Graphen: Dijkstras Algorithmus

Portfolioprüfung Algorithmen und Datenstrukturen

André-Anan Gilbert und Valentin Moritz Müller

**Inhaltsverzeichnis**

[Ehrenwörtliche Erklärung ii](#_Toc75078505)

[Abbildungsverzeichnis iii](#_Toc75078506)

[Quelltextverzeichnis iii](#_Toc75078507)

[1 Einleitung 1](#_Toc75078508)

[2 Graphen 1](#_Toc75078509)

[2.1 Ungerichtete Graphen 1](#_Toc75078510)

[2.2 Gerichtete Graphen 2](#_Toc75078511)

[2.3 Gewichtete Graphen 2](#_Toc75078512)

[3 Beschreibung des Dijkstra-Algorithmus 3](#_Toc75078513)

[3.1 Problemstellung und Voraussetzungen 3](#_Toc75078514)

[3.2 Beschreibung des Dijkstra-Algorithmus 3](#_Toc75078515)

[3.3 Lazy-Implementierung des Dijkstra-Algorithmus 3](#_Toc75078516)

[3.4 Eager-Implementierung des Dijkstra-Algorithmus 4](#_Toc75078517)

[4 A\* Search Algorithm 5](#_Toc75078518)

[5. Visualisierung der Algorithmen 7](#_Toc75078519)

[6. Anwendungsmöglichkeiten 8](#_Toc75078520)

[Anhang 9](#_Toc75078521)

[Literaturverzeichnis 9](#_Toc75078522)

# Ehrenwörtliche Erklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit mit dem Titel Graphen: Dijkstra-Algorithmus selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Unterschrift

# Abbildungsverzeichnis

[Abbildung 1: Ungerichteter Graph 1](#_Toc75078582)

[Abbildung 2: Gerichteter Graph 2](#_Toc75078583)

[Abbildung 3: Gewichteter Graph 2](#_Toc75078584)

[Abbildung 4: Darstellung der Vorteile von A\* Search gegenüber Dijkstra 5](#_Toc75078585)

[Abbildung 5: Darstellung der Heuristiken für den A\* Search-Algorithmus 6](#_Toc75078586)

[Abbildung 6: Visualisierung des Dijkstra-Algorithmus im Visualizer 8](#_Toc75078587)

# Quelltextverzeichnis

[Quellcodeauszug 1: Lazy-Implementierung des Dijkstra-Algorithmus 4](#_Toc75078590)

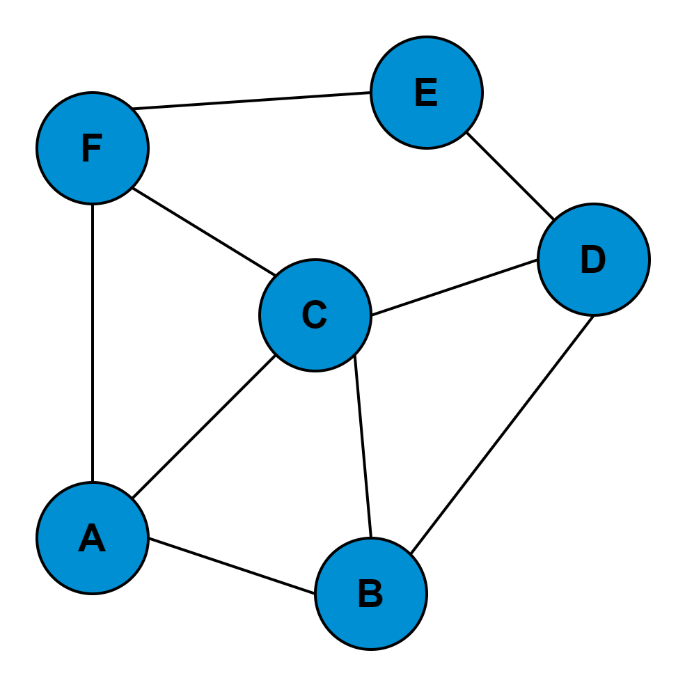
[Quellcodeauszug 2: Eager-Implementierung des Dijkstra-Algorithmus 5](#_Toc75078591)

# 1 Einleitung

Text

# 2 Graphen

## 2.1 Ungerichtete Graphen

Abbildung 1: Ungerichteter Graph

In einem ungerichteten Graphen haben die Kanten keine Orientierung oder Richtung. Dies bedeutet, dass die Kante (u, v) identisch ist mit der Kante (v, u). In einem derartigen Graphen könnten Knoten beispielsweise Städte darstellen und jede Kante eine bidirektionale Straße repräsentieren.

