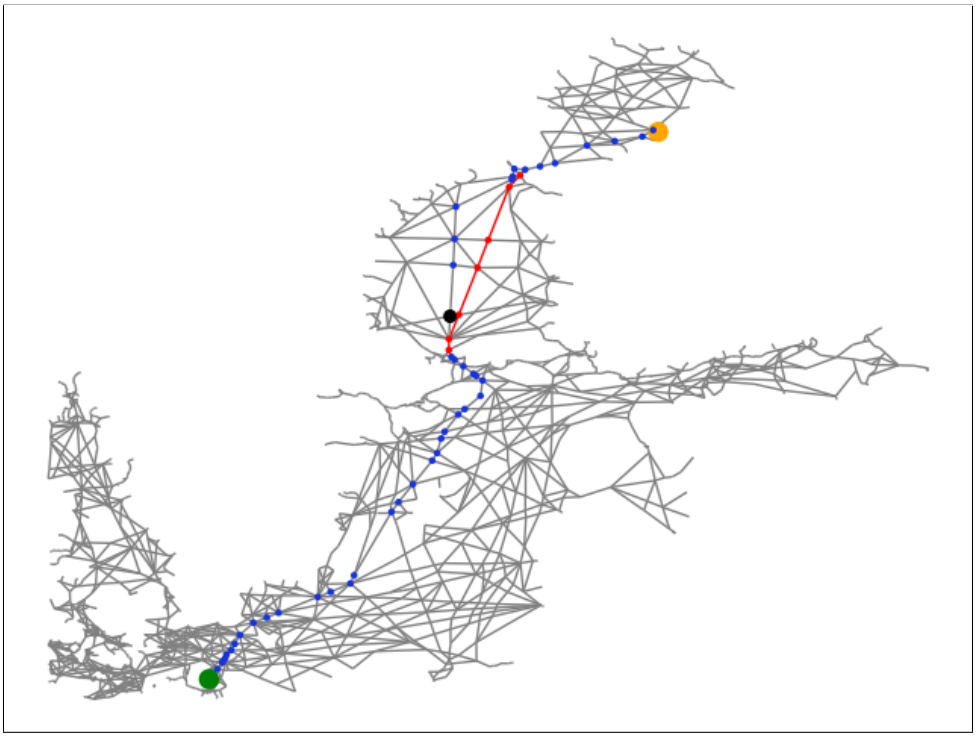


Abbildung 25: Warten überall zulässig

7.5.2 Warten nur an Häfen

Der andere Fall, der untersucht werden soll, ist, wenn das Warten nur an einem Hafen erlaubt ist. Dafür wird auch ein Algorithmus 7 vorgestellt, der eine ähnliche Struktur wie der gerade eben vorgestellte Algorithmus hat, der das Warten an jedem Knoten beschreibt. Er hat die Zeitkomplexität $O(n^2)$.

Bei der Initialisierung wird jedem Knoten v eine Warteliste l_w zugewiesen, anstatt eine Warteliste für alle Knoten. So speichert jeder Knoten für sich die möglichen Wartezeiten zu seinen Nachbarn. Anschließend wählt der Algorithmus aus der Liste l, den nicht besuchten Knoten in G, einen Knoten k mit minimalem Abstand zu s. Jeder Nachbar n der nicht besuchten Nachbarn von k wird abgearbeitet, gegebenenfalls wird sein d[n] aktualisiert. Diese Wartezeit τ wird nur berechnet, wenn $n \in V_s$ ist. Dies wird auf die gleiche Weise wie im vorherigen Algorithmus berechnet. Wenn τ einen negativen Wert hat, bedeutet dies, dass der Sturm bereits vorbei ist, dann muss nicht gewartet werden. Andernfalls wird gewartet, und wenn n kein Hafen ist, anders ausgedrückt $n \notin L_p$ Liste der Häfen, wird ein anderer Hafen v_1 gesucht, auf den gewartet wird.

Dies geschieht, indem den Vorgängern gefolgt wird, bis ein Port gefunden wird. Von v_1 bis n werden alle Knoten, die bearbeitet wurden, als nicht besucht markiert und ihre Anfangszustände werden neu geordnet. Beispielsweise, wenn das Schiff am Knoten v für 30 Minuten warten soll, bevor es zum Nachbarknoten n fährt, wird zuerst geprüft, ob v ein Hafen ist oder nicht. Ist es ein Hafen, müssen keine Auswirkungen vorgenommen werden. Ist es kein Hafen, wird der Vorgänger p[v] gesucht. p[v] wird ebenfalls geprüft, ob er ein Hafen ist oder nicht. Die Vorgänger werden solange geprüft, bis ein Vorgänger v_1 getroffen wird, der ein Hafen ist. v_1 repräsentiert dann den zuletzt entdeckten Hafen auf der Route und ist im schlimmsten Fall der Startknoten s. Der Knoten v_2, der bevor der Knoten v_1 entdeckt wurde, soll angemerkt werden. An v_1 wird dann 30 Minuten gewartet und der Abstand von k wird aktualisiert. Für jeden Knoten von v_1 bis n werden die Nachbarn und die Nachbarn von Nachbarn initialisiert, falls sie bearbeitet sind, d.h. ihre Initialzustände werden wiederhergestellt und wieder als nicht besucht markiert. Der Grund dafür ist, dass auf v_1 gewartet wird, was bedeutet, dass die Ankunft in k verzögert wird, was wiederum eine Änderung der Gewichte bedeutet. Deshalb müssen alle ab v_1 bearbeiteten Knoten noch einmal bearbeitet werden.

Algorithmus 7 Warten an Häfen - Dijkstra

```
∀ v ∈ V:
    d[v] = ∞; p[v] = null; b[v] = false; l_w[v] = [];
d[s] = 0;

While l:
    k = v', where v' ∈ l and ∀ v ∈ l: d[v'] <= d[v]
    b[k] = true
    If k = z: return path

    ∀ n ∈ N(k), where b[n] = false:
        If T(n) ≥ T:
            τ = 0
            If n ∈ V_s:
                t_1 = t_0 + d[k]                          ▷ time arriving at k
                t_2 = t_1 + w(k, n)                       ▷ time arriving at n
                If E(k, n) ∈ E_s: τ = t_e − t_1
                Else: τ = t_e − t_2

                If τ ≤ 0: τ = 0
                Else if τ > 0 and k ∉ l_p:
                    find v_1, v_2, L
                    ∀ v' ∈ L:
                        d[v'] = ∞; l_w[v'] = []; b[v'] = false;
                    add (v_1, v_2, τ) to l_w[v_1]
                    d[k] + = τ
                    b[k] = false
                    break

            W = d[k] + w(k, n) + τ
            If W < d[n]:
                d[n] = W; p[n] = k
                If τ > 0: add (k, n, τ) to l_w[k]
```

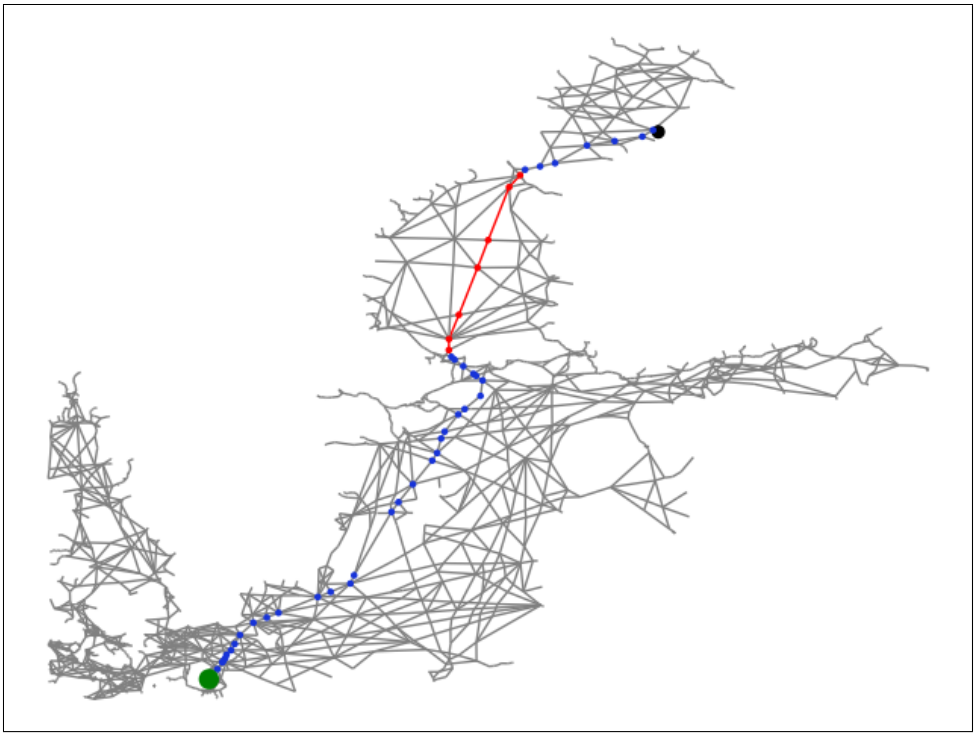


Abbildung 26: Warten an Häfen

In der obigen Abbildung 26 wurde der Algorithmus auch an dem Testszenario angewendet. Der schwarze Knoten ist dieses Mal der Startknoten, d. h. der zuletzt gefundene Hafen in der Suche war der Startknoten. Der Algorithmus hat allerdings keinen neuen Weg gefunden, sondern einen bereits bekannten Weg genommen, nämlich den ursprünglichen Weg (Abbildung 23). Die rot markierten Knoten und Kanten sind die Sturmknoten und -kanten. Sie stellen keine Gefahr mehr dar, da an dem Hafen/Startknoten für 994 Minuten gewartet wird. Der Weg hat 48 Knoten und das Gesamtgewicht beträgt ungefähr 4686.61 Minuten ≈ 78.11 Stunden ≈ 3.2 Tage. Dieser Weg dauert länger als der andere gefundene Weg (mit dem Prinzip des Wartens an jedem Knoten) und hat eine extrem lange Wartezeit, wahrscheinlich möchte kein Schiff so lange warten müssen. Allerdings kann aus diesem kleinen Testszenario keine abschließende Aussage getroffen werden. Beide Varianten haben jedoch gezeigt, dass sie eine Lösung bieten können, die das Schiff im Falle eines Unwetters absichert.

