HTL ST. PÖLTEN

HTBLuVA St. Pölten
Höhere Abteilung für Informatik

htl
bildung mit zukunft

DIPLOMARBEIT

Einsatz von LiDAR im autonomen Fahren

Ausgeführt im Schuljahr 2023/24 von:
Philip Fenk, 5AHIF-3
Emilio Zottel, 5AHIF-22
Marco Molnár, 5AHIF-10
Adrián Kalapis, 5AHIF-9

Betreuer:
Dipl.-Ing. Christoph Schreiber
Dipl.-Ing. Wolfgang Raab

St. Pölten, am 11. Februar 2024

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Philip Fenk

Emilio Zottel

Marco Molnár

Adrian Kalapis

St.Pölten, am 11. Februar 2024

Diplomandenvorstellung

Philip FENK

Geburtsdaten:
28.08.2005 in St. Pölten

Wohnhaft in:
Kressgasse 5
3040 Neulengbach

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
BRG/BORG St. Pölten
2011 - 2015:
Volksschule Neulengbach

Kontakt:
philip.fenk@gmail.com

Emilio ZOTTEL

Geburtsdaten:
11.05.2005 in St. Pölten

Wohnhaft in:
Waldstraße 8
3061 Schönfeld

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
Neue Mittelschule Neulengbach
2011 - 2015:
Volksschule St. Christophen

Kontakt:
emilio.zottel@gmail.com

Marco MOLNÀR

Geburtsdaten:
02.01.2005 in Lilienfeld

Wohnhaft in:
Josef-Reither Straße 17a
3430 Tulln an der Donau

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
Bundesgymnasium Tulln
2011 - 2015:
Volksschule Asperhofen
Kontakt:
mmarco.molnar@gmail.com

Adrián KALAPIS

Geburtsdaten:
17.06.2003 in Pancevo

Wohnhaft in:
Waldbachstraße 2/1
3041 Siegersdorf

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
NMS Neulengbach
2010 - 2015:
Grundschule Zarko Zrenjanin

Kontakt:
kalapis.adrian03@gmail.com

Danksagungen

Danke

Inhaltsverzeichnis

Kapitel 1

Pathfinding

1.1 Allgemeines

1.1.1 Einleitung

Fährt man viel mit dem Auto oder öffentlichen Verkehrsmitteln, ist ein zuverlässiges Navigationssystem heutzutage so gut wie vorausgesetzt. Man gibt sein gewünschtes Ziel ein und schon werden die kürzesten Routen ausgehend vom aktuellen Standort zum Zielort vorgeschlagen. Dieser Luxus wäre ohne *Pathfinding* undenkbar. Pathfinding, im Deutschen auch als Wegfindung bezeichnet, ist ein entscheidender Bestandteil vieler Anwendungen und Systeme, bei denen die Navigation von einem Startpunkt zu einem oder mehreren weiteren Punkten erforderlich ist. Sei es die Berechnung der schnellsten Route für den täglichen Arbeits- oder Schulweg oder die Pfadplanung für autonome Roboter in einer Fabrik. In seiner grundlegendsten Form handelt es sich um die Suche nach dem besten Pfad von einem Startpunkt zu einem Zielpunkt in einer gegebenen Menge von Punkten. Ein Beispiel für Pathfinding ist in *Abbildung 1.1* veranschaulicht. Der optimale Pfad ist grün gefärbt und verläuft von links nach rechts. [EZ:Web01]

1.1.2 Aufwand

Das Ziel von Pathfinding ist es, einen Weg zu finden, der den geringstmöglichen Aufwand erfordert. Dieser Aufwand kann in verschiedenen Kontexten unterschiedlich definiert sein, wie beispielsweise als die kumulative Entfernung oder Zeit zwischen den

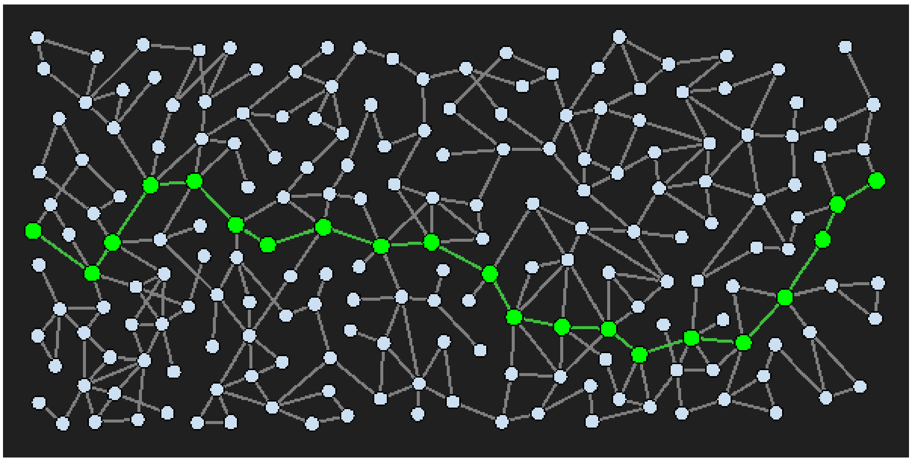

Abbildung 1.1: Beispiel für Pathfinding auf einem Graphen
[EZ:Web05]

überquerten Knoten, oder, sowohl fiktive als auch finanzielle, Kosten. Unter fiktiven Kosten kann man hier z.B. die Summe der Gewichtungen der Kanten, die der Pfad überquert, verstehen.