## 2.2 Gerichtete Graphen

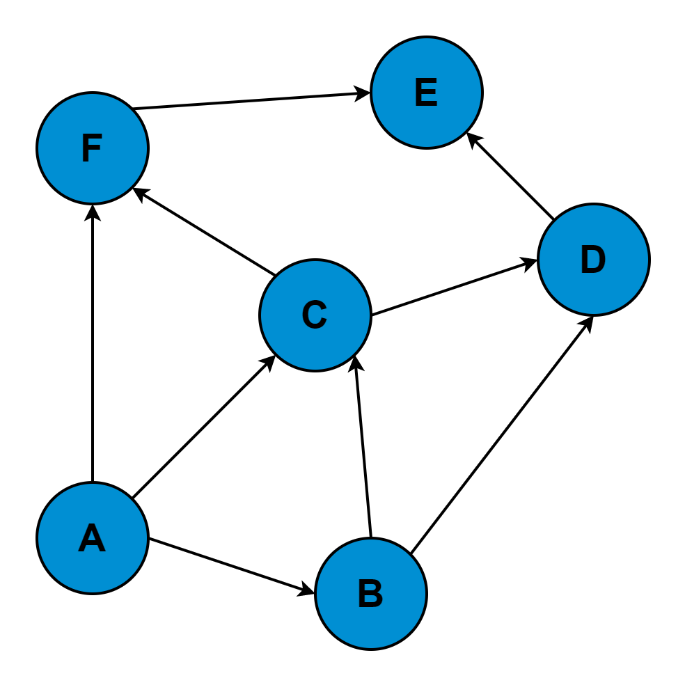


Abbildung 2: Gerichteter Graph

Im Gegensatz zum ungerichteten Graphen haben alle Kanten in einem gerichteten Graph eine Orientierung oder Richtung. Deshalb beschreibt eine Kante (u, v) den Weg von dem Knoten u zum Knoten v. In einem solchen Graphen könnten die Knoten Prozesse in einem System darstellen, während die Kanten Abhängigkeiten oder die Zugriffe der Prozesse aufeinander repräsentieren könnten.

## 2.3 Gewichtete Graphen

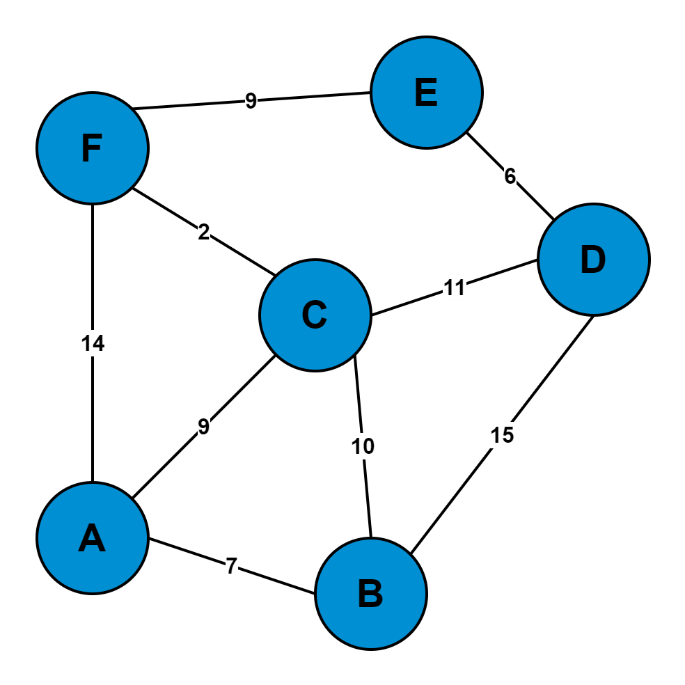


Abbildung 3: Gewichteter Graph

In einem gewichteten Graph enthalten die Kanten einen beliebigen Wert wie z.B. Kosten, eine Entfernung oder eine Menge. Dieser Wert wird als Gewicht der Kante bezeichnet. Gewichtete Kanten werden als Triplet (u, v, w) dargestellt und können in gerichteten oder ungerichteten Graphen vorkommen.