7.5.3 Unbeschränktes Warten (UW)

Das Prinzip des Wartens wurde hier nicht zum ersten Mal präsentiert und analysiert. Auch Orda und Rom haben sich damit beschäftigt (siehe Abschnitt 5.2.2), daher wird der Algorithmus von Orda und Rom implementiert und zum Vergleich mit dieser Methode herangezogen. Bei der unbeschränkten Wartezeit (UW - Algorithmus 2) wird die Wartezeit zwar berücksichtigt, aber nicht von diesem Algorithmus berechnet. Stattdessen ist eine Wartefunktion als Eingabe definiert. Daher wird dieser Algorithmus dahingehend modifiziert, dass er diese Wartezeit berechnen kann. Dies geschieht nach dem gleichen Prinzip wie oben beschrieben. Außerdem wird zwischen den beiden Varianten “Warten an einem Hafen“ und “Warten an einem beliebigen Knoten“ unterschieden.

Warten überall - UW

Zunächst wird die Variante des Wartens vorgestellt, bei der an jedem Knoten gewartet werden kann. Der Algorithmus ist eine Erweiterung des ursprünglichen UW-Algorithmus (siehe Abschnitt 5.2.2) mit der Zeitkomplexität von $O(n^2)$. Im Gegensatz zum ursprünglichen Algorithmus UW ordnet der Algorithmus 8 in Zeile 2 jedem Knoten v eine leere Warteliste w_v zu, in der die Wartezeiten und Warteknoten gespeichert werden. Und in Zeile 6 wird der Tiefgang des Knotens T(n) überprüft, so dass nur Schiffe mit einem Tiefgang T und T <= T(n) an n fahren dürfen. In den Zeilen 7-14 wird die Wartezeit τ berechnet, genau wie im Abschnitt 7.5 beschrieben.

Im Prinzip funktioniert der Algorithmus genauso wie Algorithmus 6, mit dem Unterschied der Datenstruktur. Hier arbeitet er mit einem Spanning-Tree. Der Spanning-Tree ist ein Teilgraph eines gegebenen Graphen, der alle Knoten und eine Teilmenge der Kanten enthält. Im Gegensatz zu Dijkstra, kann der Spanning-Tree verwendet werden, um die Konnektivität des Graphen zu analysieren und eine Baumstruktur zu erstellen, die den kürzesten Pfad zwischen zwei Knoten enthält. Der Startknoten wird dabei als Wurzel des Baumes betrachtet und jeder Knoten in diesem Baum hat einen Vaterknoten anstelle eines Vorgängerknotens.

Algorithmus 8 Warten überall - UW

```
1:  ∀ v ∈ V:
2:      Y_v = ∞; X_v = null; f_v = none; w_v = []
3:  X_s = t_s; f_s = none; j = s
4:
5:  ∀ n ∈ N(j), If X_n = null:
6:      If T(n) ≥ T:
7:          τ = 0
8:          If n ∈ V_s:
9:              t_1 = t_0 + Y_j                          ▷ time arriving at j
10:             t_2 = t_1 + w(j, n)                      ▷ time arriving at n
11:             If E(j, n) ∈ E_s: τ = t_e − t_1
12:             Else: τ = t_e − t_2
13:
14:             If τ ≤ 0: τ = 0
15:
16:         Y_n = min (Y_n, X_n + D_jn(X_j))
17:         If Y_n changed:
18:             f_n = j
19:             If τ > 0: add (j, n, τ) to w_j
20:
21: If j = z: return Path
22:
23: If ∀ v ∈ V: X_v ≠ null: Stop
24: Else:
25:     l = v', where X_v' = null and Y_v' ≤ Y_v ∀ v ∈ V, with X_v = null
26:     X_l = null
27:     j = l
28:     Go to line 5
```

Warten an Häfen - UW

Weiterhin wird diese Lösung modifiziert, um das Warten nur in Häfen zu ermöglichen. Wenn an einem Knoten gewartet werden soll und dieser Knoten kein Hafen ist, wird der letzte Hafen v_1 auf der Route gesucht, indem der Baum durchsucht und den Vätern gefolgt wird. Alle Kinder und Enkelkinder des gefundenen v_1 werden auf ihren Anfangszustand initialisiert. Der Algorithmus wiederholt die in den anderen Algorithmen verwendeten und erläuterten Verfahren und besitzt Laufzeitkomplixität von $O(n^2)$.

Algorithmus 9 Warten an Häfen - UW

```
1:  ∀ v ∈ V:
2:      Y_v = ∞; X_v = null; f_v = none; w_v = []
3:  X_s = t_s; f_s = none; j = s
4:
5:  ∀ n ∈ N(j), If X_n = null:
6:      If T(n) ≥ T:
7:          τ = 0
8:          If n ∈ V_s:
9:              t_1 = t_0 + Y_j                          ▷ time arriving at j
10:             t_2 = t_1 + w(j, n)                      ▷ time arriving at n
11:             If E(j, n) ∈ E_s: τ = t_e − t_1
12:             Else: τ = t_e − t_2
13:
14:             If τ ≤ 0: τ = 0
15:
16:             Else If τ > 0 and k ∉ l_p:
17:                 find v_1, v_2, L
18:                 ∀ v' ∈ L:
19:                     X_v' = none; Y_v' = ∞; w_v' = []
20:                 add (v_1, v_2, τ) to l_w[v_1]
21:                 X_k = none
22:                 Y_k + = τ
23:                 break
24:
25:         Y_n = min (Y_n, X_n + D_jn(X_j))
26:         If Y_n changed:
27:             f_n = j
28:             If τ > 0: add (j, n, τ) to w_j
29:
30: If j = z: return Path
31:
32: If ∀ v ∈ V: X_v ≠ null: Stop
33: Else:
34:     l = v', where X_v' = null and Y_v' ≤ Y_v ∀ v ∈ V, with X_v = null
35:     X_l = null
36:     j = l
37:     Go to line 5;
```

Nach der Ausführung des Testszenarios hat der Algorithmus "Warten überall - UW" eine neue Route identifiziert, die 51 Knoten und 3900,43 Minuten ≈ 65 Stunden ≈ 2,7 Tage durchläuft. Und es wurde eine Wartezeit von 84 Minuten ermittelt. Der Algorithmus "Warten an Häfen - UW" hat ebenfalls eine Lösung gefunden, einen Pfad, der 48 Knoten durchquert und eine Fahrtzeit von 4686,61 Minuten ≈ 78,11 Stunden ≈ 3,2 Tage mit einer Wartezeit von 994 Minuten am Startknoten. Die über UW ermittelten Wege sind in allen Aspekten identisch mit den von Dijkstra ermittelten Wegen 25, 26.

8 Auswertung und Evaluieren

Nachdem alle Algorithmen und Lösungsansätze vorgestellt wurden, ist es jetzt an der Zeit, sie miteinander zu vergleichen und zu testen. Dies ist ein wichtiger Schritt, da er jeden Algorithmus behandelt und seine Vor- und Nachteile aufzeigt. Am Ende sollten Aussagen über die verschiedenen Methoden getroffen werden können. Es ist wichtig, dass die Vergleiche und Tests sorgfältig und objektiv durchgeführt werden, um zu vermeiden, dass voreilige Schlüsse gezogen werden. Dabei sollten verschiedene Aspekte berücksichtigt werden, wie beispielsweise die Laufzeit, die Ergebnisse, die Genauigkeit und die Robustheit gegenüber Fehlern. Es ist zu erwähnen, dass das Hauptziel nicht darin besteht, den effizientesten Algorithmus zu finden, sondern einen Algorithmus zu finden, der einen sicheren Weg und Transport für jedes Schiff gewährleisten kann und der den kürzesten Weg zum Ziel ermitteln kann. Daher werden alle Algorithmen unter denselben Bedingungen verglichen. Alle Algorithmen werden in Python implementiert, mit den gleichen Paketen und ohne spezielle Optimierungen. Die Bewertung der Algorithmen basiert auf der Leistung unter denselben Bedingungen. Dazu wird zuerst in Abschnitt 8.1 erläutert, ob die Ansätze angemessen implementiert sind und wie sicher der ermittelte Pfad ist. Anschließend wird der Vergleich in mehrere Abschnitte unterteilt, ein Vergleich zwischen der Dijkstra-Algorithmus und der UW-Algorithmus (Abschnitt 8.2), ein Vergleich zwischen das "Warten an jedem Knoten" und "Warten an Häfen" (Abschnitt 8.3), ein Vergleich zwischen das "Warten an jedem Knoten", "Neuen Pfad suchen" und "Verlangsamen" (Abschnitt 8.4) und ein Vergleich zwischen das "Warten an jedem Knoten" und das "Neuen Pfad suchen" (Abschnitt 8.5). Zum Schluss wird in Abschnitt 8.6 ein abschließender Vergleich aller Algorithmen vorgenommen.