1.2 Graphentheorie

Die Graphentheorie ist eines der wichtigsten Konzepte im Bereich des Pathfinding. Ein Graph ist eine abstrakte, mathematische Struktur, die aus Knoten und Kanten besteht. In Bezug auf Pathfinding repräsentieren die Knoten die Standorte oder Punkte, zwischen denen man Wege finden möchte, und die Kanten stellen die theoretisch möglichen oder tatsächlich vorhandenen Verbindungen zwischen diesen Punkten dar. Bei Navigationssystemen für die reale Welt, wie Google Maps, Waze und anderen, repräsentieren Kanten z.B. Straßen und Knoten die Kreuzungen. [EZ:Web03, EZ:Web06, EZ:Web29]

1.2.1 Visualisierung

In den meisten Fällen werden Knoten als Kreise oder Punkte dargestellt, oft auch mit einer zusätzlichen Beschriftung oder Bezeichnung. Die Kanten werden meist als einfache Linien zwischen den Knoten dargestellt, ist der Graph jedoch ein gerichteter, sind es Pfeile anstatt Linien. Ist er gewichtet, sind die einzelnen Gewichtungen meist neben den dazugehörigen Kanten aufzufinden. [EZ:Web07]

1.2.2 Kantengewichtete Graphen

Ein Graph $G = (V, E)$ wird als kantengewichtet bezeichnet, wenn jeder Kante $e \in E$ eine Gewichtung $w(e)$ zugeordnet wird, wobei $w : E \to \mathbb{R}$ beziehungsweise $w : V \times V \to \mathbb{R}$ die Kantengewichtungsfunktion ist. Die Gewichtungen der Kanten können abhängig vom Anwendungsfall unterschiedlich interpretiert werden. Meistens stellen sie die euklidische Distanz zwischen zwei Knoten dar, oftmals aber auch die benötigte Zeit, um vom einen Knoten zum anderen zu gelangen. Letzteres ist zum Beispiel bei der Routenplanung im Straßenverkehr nützlicher, da stockender Verkehr und Staus berücksichtigt werden können. Es gibt auch Graphen, die knotengewichtet sind, diese finden aber nur selten Verwendung. Aus diesem Grund werden kantengewichtete Graphen meist nur als gewichtet bezeichnet. [EZ:Web04, EZ:Web10, EZ:Web22, EZ:Web32, EZ:Web35]

Verläuft in einem gewichteten Graphen zwischen zwei Knoten u und v keine Kante, gilt

$$w(u, v) = \infty$$

weil ein Wert von 0 eine Kante mit einer Gewichtung von 0 implizieren würde, solche Kanten aber durchaus sinnvoll sein können. Für ungewichtete Graphen gilt eine spezielle Kantengewichtungsfunktion, die folgendermaßen definiert ist:

$$w(u, v) = \begin{cases} 1 & \text{falls u und v verbunden sind,} \\ 0 & \text{sonst.} \end{cases}$$

Abbildung 1.2 zeigt ein einfaches Beispiel für einen Graphen. Er besteht aus fünf Knoten und fünf Kanten, die Verbindungen zwischen den Knoten darstellen. Alle Knoten sind mit einzigartigen Buchstaben beschriftet, damit man sie eindeutig identifizieren kann. Die Zahlen neben den einzelnen Kanten sind deren jeweiligen Gewichtungen, was bedeutet, dass der gezeigte Graph ein gewichteter ist. [EZ:Web42, EZ:Web43]

1.2.3 Gerichtete Graphen

Ein gerichteter Graph ist ein Graph, dessen Kanten nur in eine Richtung überquert werden dürfen. Wird eine ungerichtete Kante zwischen zwei Knoten benötigt, werden stattdessen zwei *gegenläufige*, gerichtete Kanten verwendet. Gegenläufig oder *antiparallel* nennt man zwei Kanten e_1, e_2 mit $e_1 = (a, b)$ und $e_2 = (b, a)$, wobei $a, b \in V$. Die Kanten eines gerichteten Graphen nennt man gerichtete Kanten und sind geordnete

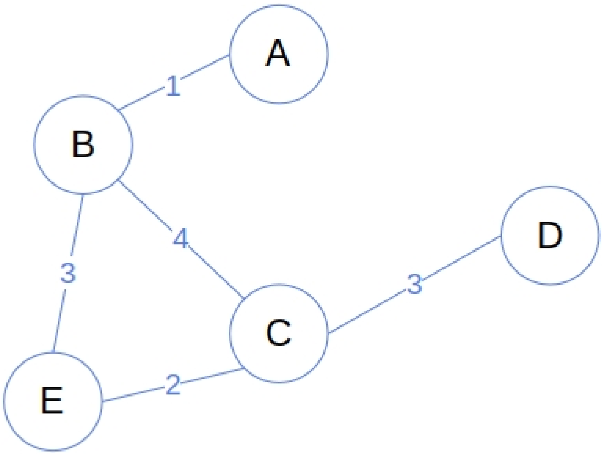

Abbildung 1.2: Ein gewichteter Graph
[EZ:Web04]

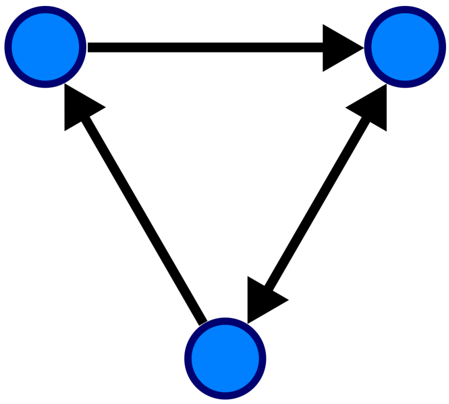

Abbildung 1.3: Ein Digraph
[EZ:Web20]

Knotenpaare $(a, b) \in E$ wobei man Kanten eines ungerichteten Graphen als ungerichtet bezeichnet werden und ungeordnete Knotenpaare $\{a, b\} \in E$ sind. Ein gerichteter Graph wird häufig auch als *Digraph*[1] bezeichnet. Ein Beispiel eines Digraphen ist in *Abbildung 1.3* zu sehen. [EZ:Web20]