# 3 Beschreibung des Dijkstra-Algorithmus

## 3.1 Problemstellung und Voraussetzungen

Der Dijkstra-Algorithmus wird verwendet, um das Problem des kürzesten Pfades auf einem gewichteten Graphen zu lösen. Das Gesamtgewicht eines Pfades ergibt sich aus der Summe der Gewichte aller Kanten entlang des Pfades. Theoretisch ließe sich dies zwar umsetzen, indem einfach alle Pfade berechnet und denjenigen mit den geringsten Pfadkosten bzw. dem geringsten Gewicht ausgewählt werden, aufgrund der sehr hohen Anzahl und Einbeziehung vieler Pfade, deren Betrachtung nicht zielführend ist, da sie beispielsweise unnötige Umwege machen oder zyklische Segmente enthalten, sind dafür in der Praxis allerdings effizientere Algorithmen notwendig.

Eine Sonderform und Vereinfachung des kürzeste-Pfade-Problems stellt das kürzeste-Pfade-Problem mit Startknoten dar, zu dessen Lösung der Dijkstra-Algorithmus verwendet werden kann. Bei diesem Problem kann sowohl der kürzeste Pfad zu einem bestimmten Zielknoten, als auch der kürzeste Pfad zu jedem Knoten berechnet werden. Dabei beruhen Algorithmen, die zur Lösung dieses Problems verwendet werden, üblicherweise auf der Eigenschaft, dass ein kürzester Pfad zwischen zwei nicht benachbarten Knoten auch weitere kürzeste Pfade enthält. Diese Eigenschaft bezeichnet man als optimale-Teilstruktur-Eigenschaft, und sie stellt eine wichtige Voraussetzung für die Anwendbarkeit von Greedy-Algorithmen, zu denen auch Dijkstras Algorithmus zählt, dar.

Der Dijkstra-Algorithmus setzt weiterhin voraus, dass alle Gewichte des zu untersuchenden Graphen nicht negativ sind, da dies sonst zu zyklischen Pfaden führen könnte. Ob der Graph gerichtet oder ungerichtet ist spielt für die Anwendbarkeit von Dijkstras Algorithmus hingegen keine Rolle.

## 3.2 Beschreibung des Dijkstra-Algorithmus

Dijkstras Algorithmus nimmt zunächst unendliche Kosten zu allen Knoten außer dem Startknoten an, dieser bekommt Kosten von null zugewiesen und wird als aktuell aktiver Knoten gesetzt. Ausgehend vom aktiven Knoten werden anschließend die vorhandenen Kosten aller benachbarten Knoten (zu denen eine Kante führt) mit der Summe aus den dem aktiven Knoten zugewiesenen Kosten und den Kosten der Kante, die den aktiven mit dem benachbarten Knoten verbindet, verglichen. Dem benachbarten Knoten wird nun der geringere der beiden Werte als neue Kosten zugewiesen. Sind alle benachbarten Knoten auf diese Art abgehandelt, so wird der der aktuelle Knoten als besucht markiert sowie der unbesuchte Knoten, dem aktuell die geringsten Kosten zugewiesen sind, als aktiver Knoten gesetzt und das Vorgehen wiederholt.

Falls alle Knoten besucht wurden oder die geringsten Kosten zu einem unbesuchten Knoten unendlich sind (dies bedeutet, dass kein Pfad vom Startknoten zu diesem Knoten existiert), ist Dijkstras Algorithmus beendet.