8.1 Spezielle Untersuchungen

Bevor der Vergleich der Algorithmen erfolgt, muss sichergestellt werden, dass die implementierten Algorithmen angemessen und ordnungsgemäß sind und dass der ermittelte Pfad für das Schiff gefahrlos ist. Zu diesem Zweck wurden verschiedene Tests vorgenommen:

- Ein Test des ursprünglichen Dijkstra-Algorithmus. Da der Dijkstra-Algorithmus der Kern dieser Arbeit ist, und ein kleiner Fehler zu falschen Ergebnissen in allen vorgestellten Lösungsansätzen führen kann, muss sichergestellt werden, dass er korrekt implementiert wurde. Zu diesem Zweck wurde eine vordefinierte Funktion von NetworkX (networkx. shortestpath(Graph, Source, Target, Weight, Method='dijkstra')) zum Vergleich herangezogen, und der implementierte Dijkstra-Algorithmus hat die gleichen Ergebnisse wie diese vordefinierte Funktion geliefert.

- Ein Test des Tiefgangswerts. Bei jedem Ansatz wird der Tiefgang berücksichtigt, da ein Schiff mit einem Tiefgang, der größer ist als der des Hafens, nicht einfahren darf. Zu diesem Zweck wurden alle gefundenen Pfade durchlaufen und jeder Knoten darin daraufhin geprüft, ob das Schiff tatsächlich einen Tiefgang besitzt, mit dem es an diesem Knoten einfahren darf. Dieser Test wurde erfolgreich abgeschlossen.

- Ein Test der Sicherheit des Schiffes. Das Hauptziel dieser Arbeit ist es, Stürme zu vermeiden und Verluste zu minimieren. Die Algorithmen suchen eine Lösung und teilen sie mit, jedoch wurde bisher nicht überprüft, ob die Gefahr eines Sturms beseitigt wurde. Zu diesem Zweck wurde jeder ermittelte Pfad durchlaufen und für jeden Knoten die Ankunftszeit des Schiffes an diesem Knoten berechnet. Anschließend wurde geprüft, ob dies während eines Sturms geschieht oder nicht. Auch dieser Test wurde erfolgreich abgeschlossen, und alle ermittelten Pfade schützen das Schiff und seine Crew vor Gefahren.

- Ein Test des kürzesten Weges. Die Algorithmen suchen einen neuen Pfad bis zum Zielknoten, aber es kann einen anderen Weg geben, der noch nicht entdeckt wurde und der sogar kürzer ist als der ermittelte Weg. Zu diesem Zweck wurde der Dijkstra-Algorithmus getestet, ohne die Schleife abzubrechen. Wenn der Zielknoten erreicht wird, wird der neue Pfad gespeichert und der Algorithmus setzt fort. Wenn er erneut auf den Zielknoten stößt, bedeutet dies, dass er einen anderen Weg gefunden hat, der ebenfalls gespeichert wird. Am Ende werden alle gefundenen Pfade verglichen, und der kürzeste Weg wird ermittelt. In diesem Test hat der Algorithmus jedoch bei allen zufälligen Testversuchen nur einen einzelnen Pfad gefunden.

Nachdem diese Tests erfolgreich abgeschlossen wurden, werden im Folgenden die Vergleiche der Algorithmen ausgeführt und erläutert.

8.2 UW vs. Dijkstra

In der Arbeit wurde das Warten-Prinzip auf den originalen Dijkstra-Algorithmus sowie auf den UW-Algorithmus von Orda und Rom angewendet und in beiden Fällen untersucht, ob das Warten nur an Häfen oder überall berücksichtigt werden soll. Ein Vergleich zwischen den beiden Ansätzen wird durchgeführt, um festzustellen, welche Gemeinsamkeiten und Unterschiede sie aufweisen.
Zu diesem Zweck wurden in den Abbildungen 27 und 28 jeweils die Gesamtwartezeit, die Gesamtfahrzeit und die Ausführungszeit berücksichtigt und für 15 zufällige Schiffe berechnet und verglichen. Jedes Schiff hat zufällig einen Start- und Zielpunkt sowie einen zufälligen Tiefgangswert erhalten. Alle Schiffe sollten durch Unwetter segeln und dafür wurden die Algorithmen eingesetzt, um eine Lösung zu finden. Es wurde darauf geachtet, dass jedes Schiff während seiner Fahrt einem zufälligen Sturm begegnet hat, wobei Start- und Endzeitstempel des Sturms ebenfalls zufällig gewählt wurden. Dabei wurde sichergestellt, dass ein Teil des Schiffsweges von dem Sturm beeinflusst ist. Die Verwendung von Zufälligkeit ist ein wichtiger Faktor, da dadurch viele Fälle getestet werden, die man nicht vorhersehen kann und die Abstraktheit und Vielfalt der Ergebnisse erhöhen.

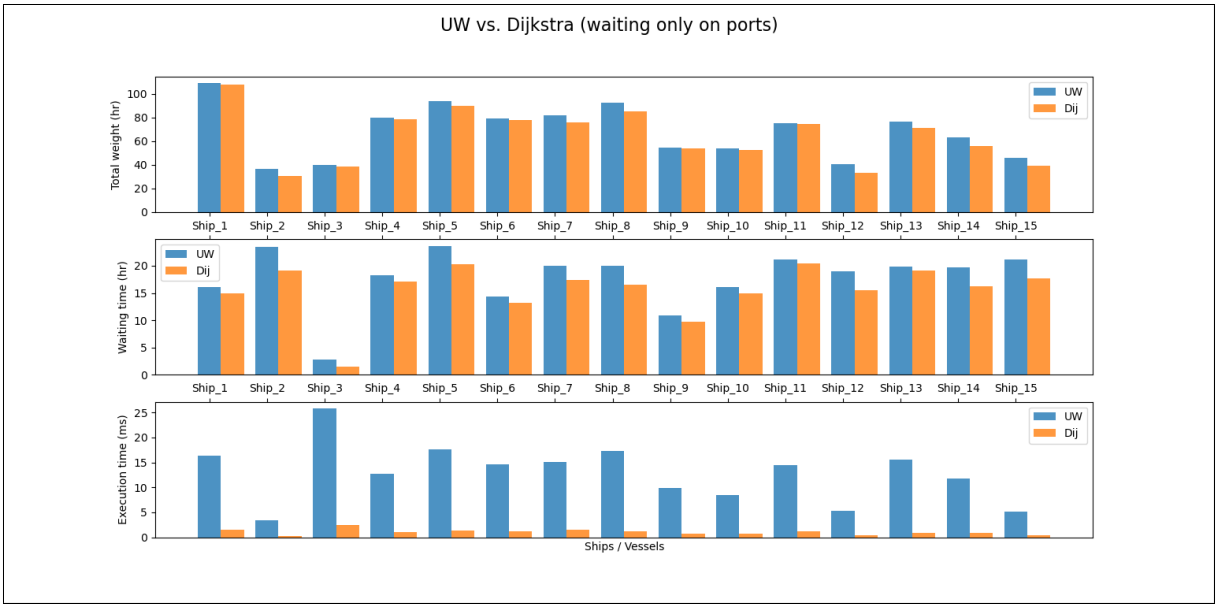


Abbildung 27: Warten an Häfen - Dijkstra vs. UW

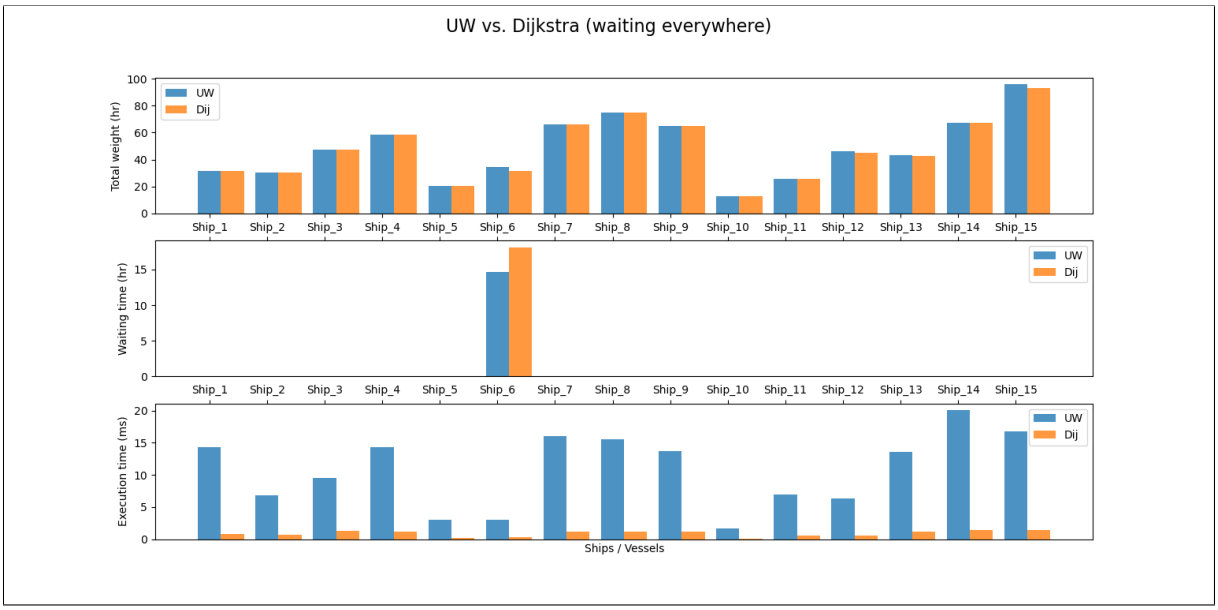


Abbildung 28: Warten überall - Dijkstra vs. UW

Beide Abbildungen zeigen, dass der Dijkstra-Algorithmus und der UW-Algorithmus ähnliche Ergebnisse hinsichtlich der Gesamtfahrzeit liefern. Die Unterschiede in den Wartezeiten sind relativ gering, jedoch hat der Dijkstra-Algorithmus in Abbildung 27 insbesondere bei "Warten an Häfen" schnellere Wege und kürzere Wartezeiten ermittelt. Es gibt jedoch einen deutlichen Unterschied in der Ausführungszeit: Der Dijkstra-Algorithmus hat in beiden Abbildungen gezeigt, dass er schneller ausgeführt wird und eine geringere Laufzeit aufweist. Dieser Unterschied scheint auf die verwendeten Datenstrukturen zurückzuführen zu sein. Beide Algorithmen verwenden eine Prioritätsschlange, aber während der UW-Algorithmus einen Spanning-Tree des Graphen bildet und auf einer Baum-Suche basiert, arbeitet der Dijkstra-Algorithmus direkt auf den Knoten des Graphen. Dies könnte der Grund dafür sein, dass der Dijkstra-Algorithmus effizienter wirkt.