1.2.4 Multigraphen

Multigraphen sind Graphen, in denen sowohl *Multikanten* als auch *Schlingen* vorkommen dürfen. Als Multikanten oder Mehrfachkanten bezeichnet man mehrere gleichartige Kanten, die durch ein und dasselbe Knotenpaar verlaufen. Multikanten, die den-

[1] directed graph

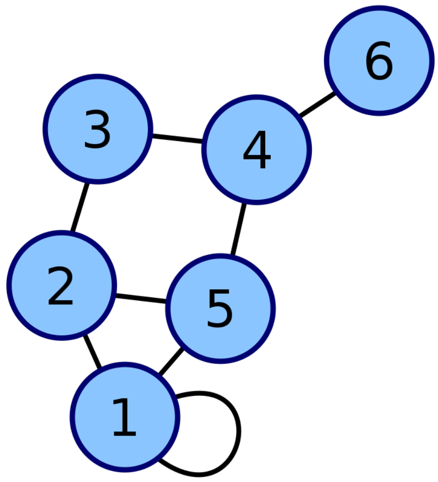

Abbildung 1.4: Ein Multigraph mit einer Schlinge
[EZ:Web27, EZ:Web28]

selben Anfangs- und Endknoten haben, nennt man *parallel*. Sind zwei Multikanten, die durch dieselben zwei Knoten verlaufen, gerichtet, und zeigen diese in entgegengesetzte Richtungen, werden sie als *gegenläufig* oder *antiparallel* bezeichnet (*siehe Kapitel 1.2.3*). [EZ:Web29, EZ:Web39]

Schlingen oder *Schleifen* sind Kanten, die einen Knoten mit sich selbst verbinden. Ab hier wird in dieser Arbeit ausschließlich der Begriff *Schlinge* verwendet, um Verwechslungen mit Schleifen aus der Programmierung zu vermeiden. *Abbildung 1.4* veranschaulicht einen Multigraphen mit einer Schlinge. In den Abbildungen 1.5 und 1.6 sind gerichtete beziehungsweise ungerichtete Multigraphen visualisiert. [EZ:Web27, EZ:Web28, EZ:Web39]

1.2.5 Einfache Graphen

Im Unterschied zu Multigraphen bezeichnet man Graphen, die ungerichtet sind und weder Multikanten noch Schlingen aufweisen, als *einfach* oder *schlicht*. In *Abbildung 1.7* ist ein Beispiel eines einfachen Graphen zu sehen. [EZ:Web26]

1.2.6 Netzwerke

Ist ein Graph sowohl gewichtet als auch gerichtet, ist er also ein gewichteter Digraph, spricht man von einem Netzwerk. Die Definition ist jedoch nicht einheitlich, zum Bei-

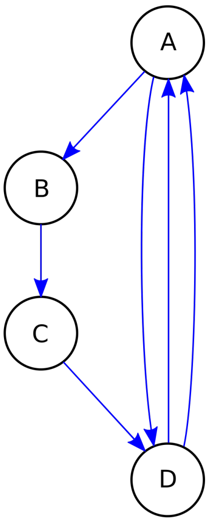

Abbildung 1.5: Ein gerichteter Graph mit Multikanten
[EZ:Web29]

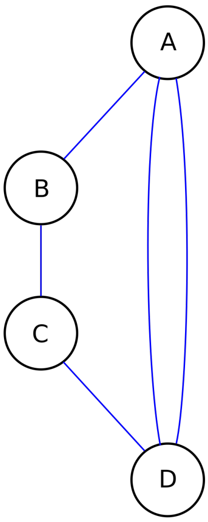

Abbildung 1.6: Ein ungerichteter Graph mit Multikanten
[EZ:Web29]

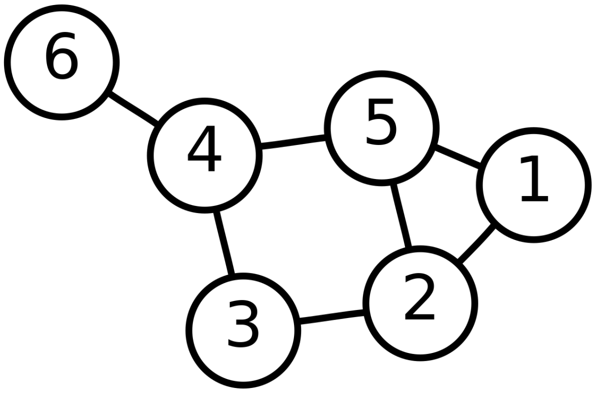

Abbildung 1.7: Ein einfacher Graph mit sechs Knoten und sieben Kanten
[EZ:Web25]

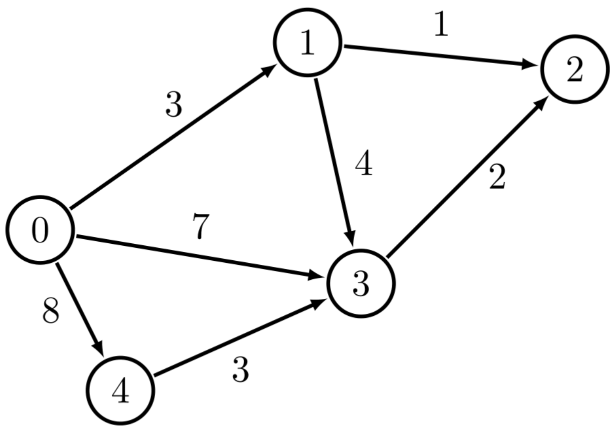

Abbildung 1.8: Ein Netzwerk
[EZ:Web10]

spiel sind umgangssprachlich oft nur gewichtete Graphen oder gar Graphen im Allgemeinen gemeint, wenn von Netzwerken die Rede ist. In *Abbildung 1.8* ist ein Netzwerk mit fünf nummerierten Knoten und sieben Kanten zu sehen. [EZ:Web09, EZ:Web25]

1.2.7 Symbolik

In den kommenden Abschnitten werden einige Symbole der Mengenlehre und Prädikatenlogik eingesetzt. Um klarzustellen, was diese bedeuten, sind die Definitionen der wichtigsten davon in Tabelle 1.1 auf Deutsch und auf Englisch aufzufinden.