Ist alternativ nur ein bestimmter Knoten als Zielknoten bekannt und ein Pfad ist nur zu diesem gewünscht, so kann der Dijkstra-Algorithmus bereits beendet werden, sobald dieser als aktiver Knoten gesetzt wird, da zu diesem Zeitpunkt bereits der kürzeste Pfad zum Zielknoten gefunden wurde. Dies wird als Early Stopping bezeichnet und stellt eine Optimierung des Algorithmus dar, da so in vielen Fällen deutlich weniger Knoten besucht und Pfade überprüft werden müssen.

In Kapitel 3.1 wurde Dijkstras Algorithmus als zu den Greedy-Algorithmen gehörig klassifiziert. Hierbei ist anzumerken, dass, obwohl Greedy-Strategien allgemein nicht immer auch zu optimalen Ergebnissen führen, der Dijkstra-Algorithmus stets korrekt ist, also tatsächlich die kostengünstigsten Pfade berechnet.

## 3.3 Lazy-Implementierung des Dijkstra-Algorithmus

In seiner einfachsten Form, in der die Knoten in einem Feld gespeichert sind und keine explizite Prioritätswarteschlange (Priority Queue) implementiert wird, besitzt Dijkstras Algorithmus eine zeitliche Komplexität von

Mit einer Priority Queue verbessert sich die Zeitkomplexität auf

Bei diesen und allen folgenden Komplexitätsangaben bezeichnet V die Anzahl der Knoten (engl. Vertices) und E die Anzahl der Kanten (engl. Edges). In Quellcodeauszug 2 ist zu sehen, wie Dijkstras Algorithmus mit einer einfachen Priority Queue implementiert werden kann.

function dijkstra\_lazy(graph, n, start, destination):

visited = [false, false, …, false] # size n

prev = [null, null, …, null] # size n

costs = [∞, ∞, …, ∞, ∞] # size n

costs[start] = 0

pq = empty priority queue

pq.insert((start, 0))

while pq.size() != 0:

current, min\_value = pq.poll()

visited[current] = true

for edge in graph[current]:

if visited[edge.destination]: continue

new\_cost = costs[current] + edge.cost

if new\_cost < costs[edge.destination]:

prev[edge.destination] = current

distance[edge.destination] = new\_cost

pq.insert((edge.destination, new\_cost))

if current == destination:

return distance[destination]

return ∞

Quellcodeauszug 1: Lazy-Implementierung des Dijkstra-Algorithmus

Der Algorithmus kann in diesem Fall dazu führen, dass Duplikate in der Priority Queue landen. Wird daraufhin ein Knoten als aktiv gesetzt, bei dem die aktuell bekannten Kosten geringer als die aus der Queue erhaltenen kosten sind, wird dieser ignoriert. Dies wird als Lazy Deleting bezeichnet und ist der Grund dafür, dass diese Implementierung Lazy-Implementierung genannt wird.

## 3.4 Eager-Implementierung des Dijkstra-Algorithmus

Die Eager-Implementierung von Dijkstras Algorithmus ist der Lazy-Implementierung sehr ähnlich, der wichtigste Unterschied besteht darin, dass statt einer normalen eine indizierte Priority Queue eingesetzt wird. Dies verhindert, dass einzelne Knoten mehrfach eingefügt werden, in der indizierten Queue werden stattdessen die Kosten vorhandener Knoten aktualisiert, wenn ein Pfad mit geringeren Kosten gefunden wird. Dieses Verfahren wird als Decrease Key bezeichnet. Decrease-Key-Operationen sind kostengünstiger als das Entfernen von Elementen aus einer Priority Queue oder einem Heap und somit für eine effiziente Implementierung zu bevorzugen. Die Eager-Implementierung ist besonders bei dicht besetzten Graphen von Vorteil, entspricht hinsichtlich der Zeitkomplexität allerdings der Lazy-Implementierung, da bei der Komplexität immer der ungünstigste Fall betrachtet wird.