Es ist jedoch zu beachten, dass der implementierte und modifizierte Dijkstra-Algorithmus der originale Dijkstra ist. Wenn beispielsweise der bidirektionale Dijkstra verwendet wird, kann der Code doppelt so schnell ausgeführt werden. Dennoch kann keine allgemeine Aussage zur Ausführungszeit getroffen werden, da dies je nach Implementierung variieren kann. Der Vergleich wurde dennoch durchgeführt, um zu sehen, wie die Algorithmen in diesem Zustand arbeiten.

8.3 Warten an Häfen vs. Warten überall

Im Kontext des Wartens können nicht nur verschiedene Algorithmen untersucht werden, sondern auch die beiden Varianten, bei denen das Schiff entweder an jedem beliebigen Knoten oder nur an Häfen warten darf. Es ist allgemein bekannt, dass das Warten an Häfen bevorzugt wird, aber es ist nicht ausreichend, um zu behaupten, dass das Warten an Häfen die bessere Variante ist. Deshalb werden im Folgenden 15 zufällige Schiffe mit zufälligen Stürmen generiert und auf Gesamtfahrtzeit, Ausführungszeit und insbesondere auf die Gesamtwartezeit verglichen. Um diesen Vergleich durchzuführen, werden sowohl der Dijkstra-Algorithmus als auch der UW-Algorithmus wie im vorherigen Abschnitt verwendet, obwohl UW dort ein schlechteres Verhalten als Dijkstra gezeigt hat. Hierbei wird für einen tieferen Einblick der Dijkstra-Algorithmus mit 15 zufälligen Schiffen und Stürmen und der UW-Algorithmus mit weiteren 15 zufälligen Schiffen und Stürmen getestet.

Die beiden Abbildungen 29 und 30, die beide Algorithmen wiedergeben, kommen zu dem gleichen Ergebnis, nämlich dass “Warten an Häfen“ zu einer längeren Fahrstrecke, einer extrem längeren Wartezeit und einer ähnlichen Laufzeit führt, im Vergleich zu “Warten an jedem Knoten“.

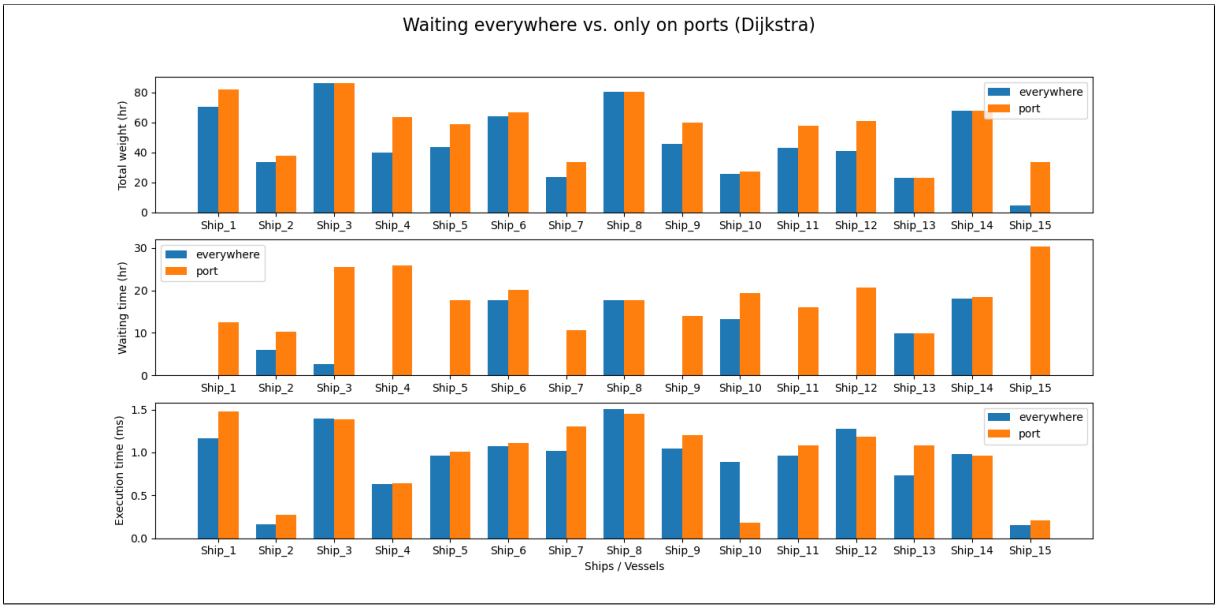


Abbildung 29: Dijkstra - überall vs. an Häfen warten

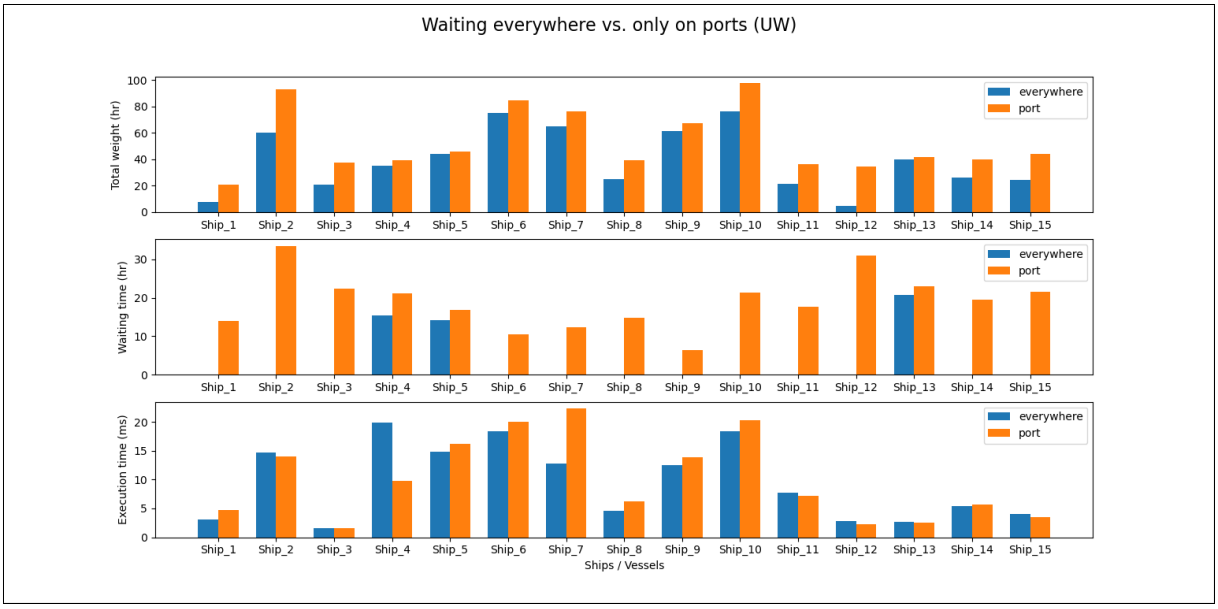
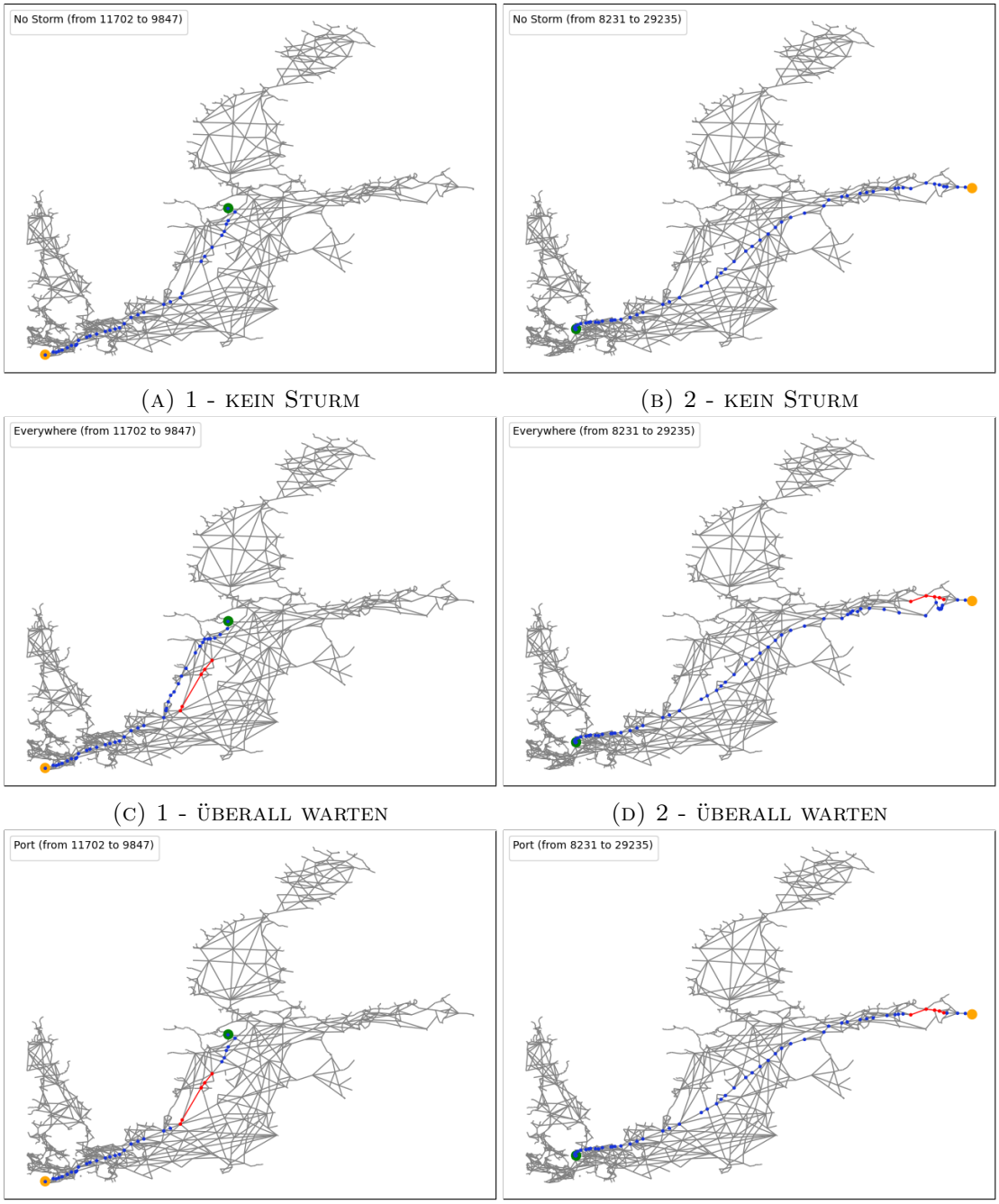