Symbol	Bedeutung	Meaning
$\{\ldots\}$	Menge	set
$\|\ldots\|$	Kardinalität (Mächtigkeit)	cardinality
$\in$	Element von	element of
$\notin$	kein Element von	not element of
$\exists$	es existert mindestens ein	there exists at least one
$\exists!$	es existiert genau ein	there exists one and only one
$\forall$	für alle	for all

Tabelle 1.1: Wichtige Symbole der Mengenlehre
[EZ:Web23, EZ:Web24]

1.2.8 Zyklen

Ein Graph $G = (V, E)$ ist genau dann zyklisch, wenn seine Kanten mindestens einen *Zyklus* bilden. Ein Zyklus ist eine Teilmenge der Kantenmenge E eines Graphen, die einen Pfad formt, sodass der erste Knoten des Pfades dem letzten entspricht. Einen Zyklus in einem gerichteten Graphen bezeichnet man als gerichteten Zyklus und einen Zyklus in einem ungerichteten Graphen als ungerichteten Zyklus. Für einen ungerichteten Zyklus der *Länge* $k \in \mathbb{N}$ mit der Knotenfolge $(v_1, v_2, \ldots, v_k, v_1)$ gilt somit $\forall i \in \{1, 2, \ldots, k\} \exists e_i$ wobei $e_i = \{v_i, v_{i+1}\} \in E$ eine Kante ist, die v_i mit v_{i+1} verbindet und $v_{k+1} = v_1$. Somit sind v_k und v_{k+1} miteinander verbunden, was impliziert, dass v_k mit v_1 verbunden ist und sich somit ein Zyklus bildet. [EZ:Web11, EZ:Web12]

Obiges gilt auch für gerichtete Zyklen, jedoch ist die Länge dort oft nicht, wie bei ungerichteten Zyklen, als die Anzahl der Knoten oder Kanten, aus denen der Zyklus besteht, definiert, sondern als die Summe der Gewichtungen der im Zyklus überquerten Kanten. [EZ:Web16, EZ:Web17]

1.2.9 Kreise

Ein *Kreis* ist eine Sonderform eines Zyklus, bei dem der Pfad ein einfacher ist, also sind nicht nur die überquerten Kanten des Pfades einzigartig, sondern auch die überquerten Knoten. Somit müssen sich die Knoten $v_1, v_2, \ldots, v_k$ alle voneinander unterscheiden. *Abbildung 1.9* zeigt ein simples Beispiel eines zyklischen Graphen, dessen Zyklus ein Kreis ist. [EZ:Web13]

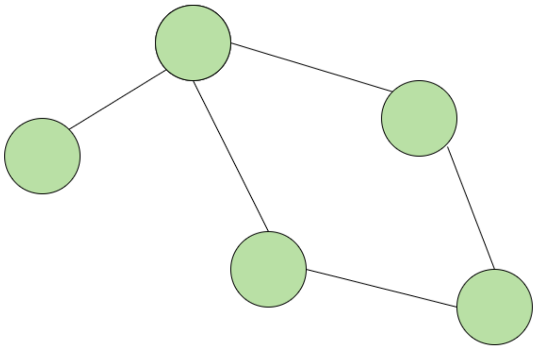

Abbildung 1.9: Ein zyklischer Graph mit einem Kreis [EZ:Web12]

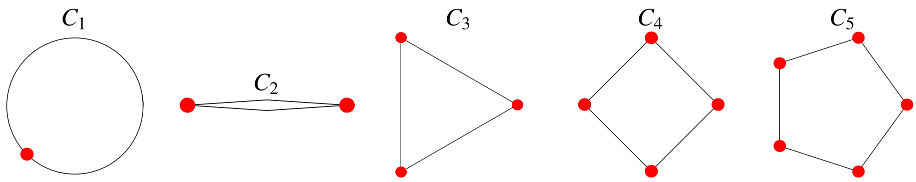

Abbildung 1.10: Die Kreisgraphen C_1, C_2, C_3, C_4 und C_5 [EZ:Web14]

1.2.10 Kreisgraphen

Enthält ein Graph genau einen Zyklus, welcher gleichzeitig ein Kreis ist, und besteht dieser aus der gesamten Knotenmenge V des Graphen, bezeichnet man den Graphen als *Kreisgraph*. Für einen Kreisgraphen gilt immer

$$|V| = |E|$$

was bedeutet, dass die Anzahl der Knoten mit der der Kanten übereinstimmt. Die ersten fünf Kreisgraphen C_1, C_2, C_3, C_4 und C_5 sind in *Abbildung 1.10* veranschaulicht. Der Knoten des Kreisgraphen C_1 ist durch eine Schlinge mit sich selbst verbunden. [EZ:Web14, EZ:Web15]

1.2.11 Vollständige Graphen

Ein vollständiger Graph ist ein einfacher Graph, in dem jeder Knoten eine Kante zu allen anderen im Graphen enthaltenen Knoten hat. Anders ausgedrückt: Jedes Paar von unterschiedlichen Knoten ist durch eine Kante verbunden. Für einen *vollständigen*

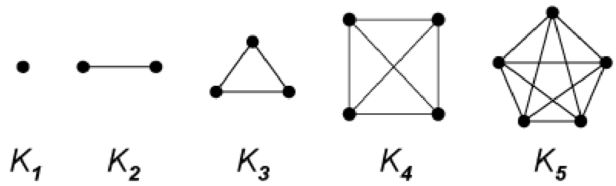

Abbildung 1.11: Die vollständigen Graphen K_1, K_2, K_3, K_4 und K_5 [EZ:Web33]

Digraphen gilt ähnlich: Jedes Paar von unterschiedlichen Knoten ist durch *ein Paar von unterschiedlichen Kanten* verbunden, also mit einer Kante je Richtung. In *Abbildung 1.11* sind die vollständigen Graphen K_1, K_2, K_3, K_4 und K_5 zu sehen. [EZ:Web33, EZ:Web34]

1.2.12 Bipartite Graphen

Ein Graph ist *bipartit*, wenn eine der beiden folgenden Aussagen gilt:

- Der Graph ist azyklisch
- Für jeden Zyklus mit der Knotenfolge $(v_1, v_2, \ldots, v_n, v_1) \in V$ gilt $n \equiv 0 \pmod 2$. Es gilt also für keinen Zyklus $n \equiv 1 \pmod 2$.