In Quellcodeauszug 2 ist die Eager-Implementierung von Dijkstras Algorithmus in Form von Pseudocode dargestellt.

function dijkstra\_eager(graph, n, start, destination):

visited = [false, false, …, false] # size n

prev = [null, null, …, null] # size n

costs = [∞, ∞, …, ∞, ∞] # size n

costs[start] = 0

ipq = empty indexed priority queue

ipq.insert((start, 0))

while pq.size() != 0:

current, min\_value = ipq.poll()

visited[current] = true

for edge in graph[current]:

if visited[edge.destination]: continue

new\_cost = costs[current] + edge.cost

if new\_cost < costs[edge.destination]:

prev[edge.destination] = current

distance[edge.destination] = new\_cost

if edge.destination not in ipq:

ipq.insert(edge.destination, new\_cost)

else:

ipq.descreaseKey(edge.destination, new\_cost)

if current == destination:

return distance[destination]

return ∞

Quellcodeauszug 2: Eager-Implementierung des Dijkstra-Algorithmus

## 3.5 Weitere Optimierungsmöglichkeiten

Neben dem bereits erwähnten Early Stopping bei bekanntem Zielknoten und der Eager-Implementierung existieren weitere Optimierungsmöglichkeiten für den Dijkstra-Algorithmus. Im Allgemeinen, insbesondere aber auf dichten Graphen, überwiegt die Anzahl der Updates wie Decrease-Key gegenüber der Anzahl der Removals, beispielsweise Dequeue bzw. Poll-Operationen. Durch einen Binary oder D-ary Heap, wobei D die Anzahl der Kinder pro Knoten bezeichnet und sich aus dem Quotienten von Kanten und Knoten im Graphen errechnet, können Decrease-Key-Operationen auf Kosten ineffizienterer Removal-Operationen beschleunigt werden, was durch das Überwiegen der Update-Operationen jedoch insgesamt die Laufzeit beziehungsweise Kosten des Algorithmus verbessert.

Der Binary Heap besitzt eine Zeitkomplexität von

, der D-ary Heap sogar von

sofern alle Knoten vom Startknoten erreichbar sind.

Für eine optimale Implementierung von Dijkstras Algorithmus kann durch Verwendung eines Fibonacci-Heaps eine Zeitkomplexität von

erreicht werden.

Da die in Python integrierte Binary Heap weder über Constant Lookup noch über eine Decrease-Key-Operation verfügt und diese selbst implementiert werden mussten, erreicht die umgesetzte Eager-Implementierung mit Binary Heap nicht die optimale Zeitkomplexität für diesen Fall. Durch den manuellen Lookup mithilfe einer Iteration über im Heap befindliche Knoten findet der Lookup in linearer statt konstanter Zeit statt, was wiederum die Zeitkomplexität einer einzelnen Decrease-Key-Operation bei Graphen mit vielen Knoten erhöht.

# 4 A\* Search Algorithm

Ein bekanntes Problem von Dijkstra‘s Algorithm ist, dass Dijkstra immer die vielversprechendsten Kanten der Reihe nach verarbeitet. Das führt ggf. dazu, dass Dijkstra in die falsche Richtung geht (siehe Abbildung 4), da die Kante in diese Richtung ein niedrigeres Gewicht aufweist.