Abbildung 30: UW - überall vs. an Häfen warten

Auf solche Aussagen zu kommen, ist ziemlich verwirrend. Warum sollte das Warten an einem Hafen extrem lange Wartezeit erfordern im Vergleich zum Warten an jedem Knoten? Als Versuch, eine Antwort auf diese Frage zu finden, werden weitere zufällige Schiffe und Stürme erzeugt und die Algorithmen erneut getestet. Dieses Mal werden die Ergebnisse nicht in einem Diagramm dargestellt, sondern visualisiert, mit der Hoffnung, etwas zu erkennen, was in einem Diagramm nicht sichtbar ist.

Tatsächlich haben diese Plots weitere Erkenntnisse geliefert. Aus den vielen Plots wurden zwei Fahrten ausgewählt, die in Abbildung 31 dargestellt sind. Jede Fahrt wurde in drei Plots unterteilt, die jeweils eine Situation zeigen. Für die erste Fahrt ist in Abbildung 31a der Weg ohne Gefahr von Unwetter gezeichnet, in Abbildung 31c der Weg mit einem Sturm und Warten an jedem Knoten und in Abbildung 31e der Weg mit einem Sturm und Warten an einem Hafen. Das Gleiche gilt für die zweite Fahrt, die in den Abbildungen 31b, 31d und 31f dargestellt ist. Es ist erkennbar, dass unter der Variante des “Wartens an jedem Knoten“ in beiden Fahrten und in den meisten Fällen ein neuer Pfad ermittelt wird, der sich vom Originalpfad, der nicht vom Sturm betroffen ist, unterscheidet. Auf der anderen Seite wird unter der anderen Variante des “Wartens an einem Hafen“ immer wieder der gleiche Originalpfad ermittelt.

Das ist der Grund. Das “Warten an jedem Knoten“ berechnet die möglichen Wartezeiten und sucht sich den kürzesten Weg mit der minimalen Wartezeit. Im Gegensatz dazu geht das “Warten an Häfen“ zum letzten Hafen und wartet dort. Dadurch entfällt beim Ankommen an den nachfolgenden Knoten das Warten, wodurch dieser Unterschied in den Wartezeiten nicht sichtbar und somit auch nicht berücksichtigt wird.



(a) 1 - kein Sturm (b) 2 - kein Sturm

(c) 1 - überall warten (d) 2 - überall warten

(e) 1 - an Hafen warten (f) 2 - an Hafen warten

Abbildung 31: Pfadauswahl Vergleich

8.4 Warten vs. Neuer Pfad vs. Verlangsamen

Insofern wurde gezeigt, dass das "Warten an jedem Knoten" sinnvoller Ergebnisse als das "Warten nur an Häfen" liefert. Allerdings wurden diese Ansätze bisher nicht mit anderen verglichen. Dies wird im folgenden Abschnitt geschehen, indem die drei Ansätze verglichen werden: das "Warten an jedem Knoten", das Ignorieren von Sturmknoten und -kanten und "Neuen Pfad suchen" sowie das Nichtsuchen eines neuen Pfads und das "Verlangsamen" auf der Route. Dazu werden 15 zufällige Schiffe und Stürme verwendet und zwei Aspekte betrachtet und untersucht: die gesamte Fahrtzeit sowie die Ausführungszeit.

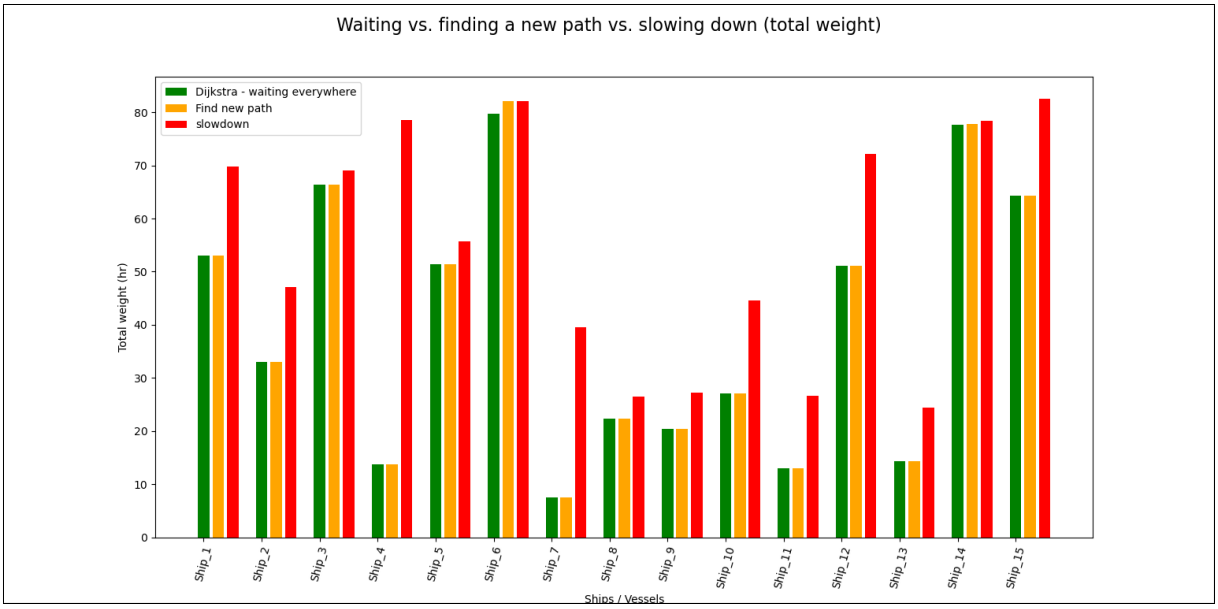


Abbildung 32: Warten vs. neuer Pfad vs. Verlangsamen - Totalgewicht

Abbildung 33: Warten vs. neuer Pfad vs. Verlangsamen - Ausführungszeit

In der obigen Abbildung 32 ist zu beobachten, dass sowohl die Variante “Neuen Pfad suchen“ als auch “Warten an jedem Knoten“ die gleiche Fahrtzeit ergeben. Im Gegensatz dazu ergibt die rote Linie, die für das “Verlangsamen“ steht, eine deutlich längere Fahrtzeit. Die andere Abbildung 33 illustriert die Laufzeit der verschiedenen Varianten. Hierbei hat die Variante des “Verlangsamen“ eine äußerst kurze Laufzeit erreicht. Dies kommt daher, dass dieser Algorithmus den Graphen nicht durchläuft und keinen neuen Pfad sucht, sondern nur die neue Geschwindigkeit berechnet. Obwohl diese Variante einen interessanten Rekord hält, liegt sie auf dem dritten Platz. Eine schnelle Ausführungszeit ist für die Methode signifikant, sodass die Methode den Pfad für hunderte von Schiffen berechnen soll und je schneller dies erfolgt, desto bessere Leistung wird erbracht, jedoch können verschiedene Implementierungen die Laufzeit auswirken und eine kürzere Fahrtstrecke und weniger Zeit auf Meer haben in dieser Arbeit einen höheren Wert.