Ist ein Graph bipartit, kann man seine Knoten in zwei disjunkte Teilmengen aufteilen, sodass zwischen den Knoten innerhalb beider Teilmengen keine Kanten verlaufen. Damit das Ganze etwas greifbarer ist, ist die genannte Bedingung in *Abbildung 1.12* mithilfe eines Beispiels für solch einen Graphen verdeutlicht. Der gezeigte Graph ist nicht *vollständig bipartit*, da nicht jeder Knoten der Teilmenge U eine Kante zu jedem Knoten der Teilmenge V hat. Ein Beispiel für einen Graphen, der diese Bedingung erfüllt und somit vollständig bipartit ist, ist in *Abbildung 1.13* veranschaulicht. [EZ:Web12, EZ:Web19]

1.2.13 Grids

Neben Graphen werden für Pathfinding häufig auch Grids verwendet, da diese für einige Anwendungsfälle besser geeignet sind, da es keine vordefinierten Kanten gibt. Ein Grid kann als Sonderfall eines Graphen betrachtet werden, bei dem die Knoten in gleichmäßigen Abständen platziert sind und jeder Knoten Kanten zu allen Knoten hat, die ihn umgeben, sofern diese nicht von Hindernissen oder Ähnlichem blockiert

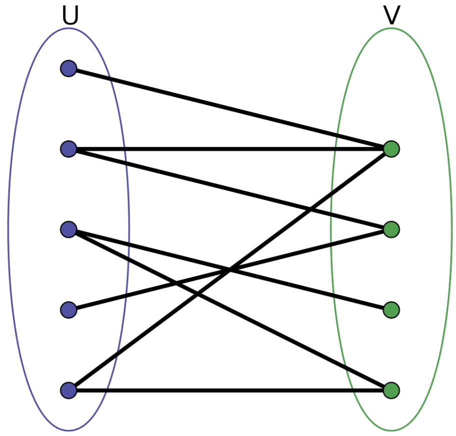

Abbildung 1.12: Ein einfacher, nicht vollständig bipartiter Graph mit Partitionsklassen U und V [EZ:Web19]

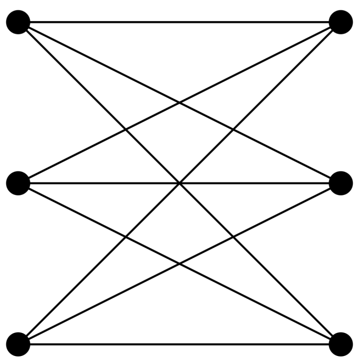

Abbildung 1.13: Ein einfacher, vollständig bipartiter Graph [EZ:Web19]

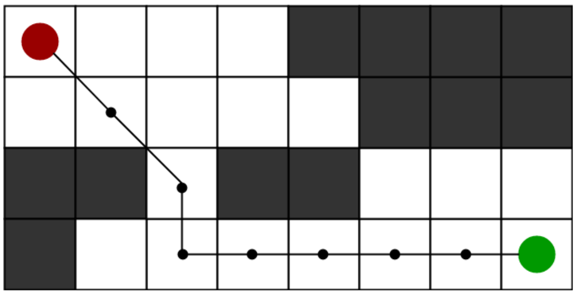

Abbildung 1.14: Beispiel für Pathfinding auf einem Grid
[EZ:Web02]

sind. Ein Beispiel für Pathfinding auf einem Grid ist in *Abbildung 1.14* zu sehen. Hier ist der rote Kreis der Startpunkt und der grüne der Zielpunkt. Die dunklen Zellen stellen Hindernisse dar, die der Pfad vermeiden muss. In diesem spezifischen Fall sind diagonale Schritte erlaubt, weshalb der kürzeste Weg zwei Diagonalen beinhaltet. Stellt man das Grid als Graph dar, existieren entweder keine Kanten zu den blockierten Knoten oder die Hindernisse werden erst gar nicht als Knoten dargestellt, sondern schlicht und ergreifend verworfen.

1.2.14 Darstellung

Adjazenzliste

Mithilfe einer *Adjazenzliste* kann ein Graph mitsamt dessen Knoten und Kanten dargestellt werden. Sie ist eine einfache Auflistung aller Nachbarknoten für jeden Knoten, wobei mit Nachbarknoten alle Knoten gemeint sind, zu denen ein Knoten eine Kante hat. Will man einen gewichteten Graphen als Adjazenzliste darstellen, speichert man gemeinsam mit jeder Kante ihre zugehörige Gewichtung ab. Adjazenzlisten eignen sich gut für gerichtete Graphen, da man mit ihnen nur die von Knoten *ausgehenden* Kanten beschreibt, nicht die eingehenden. Für ungerichtete Graphen ergeben sich somit zwei Möglichkeiten sie als Adjazenzliste abzuspeichern:

1. Man speichert jede ungerichtete Kante $\{v_i, v_j\}$ als zwei gegenläufige gerichtete Kanten ab: bei Knoten v_i als (v_i, v_j) und bei Knoten v_j als (v_j, v_i). Dadurch ist es egal, auf welcher Seite man auf die Existenz einer Kante prüft. Ein Vorteil dieser Methode ist, dass man garantiert immer nur einen Kanten-Check durchführen muss und man deshalb auch nicht klarstellen muss, ob es sich um einen gerichteten oder einen ungerichteten Graphen handelt, solange man den Graphen nicht mehr verändert. Man kann bei Anwendung dieser Methode eine Art