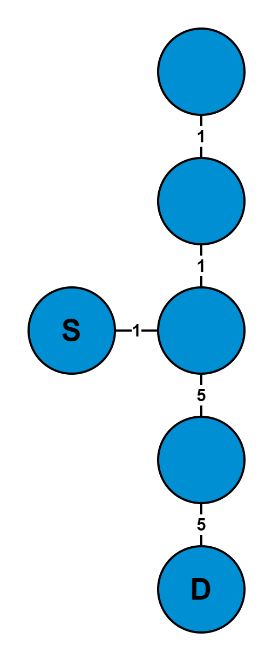


Abbildung 4: Darstellung der Vorteile von A\* Search gegenüber Dijkstra

A\* Search ist ein Algorithmus, der bei der Wegfindung und Durchquerung von Graphen weit verbreitet ist. Der Algorithmus zeichnet effizient einen begehbaren Pfad zwischen mehreren Knoten oder Punkten auf dem Graphen. Auf einer Karte mit vielen Hindernissen kann die Wegfindung von den Punkten AA bis BB schwierig sein. Der A\* Search Algorithmus führt jedoch eine Heuristik in einen regulären Algorithmus ein, der im Wesentlichen bei jedem Schritt vorausschaut, damit eine bessere Entscheidung getroffen wird. Deshalb kann A\* Search als eine Erweiterung des Dijkstra-Algorithmus mit einigen Merkmalen der Breitensuche (BFS) betrachtet werden. Wie Dijkstra arbeitet A\*, indem es einen kostengünstigsten Pfad vom Startknoten zum Zielknoten erstellt. Was der A\* Search für viele Suchen anders und besser macht, ist, dass der Algorithmus für jeden Knoten eine Funktion f(n) verwendet, die eine Schätzung der Gesamtkosten eines Pfades unter Verwendung dieses Knotens liefert. Daher ist A\* Search eine heuristische Funktion, die sich von einem Algorithmus dadurch unterscheidet, dass eine Heuristik eher eine Schätzung ist, die nicht zwangsläufig korrekt sein muss.

A\* erweitert Pfade, die bereits kostengünstiger sind, indem diese Funktion verwendet wird:

f(n) := geschätzte Gesamtkosten des Pfades durch den Knoten n

g(n) := Kosten bis zum Erreichen von Knoten n

h(n) := geschätzte Kosten von n bis zum Ziel.

Die Verwendung einer guten Heuristik ist wichtig, um die Leistung von A\* Search zu bestimmen. Der Wert von h(n) würde idealerweise den genauen Kosten für das Erreichen des Ziels entsprechen. Dies ist jedoch nicht möglich, da der Weg nicht bekannt ist. Wir können jedoch eine Methode wählen, die uns manchmal den genauen Wert liefert, z.B. wenn Sie geradeaus ohne Hindernisse fahren. Dies führt zu einer perfekten Leistung von A\* in einem solchen Fall.

Dazu ist es notwendig, in der Lage zu sein, eine Funktion h(n) auszuwählen, die geringer ist als die Kosten zum Erreichen des Zieles. Dadurch kann h genau arbeiten. Wenn wir einen höheren Wert für h wählen, führt dies zu einer schnelleren, aber weniger genauen Leistung. Daher ist es üblich ein h(n) zu wählen, das geringer ist als die realen Kosten. Der A\* Search implementiert einer der beiden Heuristiken Manhattan Distanz oder Euklidische Distanz.

Die erste der beiden Methoden zur Berechnung von h(n) wird als Manhattan-Methode bezeichnet, weil sie die Gesamtzahl der Quadrate berechnet, die horizontal und vertikal bewegt wurden, um das Zielquadrat vom aktuellen Quadrat aus zu erreichen. In diesem Fall ignorieren wir diagonale Bewegungen und eventuelle Hindernisse.

Diese Heuristik ist immer dann genau, wenn unser Weg einer geraden Linie folgt. Das heißt, A\* Search findet Pfade, die Kombinationen von geradlinigen Bewegungen sind. Manchmal bevorzugen wir vielleicht einen Weg, der einer geraden Linie direkt zu unserem Ziel folgt.

Die Euklidische Distanz ist etwas genauer als ihr Gegenstück. Wenn wir versuchen, beide gleichzeitig im selben Labyrinth zu laufen, bevorzugt die euklidische Heuristik einen Pfad entlang einer geraden Linie. Dies ist genauer, aber auch langsamer, da ein größeres Gebiet erkundet werden muss, um den Weg zu finden.



Abbildung 5: Darstellung der Heuristiken für den A\* Search-Algorithmus

Der Hauptnachteil des A\* Algorithmus und in der Tat jeder Best-First-Suche ist sein Speicherbedarf. Da zumindest die gesamte offene Liste gespeichert werden muss, ist der A\*-Algorithmus in der Praxis stark räumlich begrenzt und nicht praktischer als der Best-First-Suchalgorithmus auf aktuellen Maschinen.