8.5 Warten vs. Neuer Pfad

Da in den letzten Tests keine eindeutigen Unterschiede zwischen den Ansätzen “Warten an jedem Knoten“ und “Neuen Pfad suchen“ gezeigt wurden, sollen diese nun an einer größeren Anzahl von zufälligen Schiffen und Stürmen untersucht werden. In der Abbildung 34 kann nur bei zwei einzelnen Schiffen eine Abweichung festgestellt werden, die jedoch keinen signifikanten Unterschied zwischen den beiden Varianten darstellt. Auch in der Abbildung 35, die die Laufzeit beschreibt, ist die Situation unverändert geblieben. Die Steigerung der Anzahl der Testschiffe hatte keinen Einfluss, daher sollen weitere Tests mit anderen zufälligen Schiffen durchgeführt werden, um diese zu visualisieren.

Abbildung 34: Warten vs. neuer Pfad - Totalgewicht

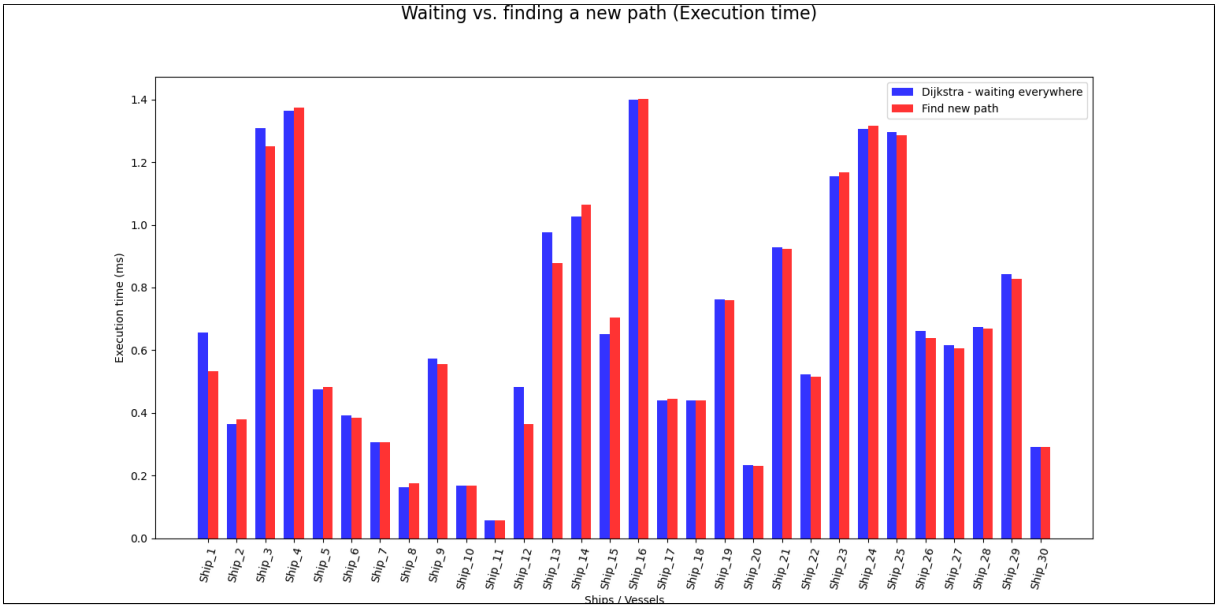
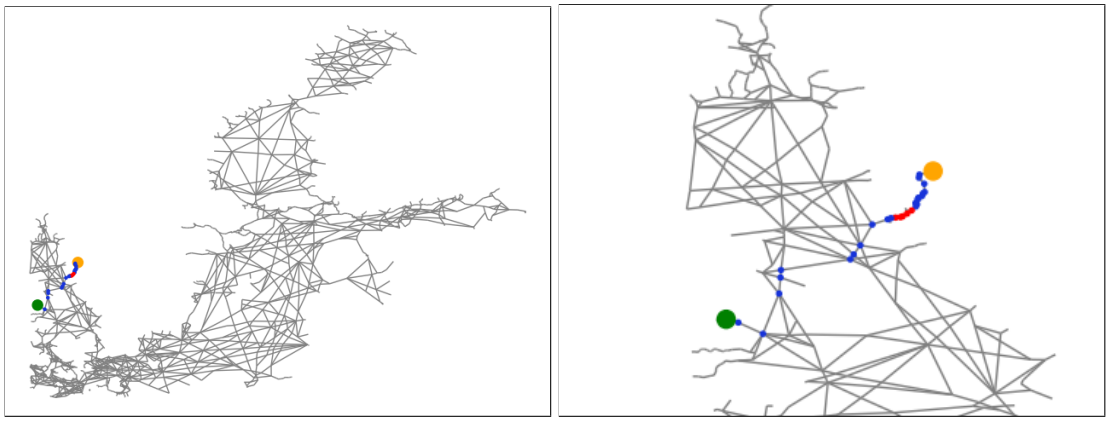


Abbildung 35: Warten vs. neuer Pfad - Ausführungszeit

Durch weitere Tests und Visualisierungen wurden neue Erkenntnisse gewonnen. Der Ansatz, “Neuen Pfad suchen“ zu suchen, hat einen Schwachpunkt. Obwohl er versucht, dem Sturm auszuweichen und einen neuen Weg zu finden, gibt es einige Fälle, in denen es keinen alternativen Weg gibt und er keinen Weg finden kann. Abbildung 36a zeigt ein solches Beispiel und die danebenstehende Abbildung 36b veranschaulicht dasselbe Beispiel, nur vergrößert, um es besser erkennbar zu machen. Wie zu bemerken ist, gibt es keinen alternativen Weg für das Schiff, und der Ansatz, einen neuen Weg zu finden, hat keine Lösung gefunden. Auf der anderen Seite hat der Ansatz “Warten an jedem Knoten“ funktioniert und eine Lösung gefunden, um das Schiff nicht in den Sturm zu führen.



(a) Neuer Pfad - Kein Pfad gefunden (b) Kein Pfad gefunden - zoom

8.6 Vollständiger Vergleich

Ein weiterer Vergleich könnte durchgeführt werden zwischen allen Arten der Lösungsansätze im Vergleich zum originalen Dijkstra-Algorithmus, der ohne Sturm berechnet wird. Dies ist in den folgenden Abbildungen 37 und 38 dargestellt. Keiner der vorgeschlagenen Ansätze hat einen kürzeren Weg als der originale Dijkstra-Algorithmus gefunden. Zwei Ansätze haben jedoch sehr ähnliche Werte wie der originale Algorithmus ermittelt, nämlich der Ansatz “Warten an jedem Knoten“ und der Ansatz “Neuen Weg suchen“. Die anderen Ansätze haben längere Fahrtzeiten aufgewiesen.
Betrachtet man die Ausführbarkeit, ist der Ansatz “Verlangsamen“ der schnellste, wie erwartet. Danach kommen die Ansätze “Warten an jedem Knoten“ und “Neuen Weg suchen“, und der langsamste ist die Variante “Warten an Häfen“. Dies bestätigt erneut alle Ergebnisse, die bei allen letzten Tests erhalten wurden.