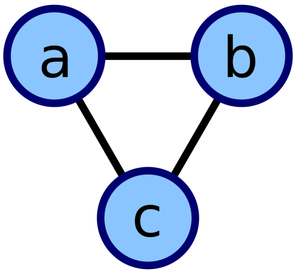

Abbildung 1.15: C_3 beziehungsweise K_3, der kleinste Dreiecksgraph [EZ:Web31]

Knoten	adjazent mit
a	b, c
b	a, c
c	a, b

Tabelle 1.2: Adjazenzliste

Mischung aus einem gerichteten und einem ungerichteten Graphen darstellen, da gerichtete Kanten nur bei ihrem Startknoten abgespeichert werden müssen. Der klare Nachteil ist der doppelte Speicherverbrauch für ungerichtete Kanten.

2. Man speichert jede ungerichtete Kante $\{v_i, v_j\}$ nur einmal ab, entweder bei v_i oder bei v_j, wodurch weniger Speicherplatz beansprucht wird. Prüft man jedoch von Knoten v_i ausgehend, ob zum Knoten v_j eine Kante (v_i, v_j) existiert und stellt fest, dass das nicht der Fall ist, so muss auch auf die Existenz der gegenläufigen Kante (v_j, v_i) geprüft werden. Geht man davon aus, dass man in durchschnittlich 50% der Fälle vom Startknoten und in 50% der Fälle vom Endknoten der zu überprüfenden Kante ausgehend prüft, so müssen in der Hälfte aller Fälle zwei Checks durchgeführt werden, was bedeutet, dass durchschnittlich $0.5 \cdot 1 + 0.5 \cdot 2 = 1.5$ Checks durchgeführt werden müssen, was ein klarer Nachteil ist. Damit die Existenz der gegenläufigen Kante (v_j, v_i) auch wirklich geprüft wird, falls (v_i, v_j) nicht existiert, muss klargestellt werden, dass es sich um einen ungerichteten Graphen handelt. Durch diesen Nachteil ergibt sich auch schon der nächste: Es gibt bei dieser Methode keine Möglichkeit, gerichtete Kanten darzustellen, sie kann also nur bei rein ungerichteten Graphen eingesetzt werden.

Tabelle 1.2 stellt die Adjazenzliste für den Graphen, der in *Abbildung 1.15* zu sehen ist, dar. Der verwendete Graph ist ungerichtet und zur Darstellung wird Methode 1 angewendet. [EZ:Web31, EZ:Web40]

Case	Adjacency list	Adjacency matrix
Average	$O(\lvert V\rvert + \lvert E\rvert)$	$O(\lvert V\rvert^2)$
Worst	$O(\lvert V\rvert^2)$	$O(\lvert V\rvert^2)$

Tabelle 1.3: Die Platzkomplexitäten der Adjazenzliste und -matrix [EZ:Web40]

	a	b	c
a	0	1	1
b	1	0	1
c	1	1	0

Tabelle 1.4: Adjazenzmatrix des Graphen aus *Abbildung 1.15*

Adjazenzmatrix

Adjazenzmatrizen bieten eine weitere Möglichkeit, einen Graphen $G = (V, E)$ darzustellen. Eine Adjazenzmatrix A ist eine quadratische $|V| \times |V|$-Matrix. Sie speichert in jedem Element $A_{i,j}$ die Gewichtung der Kante zwischen den Knoten v_i und v_j. Wird sie an einem ungewichteten Graphen angewendet, wird eine 1 gespeichert, falls es eine Kante zwischen den beiden Knoten gibt. Verläuft keine Kante durch v_i und v_j, speichert sie den Wert 0 oder ∞. Für gewichtete Graphen eignet sich ∞ besser, da 0 in manchen Fällen eine sinnvolle Gewichtung für Kanten sein kann. Adjazenzmatrizen haben den Vorteil, dass sie übersichtlich und, wie Adjazenzlisten, gut für gerichtete Graphen geeignet sind. Ihr großer Nachteil ist jedoch, dass es viele redundante Werte gibt, da Kanten, die nicht existieren, nicht abgespeichert werden müssten. Adjazenzmatrizen haben im Durchschnittsfall eine quadratische Platzkomplexität, wie in Tabelle 1.3 angeführt. Aus diesem Grund sind Adjazenzmatrizen nur für eher kleine Graphen geeignet. In Tabelle 1.4 ist die Adjazenzmatrix für den Graphen aus *Abbildung 1.15* zu sehen. [EZ:Web36, EZ:Web37, EZ:Web41, EZ:Web42, EZ:Web43]

1.2.15 Implementierung

Für fertige Implementierungen gibt es zahlreiche Libraries, wie z.B. *NetworkX* für Python oder *JGraphT* für Java. Eine simple, generische Graph-Klasse könnte in Java in etwa wie in *Listing 1.1* aussehen. Die Klasse ist generisch, da Graphen eine Unzahl an Anwendungsfällen haben. Zum Beispiel können neben Straßennetzen und vielem mehr auch Freundschaften in einem sozialen Netzwerk dargestellt und analysiert werden. In diesem Fall könnte man für den Typparameter `T` beispielsweise eine `Person` verwenden, oder `String` für den Namen. Sind die Namen der Personen nicht eindeutig, können Klassen wie `Integer`, `Long` oder `UUID` als Identifikator verwendet werden.