Die Zeitkomplexität von A\* hängt von der Heuristik ab. Im schlimmsten Fall ist die Anzahl der expandierten Knoten exponentiell in der Länge der Lösung (der kürzeste Weg), aber polynomiell, wenn der Suchraum ein Baum ist.

# 5. Visualisierung der Algorithmen

Der Pathfinding Visualizer basiert auf dem Modul Pygame und implementiert sowohl den Dijkstra-Algorithmus mit einer Prioritätswarteschlange als auch den A\* Search Algorithmus mit der Manhatten Distanz als Heuristikfunktion auf einem Graphen, der aus x, y Koordinaten besteht. Anhand eines selbst ausgewählten Start- und Zielknoten können Sie beliebig oft die Algorithmen visualisieren und zudem auch selbst gewählte Hindernisse dem Graphen hinzufügen oder sich ein Labyrinth aus Hindernissen generieren lassen.

Im Folgenden werden die Bedeutungen der Farben im Visualizer definiert:

* Startknoten: grün
* Zielknoten: rot
* Hindernis: schwarz
* Kürzester Pfad: gelb
* Besuchte Knoten: blau
* Zu besuchende Knoten: lila
* Unbesuchte Knoten: weiß

Im Folgenden wird die Funktionsweise des Visualizers erklärt:

* Drücken Sie die linke Maustaste um Start-, Zielknoten sowie Hindernisse festzulegen
* Drücken Sie die rechte Maustaste um Start-, Zielknoten sowie Hindernisse zu entfernen
* Drücken Sie c um all Knoten zurückzusetzen
* Drücken Sie a um den A\* Search Algorithmus zu visualisieren
* Drücken Sie d um den Dijkstra-Algorithmus zu visualisieren
* Drücken Sie m um ein zufälliges Labyrinth aus Hindernisse zu generieren