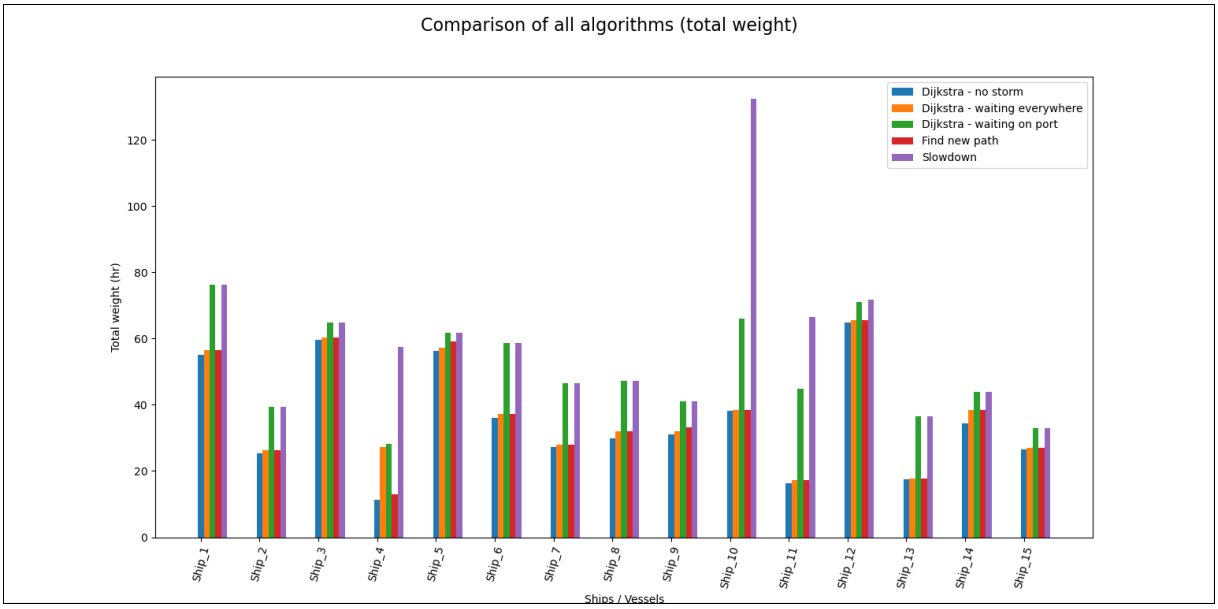


Abbildung 37: Vergleich aller Algorithmen - Totalgewicht

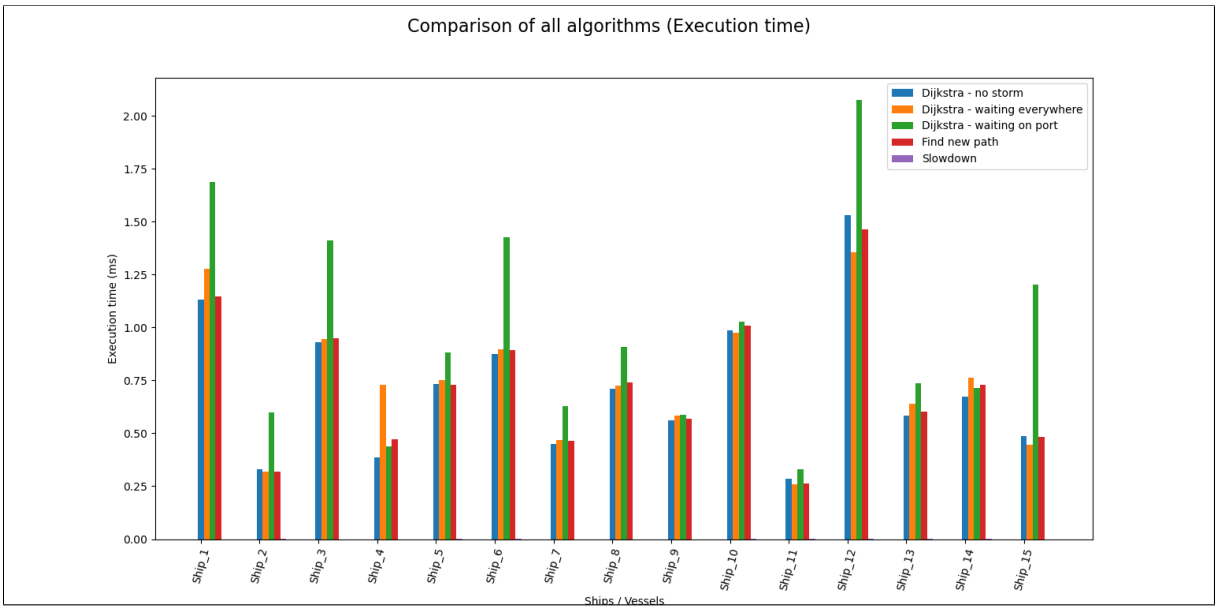


Abbildung 38: Vergleich aller Algorithmen - Ausführungszeit

Wie bereits erwähnt kann sich die Ausführungszeit je nach Programmiersprache und Implementierung ändern. Daher ist es wichtig, die Zeitkomplexität der verschiedenen Algorithmen zu vergleichen. Der Algorithmus 5 hat die geringste Zeitkomplexität von O(n), während alle anderen Algorithmen eine Zeitkomplexität von $O(n^2)$ aufweisen. Dies wurde auch in mehreren Abbildungen deutlich, da der Algorithmus "Verlangsamenëxtrem kurze Laufzeiten gezeigt hat.

9 Fazit und Ausblick

Die vorliegende Arbeit beschäftigt sich mit der Suche nach dem kürzesten Pfad im maritimen Kontext. Dabei wird ein spezifischer Fall des Problems betrachtet, nämlich das Single-Pair Shortest Path Problem, das beschreibt, die Suche nach dem kürzesten Pfad zwischen einem gegebenen Start- und Zielknoten. Allerdings wird dies nicht in einem konstanten, statischen Graphen untersucht, sondern in einem dynamischen Graphen, der die Berücksichtigung von Wetter als Auslöser für dynamische Änderungen beinhaltet.

Um das Ziel, eine Lösung des Single-Pair Shortest Path Problems unter dynamische Aspekte im maritimen Kontext, zu erreichen, wurden zunächst relevante Literaturquellen gesichtet und ausgewertet. Dabei wurde deutlich, dass das Problem bereits seit langer Zeit existiert und es in diesem Bereich bereits eine Vielzahl von Forschungsarbeiten gibt. Der Fokus lag dabei auf Arbeiten, die das Problem unter dynamischen Aspekten betrachtet haben. Am Ende der Literaturrecherche wurde ein Überblick darüber gewonnen, wie andere das Problem behandelt haben, und es wurde beschlossen, den unidirektionalen Dijkstra-Algorithmus in der Arbeit weiter einzusetzen. Außerdem wurde eine Arbeit von Orda und Rom entdeckt, in dem eine Lösung mit dem Prinzip des Wartens für das betrachtete Problem angeboten wird. Dieses wurde im weiteren Verlauf zur Vergleichszwecken verwendet.

Zur Lösung des Problems wurden fünf verschiedene Lösungsansätze vorgeschlagen. Zwei davon, “schneller fahren“ und “vorzeitig abfahren“, wurden aufgrund höherer Kosten bzw. mangelnder praktikabler Umsetzbarkeit abgelehnt. Daraus haben sich drei einsetzbare Lösungen basierend auf dem Dijkstra-Algorithmus ergeben:

- Neuen Weg suchen: Alle Knoten und Kanten, die von den dynamischen Veränderungen betroffen sind, werden ignoriert. Es wird ein neuer Pfad gesucht, der sich nicht mit diesen Knoten und Kanten kreuzt. Diese Lösung hat gezeigt, dass sie einen kürzesten Weg ermitteln kann, ohne auf das Unwetter zu stoßen. Der Algorithmus hat jedoch den Nachteil, dass er keinen Pfad bestimmen oder eine Lösung für Fahrten finden kann, die nur einen Weg zum Ziel haben oder die den Sturm auf allen möglichen Wegen durchqueren müssen.

- Langsamer fahren: Dieser Ansatz zielt darauf ab, keinen neuen kürzesten Weg zu finden, sondern die Geschwindigkeit des Schiffes anzupassen, um bei Ankunft im Unwettergebiet zu vermeiden, dass das Schiff direkt auf den Sturm trifft. Stattdessen wird darauf geachtet, dass der Sturm bereits vorüber ist, bevor das Schiff das betroffene Gebiet erreicht. Dies ermöglicht es dem Schiff, mit der erforderlichen Geschwindigkeit zu fahren, um den Zielpunkt zu erreichen. Da dieser Ansatz keinen neuen Pfad sucht und die Knoten und Kanten des Graphen nicht durchlaufen werden müssen, ist er äußerst schnell. Allerdings dauert die Fahrt in diesem Fall im Vergleich zu den anderen Varianten deutlich länger.

- Warten: Das Prinzip des Wartens wurde in zwei Varianten unterteilt: das Warten an jedem beliebigen Knoten und das Warten nur an Häfen. Für beide Varianten wurde der Dijkstra-Algorithmus sowie die Methode UW, die Orda und Rom vorgeschlagen haben, modifiziert, um das Warten nur an den vordefinierten Knoten zu ermöglichen. Anschließend wurden

Dijkstra und UW miteinander verglichen. Beide Ansätze haben die gleichen Ergebnisse erzielt, aber der Dijkstra-Algorithmus hat eine schnellere Ausführungszeit gezeigt.

- Warten an jedem Knoten: Zulässig, ist an einem oder mehreren Knotenpunkten zu warten, auch wenn der Knoten einen Zwichenpunkt auf dem Meer darstellt. Das konnte erfolgreich einen neuen pfad ermittelen und ihr weg hat in meisten Fällen ungefähr die gleiche Länge wie der Weg der nach der Ansatz "Neuen Weg suchen" gefunden wird.
- Warten an Häfen: Zulässig, ist nur an Häfen zu warten. Diese Variante ist vorzuziehen, da ein Aufenthalt an Land jedem Schiff viele Möglichkeiten bieten kann. Sie konnte ebenfalls einen kürzesten Weg finden, erfordert jedoch im Vergleich zur ersten Variante längere Wartezeiten und führt somit zu einer längeren Fahrtzeit. Der Grund dafür ist, dass diese Variante meistens keinen neuen Pfad bestimmt, sondern auf dem ursprünglichen Pfad bleibt.

Zusammenfassend lässt sich sagen, dass alle drei vorgestellten Lösungsansätze das angestrebte Ziel erreichen und einen sicheren und schnellen Transport von Schiffen gewährleisten. Der Ansatz "Warten an jedem Knoten" hat sich jedoch gut bewährt und stellt die beste Lösung für Schiffe dar, die sicher und schnell navigieren möchten, ohne dabei Gefahr zu laufen, aufgrund von Unwettern oder anderen unvorhergesehenen Ereignissen in Schwierigkeiten zu geraten. Dabei schützt der Ansatz das Schiff, die Ladung sowie die Crew-Mitglieder bzw. Passagiere. Darüber hinaus kann er zu jedem Start- und Zielknoten, sofern mindestens ein Weg existiert, den kürzesten Weg finden. Dadurch spart er nicht nur Zeit, sondern auch Kosten.
Mehr Flexibilität könnte erreicht werden, indem der Algorithmus nicht nur eine einzelne Wartezeit und einen Warteknoten zurückgibt, sondern mehrere mögliche Wartezeiten und Warteknoten vorschlägt. Dadurch könnte das Schiff selbst entscheiden, an welchen Knoten es anlegt.