[EZ:Web21, EZ:Web45, EZ:Web46]

In der Variable `Map<T, Map<T, Double>> adjacencies` werden sowohl die Knoten als auch die von ihnen ausgehenden Kanten gespeichert, indem jedem Knotenwert in einer `HashMap` eine weitere `HashMap` zugeordnet wird, die jedem Nachbarn des Knoten, zu dem die `HashMap` gehört, die Gewichtungen der jeweiligen Kanten als `Double` zuordnet. Der soeben beschriebene Ansatz, einen Graphen zu speichern, ist eine Form der in *Kapitel 1.2.14* beschriebenen Adjazenzliste, mit dem lediglichen Zusatz der Kantengewichtungen. Für eine Implementierung eines ungewichteten Graphen wäre eine `Map<T, List<T>>` völlig ausreichend, jedoch ist der Zweck dieser Klasse, möglichst viele Arten von Graphen darstellen zu können.

Die Klasse ist mit einigen Lombok-Annotations ausgestattet, wie z.B. `@ToString` über der Klassendefinition für die automatische Generierung einer `toString`-Methode, die einen `String` zurückgibt, der den Namen der Klasse sowie sämtliche Instanzvariablennamen und -werte von dieser enthält. Außerdem sind die Variablen `boolean directed` und `Map<T, Map<T, Double>> adjacencies` mit `@Getter` annotiert, damit für diese automatisch Getter mit den Namen `isDirected` und `getAdjacencies` erstellt werden.

Dem Konstruktor des Graphen wird ein `boolean directed` übergeben, der angibt, ob der Graph gerichtet ist, oder nicht. Ist er ungerichtet, wird für jede hinzugefügte Kante eine gegenläufige Kante, also mit `source` und `destination` vertauscht, abgespeichert. Es kommt also die erste der beiden in *Kapitel 1.2.14* genannten Möglichkeiten zur Darstellung für ungerichtete Graphen als Adjazenzliste zum Einsatz. Ist der Graph ungewichtet, kann der Parameter `double weight` der Methode `addEdge` weggelassen werden, wodurch hinzugefügten Kanten eine defaultmäßige Gewichtung von 1 zugeteilt wird. Da außerdem in `getEdgeWeight` die Methode `Map::getOrDefault` mit dem Defaultwert ∞ verwendet wird, ist `getEdgeWeight` eine Mischung der beiden in *Kapitel 1.2.2* erwähnten Kantengewichtungsfunktionen.

Es sind auch einige Hilfsmethoden vorhanden, um den Graphen zu modifizieren oder Informationen über ihn auszulesen. In `addVertex` wird `Map::computeIfAbsent` verwendet, weil

1. mit `Map::put` die von dem Knoten ausgehenden Kanten mit einer `new HashMap<>()` überschrieben werden würden,
2. mit `Map::putIfAbsent` unnötig eine `new HashMap<>()` erzeugt werden würde,

wenn bereits ein Knoten mit dem übergebenen Wert existiert. Da `Map::<K, V>put` und `Map::<K, V>putIfAbsent` als zweiten Parameter ein `V` erwarten, welches dem Value entspricht, `Map::<K, V>computeIfAbsent` aber eine `Function<K, V>`, die den Value zurückgibt, wird die `new HashMap<>()` mit `Map::computeIfAbsent` nur dann erzeugt, wenn der einzufügende Key noch nicht in der Map vorhanden ist. Aus diesem Grund sind die Methoden `addEdge`, `addVertex` und `addVertices` *idempotent*, was bedeutet, dass sich der Zustand des Graphen, wenn eine der genannten Methoden bereits einmal aufgerufen wurde, nach erneutem Aufrufen mit denselben Parametern nicht mehr verändert. Somit ermöglicht diese Implementierung weder mehrere Knoten mit demselben Wert noch Multikanten, eine Schlinge pro Knoten ist jedoch möglich. Die restlichen Methoden sind ebenfalls idempotent, jedoch ist diese Eigenschaft bei `get`- und `remove` -Methoden weniger besonders. [EZ:Web30]

```java
package pathfinding.graphs;

import lombok.Getter;
import lombok.ToString;

import java.util.*;

@ToString
public class Graph<T> {

    @Getter
    private final boolean directed;

    @Getter
    private final Map<T, Map<T, Double>> adjacencies = new HashMap<>();

    /**
     * Undirected graph constructor.
     */
    public Graph() {
        this(false);
    }

    /**
     * Graph constructor.
     *
     * @param directed whether the graph is directed or not
     */
    public Graph(boolean directed) {
        this.directed = directed;
    }

    /**
     * Adds an unweighted edge between two vertices.
     *
     * @param source      the source vertex of the edge
     * @param destination the destination vertex of the edge
     */
    public void addEdge(T source, T destination) {
        addEdge(source, destination, 1);
    }
```

```java

    /**
     * Adds a weighted edge between two vertices with a weight.
     *
     * @param source      the source vertex of the edge
     * @param destination the destination vertex of the edge
     * @param weight      the weight of the edge
     */
    public void addEdge(T source, T destination, double weight) {
        addVertex(source);
        addVertex(destination);
        adjacencies.get(source).put(destination, weight);

        if (!directed) {
            adjacencies.get(destination).put(source, weight);
        }
    }

    /**
     * Adds a Collection of vertices to the graph.
     */
    public void addVertices(Collection<T> vertices) {
        for (T vertex : vertices) {
            addVertex(vertex);
        }
    }

    /**
     * Adds a vertex to the graph.
     *
     * @param vertex the value of the vertex to be added
     */
    public void addVertex(T vertex) {
        adjacencies.computeIfAbsent(vertex, key -> new HashMap<>());
    }

    /**
     * Removes an edge between two vertices.
     *
     * @param source      the source vertex of the edge to be removed
     * @param destination the destination vertex of the edge to be removed
     */
    public void removeEdge(T source, T destination) {
        adjacencies.get(source).remove(destination);

        if (!directed) {
            adjacencies.get(destination).remove(source);
        }
    }

    /**
```

```
     * Removes a vertex from the graph.
     *
     * @param vertex the value of the vertex to be removed
     */
    public void removeVertex(T vertex) {
        adjacencies.remove(vertex);
    }

    /**
     * @return the number of edges in the graph
     */
    public int getEdgeCount() {
        int count = adjacencies
                .values()
                .stream()
                .mapToInt(Map::size)
                .sum();

        return (directed) ? count : count / 2;
    }

    /**
     * @return the number of vertices in the graph
     */
    public int getVertexCount() {
        return adjacencies.size();
    }

    /**
     * @param source      the source vertex of the edge to be checked for
     * @param destination the destination vertex of the edge to be checked
         for
     * @return the weight of the edge between the two vertices
     */
    public boolean hasEdge(T source, T destination) {
        return adjacencies.get(source).containsKey(destination);
    }

    /**
     * @param vertex the vertex to be checked for
     * @return the weight of the edge between the two vertices
     */
    public boolean hasVertex(T vertex) {
        return adjacencies.containsKey(vertex);
    }

    /**
     * @param path the path to calculate the total weight of
     * @return the accumulated weight of the path
     */
    public double sumEdgeWeights(List<T> path) {
```

```java
        double weight = 0;

        for (int i = 0; i < path.size() - 1; i++) {
            weight += getEdgeWeight(path.get(i), path.get(i + 1));
        }

        return weight;
    }

    /**
     * @param source      the source vertex of the edge
     * @param destination the destination vertex of the edge
     * @return the weight of the edge between the two vertices
     */
    public double getEdgeWeight(T source, T destination) {
        return adjacencies.get(source).getOrDefault(destination, Double.POSITIVE_INFINITY);
    }

    /**
     * @return a Set of all the vertices in the graph
     */
    public Set<T> getVertices() {
        return adjacencies.keySet();
    }

    /**
     * @param vertex the vertex to get the neighbors of
     * @return a Map of the neighbors of the vertex and the weights of the edges between them
     */
    public Map<T, Double> getNeighbors(T vertex) {
        return adjacencies.get(vertex);
    }

}
```