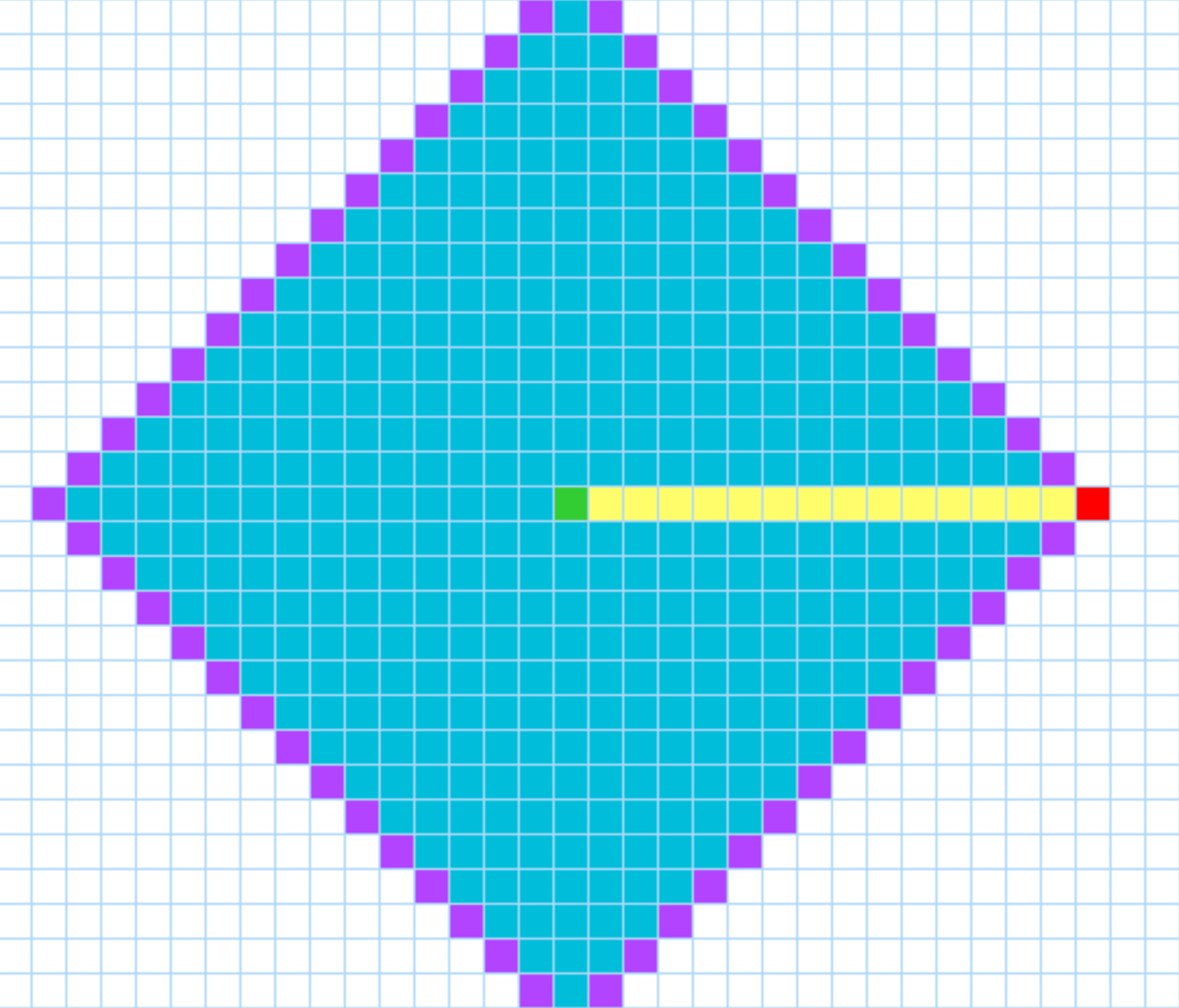


Abbildung 6: Visualisierung des Dijkstra-Algorithmus im Visualizer

# 6. Anwendungsmöglichkeiten

Wir haben oft versucht, in Google Maps oder Apple Karten die Entfernung von einer Stadt zur anderen oder von Ihrem Standort zum nächsten gewünschten Standort zu ermitteln. Dort trifft man auf den Dijkstra-Algorithmus, da es verschiedene Routen bzw. Pfade zum Ziel gibt. Betrachten Sie Deutschland als Graph und stellen Sie eine Stadt oder einen Ort mit einem Scheitelpunkt und die Route zwischen zwei Städten oder Orten als Kante dar. Dann können Sie Dijkstras Algorithmus verwenden, um die kürzeste Route zwischen zwei beliebigen Städten oder Orten zu berechnen.

Auch bei der Erstellung eines Flugplans für die Kunden kommt Dijkstra zum Einsatz. Der Agent hat Zugriff auf eine Datenbank mit allen Flughäfen und Flügen. Neben Flugnummer, Abflughafen und Zielort haben die Flüge Abflug- und Ankunftszeit. Insbesondere möchte der Agent die früheste Ankunftszeit für das Ziel bei gegebenem Startflughafen und Startzeit bestimmen.

Dijkstras Algorithmus wird auch im Internet als Routing-Algorithmus im Open-Shortest-Path-First-Protokoll (OSPF) eingesetzt, das verwendet wird, um den besten Pfad zwischen dem Quell- und dem Zielrouter mithilfe seines eigenen Shortest Path First zu finden. Dijkstras Algorithmus wird häufig in den Routing-Protokollen verwendet, die von den Routern benötigt werden, um ihre Weiterleitungstabelle zu aktualisieren. Der Algorithmus stellt dabei den kürzesten Kostenpfad vom Quellrouter zu anderen Routern im Netzwerk bereit.

Um einen Dateiserver in einem LAN zu bestimmen, kann der Dijkstra Algorithmus ebenfalls verwendet werden. Bedenken Sie, dass für die Übertragung von Dateien von einem Computer auf einen anderen Computer unendlich viel Zeit benötigt wird. Um die Anzahl der „Hops“ vom Dateiserver zu jedem anderen Computer im Netzwerk zu minimieren, besteht die Idee darin, den kürzesten Pfad zwischen den Netzwerken zu minimieren, was zu einer minimalen Anzahl von Hops führt.

# Anhang

# Literaturverzeichnis