Der Ansatz "Warten an jedem Knoten" könnte weiter optimiert werden, um eine höhere Genauigkeit zu erreichen. Dazu könnte die Haversinus-Formel durch eine andere Formel ersetzt werden. Diese Formel wurde verwendet, um die Entfernung zwischen den Knoten auf der Erde zu berechnen. Sie ist aber nicht immer genau, insbesondere bei sehr kurzen Entfernungen oder in der Nähe der Pole. Eine mögliche gute Alternative ist die Vincenty-Formel.

Eine mögliche Weiterentwicklung könnte in mehreren Phasen erfolgen. Zunächst sollten die Hafenkapazitäten berücksichtigt werden, da angenommen wurde, dass das Schiff, wenn es an einem Hafen warten soll, zu jeder Zeit einen freien Ankerplatz zur Verfügung hat. Daher sollte, bevor der Knoten zum Warten ausgewählt wird, überprüft werden, ob ausreichende Ankerplätze vorhanden sind. Des Weiteren könnte die Größe des Schiffs und die Art der Ladung bei der Optimierung des Algorithmus berücksichtigt werden. Ein großes Schiff könnte beispielsweise einem schwächeren Sturm standhalten als ein kleines Schiff. In diesem Fall müsste kein neuer Pfad gesucht werden, da das Schiff möglicherweise in der Lage ist, den Sturm zu überstehen. Darüber hinaus wäre es sinnvoll, bei Unwettern die Windrichtung zu beobachten und einen Hafen in anderer Richtung zu suchen. Wenn der Sturm in die Richtung des Windes bläst und das Schiff in dieser Richtung wartet, bedeutet dies, dass die Gefahr auf das Schiff zukommt. Daher sollten

bei der Planung der Route auch die Windrichtung einbezogen werden, um das Risiko von Unwetterereignissen zu vermeiden.
Interessant wäre es, auch den Fall von parallel verkehrenden Schiffen zu betrachten. Der Algorithmus berechnet zwar den kürzesten Weg für ein einzelnes Schiff, aber wenn man die gesamte Flotte betrachtet, wird es komplizierter. Es kann vorkommen, dass mehrere Schiffe das gleiche Ziel haben bzw. dass alle Schiffe an demselben Knoten auf einen Sturm treffen und an demselben Hafen warten müssen. In solchen Fällen wäre es nützlich, eine koordinierte Planung für die Schiffe durchzuführen, um die Ressourcen optimal zu nutzen und die Wartezeiten für die Schiffe zu minimieren.

Wird dieser Ansatz weiterentwickelt und in der Praxis eingesetzt, wird sie einen entscheidenden Beitrag für Schiffe und den Seeverkehr leisten.

Literaturverzeichnis

[1] Richard Bellman. “On a routing problem”. In: *Quarterly of applied mathematics* 16.1 (1958), S. 87–90.

[2] Kenneth L Cooke und Eric Halsey. “The shortest route through a network with time-dependent internodal transit times”. In: *Journal of Mathematical Analysis and Applications* 14.3 (1966), S. 493–498. ISSN: 0022-247X. DOI: https://doi.org/10.1016/0022-247X(66)90009-6. URL: https://www.sciencedirect.com/science/article/pii/0022247X66900096.

[3] Ariel Orda und Raphael Rom. “Shortest-Path and Minimum-Delay Algorithms in Networks with Time-Dependent Edge-Length”. In: *J. ACM* 37.3 (Juli 1990), S. 607–625. ISSN: 0004-5411. DOI: 10.1145/79147.214078. URL: https://doi.org/10.1145/79147.214078.

[4] F. Harary und G. Gupta. “Dynamic graph models”. In: *Mathematical and Computer Modelling* 25.7 (1997), S. 79–87. ISSN: 0895-7177. DOI: https://doi.org/10.1016/S0895-7177(97)00050-2. URL: https://www.sciencedirect.com/science/article/pii/S0895717797000502.

[5] Ismail Chabini und Shan Lan. “Adaptations of the A* algorithm for the computation of fastest paths in deterministic discrete-time dynamic networks”. In: *IEEE Transactions on intelligent transportation systems* 3.1 (2002), S. 60–74. DOI: 10.1109/6979.994796. URL: https://www.researchgate.net/publication/3427825.

[6] M. Crang. “Speed = distance/time : chronotopographies of action.” In: *24/7 : time and temporality in the network society*. Hrsg. von R. Hassan und R. Purser. Stanford, CA: Stanford University Press, 2007, S. 62–88. URL: http://dro.dur.ac.uk/5027/.

[7] Giacomo Nannicini und Leo Liberti. “Shortest paths on dynamic graphs”. In: *International Transactions in Operational Research* 15.5 (2008), S. 551–563. DOI: https://doi.org/10.1111/j.1475-3995.2008.00649.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1475-3995.2008.00649.x.

[8] Bradley Casey u. a. “Critical Review of Time-Dependent Shortest Path Algorithms: A Multimodal Trip Planner Perspective”. In: *Transport Reviews* 34 (Juli 2014), S. 522–539. DOI: 10.1080/01441647.2014.921797.

[9] Dong Yang u. a. “How big data enriches maritime research – a critical review of Automatic Identification System (AIS) data applications”. In: *Transport Reviews* 39 (Juli 2019), S. 1–19. DOI: 10.1080/01441647.2019.1649315.

[10] Sunita und Deepak Garg. “Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue”. In: *Journal of King Saud University - Computer and Information Sciences* 33.3 (2021), S. 364–373. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.03.003. URL: https://www.sciencedirect.com/science/article/pii/S1319157817303828.

[11] NetworkX documentation. *NetworkX*. https://en.wikipedia.org/wiki/Graph_(discrete_mathematics). [Online; accessed 05-December-2022]. 2022.

[12] JAKOTA. *JAKOTA Cruise Systems GmbH*. https://www.fleetmon.com/. [Online; accessed 13-November-2022]. 2022.

[13] Study.com. *Locating Points on the Surface of the Earth*. https://study.com/academy/lesson/locating-points-on-the-surface-of-the-earth.html. [Online; accessed 8-December-2022]. 2022.

[14] Wikipedia. *A*-Algorithmus — Wikipedia, The Free Encyclopedia*. https://de.wikipedia.org/wiki/A*-Algorithmus. [Online; accessed 06-November-2022]. 2022.

[15] Wikipedia. *Dijkstra Algorithmus — Wikipedia, The Free Encyclopedia*. https://de.wikipedia.org/wiki/Dijkstra-Algorithmus. [Online; accessed 06-November-2022]. 2022.

[16] Wikipedia. *Fleet Management — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Fleet_management. [Online; accessed 07-November-2022]. 2022.

[17] Wikipedia. *Graph (discrete mathematics)*. https://en.wikipedia.org/wiki/Graph_(discrete_mathematics). [Online; accessed 05-December-2022]. 2022.

[18] Wikipedia. *Haversine Formula*. https://en.wikipedia.org/wiki/Haversine_formula. [Online; accessed 26-November-2022]. 2022.

[19] Wikipedia. *Pandas (Software)*. https://de.wikipedia.org/wiki/Pandas_(Software). [Online; accessed 28-December-2022]. 2022.

[20] Wikipedia. *Shortest path problem — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Shortest_path_problem. [Online; accessed 06-November-2022]. 2022.

[21] Maritime Injury Center. *Maritime Weather Forecasting*. [Online; accessed 21-January-2023]. URL: https://www.maritimeinjurycenter.com/accidents-and-injuries/weather-forecasting/.

[22] International Cargo Express. *How Bad Weather Impacts Shipping (and How to Deal With It)*. [Online; accessed 21-January-2023]. URL: https://icecargo.com.au/weather-impacts-shipping/.

[23] GIS Geography. *Latitude and Longitude - What Are They?* [Online; accessed 16-February-2023]. URL: https://gisgeography.com/latitude-longitude-coordinates.

[24] Matplotlib Development Team. *Matplotlib*. [Online; accessed 16-February-2023]. URL: https://matplotlib.org/stable/index.html.

[25] World Meteorological Organization. *Extreme Maritime Weather: Improving Safety of Life at Sea*. [Online; accessed 21-January-2023]. URL: https://public.wmo.int/en/resources/bulletin/Products_and_services/Extereme_maritime_weather.

[26] Sofar Ocean - Connecting the World's Oceans. *How Maritime Weather Forecasting Minimizes Risks in Shipping Operations*. [Online; accessed 21-January-2023]. URL: https://www.sofarocean.com/posts/how-maritime-weather-forecasting-minimizes-risks-in-shipping-operations.

Algorithmenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AIS (engl. Automatic Identification System) Automatisches Identifizierungssystem. 2, 19

CSV (engl. Comma-separated values) Durch Kommas getrennte Werte. 26–28

E Menge der Kanten im Graphen. 8, 29

E_s Jede Kante e ∈ E, der vom Sturm betroffen ist. 32

FIFO First-in-First-out: ist der Anglizismus für eine Reihenfolge, in der bestimmte Vorgänge zeitlich nacheinander abgearbeitet oder erledigt werden. 22

FW (engl. Forbidden waiting) Verbotenes Warten. 23

G Graph. 8, 29

LKW Lastkraftwagen. 19

PKW Personenkraftwagen. 19

SW (engl. Source waiting) Start-Warten. 23

UW (engl. Unrestricted waiting) Unbeschränktes Warten. 3, 23, 43–49, 56, 57

V Menge der Knoten im Graphen. 8, 29

V_s Jeder Knoten v ∈ V, der vom Sturm betroffen ist. 32

Erklärung über die selbständige Abfassung einer schriftlichen Arbeit

Hiermit erkläre ich, 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 Prüfungsbehörde vorgelegt.

Rostock, den 28.02.2023