Listing 1.1: Implementierung einer Graph-Klasse in Java

Ein Anwendungsbeispiel dieser Klasse ist in *Listing 1.2* demonstriert. Der erzeugte Graph kann wie in Tabelle 1.5 als eine Art gewichtete Adjazenzliste dargestellt werden.

```java
package pathfinding;

import pathfinding.service.GraphRandomizer;

import java.util.List;

public class GraphExample {

    public static void main(String[] args) {
```

Knoten	Adjazenzen
A	B=1
B	C=3
C	E=7
D	A=8, C=1
E	A=5, B=3, C=2, D=5

Tabelle 1.5: Gewichtete Adjazenzliste des generierten Graphen (*siehe Listing 1.2*)

```
        var graphRandomizer = GraphRandomizer.<Character>builder()
                .maxRandomWeight(10)
                .vertices(List.of('A', 'B', 'C', 'D', 'E'))
                .build();

        var graph = graphRandomizer.randomizeDirectedEdges();
        System.out.println(graph);
    }

}
```

Listing 1.2: Beispielanwendung der Graph-Klasse (*siehe Listing 1.1*)

Die verwendete `GraphRandomizer`-Klasse ist in *Listing 1.3* definiert. Das Builder-Pattern wird hier angewendet, um ein übersichtlicheres Erzeugen von neuen `GraphRandomizer` -Instanzen zu ermöglichen. Zur automatischen Generierung des Builders für diese Klasse wird die Lombok-Annotation `@Builder` verwendet, die eine `builder()`-Methode zur Verfügung stellt, die den Builder zurückgibt. Die Annotation `@Builder.Default` sorgt dafür, dass die den Variablen zugewiesenen Defaultwerte im Builder übernommen werden und dadurch nicht immer alle Instanzvariablen selbst gesetzt werden müssen. Zudem ist die Klasse mit `@Getter` und `@Setter` annotiert, wodurch automatisch für alle Instanzvariablen Getter und für alle `non-final` Instanzvariablen Setter erzeugt werden. [EZ:Web44]

Der `DoubleUnaryOperator weightMapping` dient dazu, die Zufallsvariable für die Gewichtungen beliebig zu modifizieren, bevor sie einer Kante zugewiesen wird. Defaultmäßig wird hier die Methode `Math::round` verwendet, um Nachkommastellen zu vermeiden und den Wert `maxRandomWeight` zu inkludieren, falls die Zufallszahl im Intervall $[\mathtt{maxRandomWeight} - 0.5, \mathtt{maxRandomWeight}[$ liegt. Die Klasse `DoubleUnaryOperator` ist funktional identisch mit `UnaryOperator<Double>` und `Function<Double, Double>`. Der größte Unterschied besteht darin, dass `DoubleUnaryOperator` intern den primitiven Datentyp `double` verwendet, während die generischen Klassen die Wrapperklasse `Double` verwenden. So wird dem Computer mit dem Einsatz von `DoubleUnaryOperator` ein wenig Arbeit erspart, da der Rechenaufwand von Boxing und Unboxing wegfällt.

In der Methode `randomizeUndirectedEdges` fängt die Laufvariable `int j` der inneren `for`-Schleife nicht bei 0 an, weil der Graph ungerichtet ist und ansonsten jede Kante *zwei* Chancen bekäme, zu entstehen. Somit wäre die Wahrscheinlichkeit, dass zwei Knoten miteinander verbunden werden, nicht mehr `edgeProbability`, sondern $1-(1-p)^2$, wobei $p =$ `edgeProbability`. Man könnte annehmen, dass sich die Wahrscheinlichkeit verdoppelt, oder allgemeiner, dass die Wahrscheinlichkeit p_n, dass zwei Knoten nach n Versuchen miteinander verbunden wurden, np beträgt. Dies kann jedoch durch eine einfache *Reductio ad absurdum* widerlegt werden: Angenommen $p = 1$ und $n = 2$, so wäre $p_2 = 2p = 2$, also gäbe es eine 200%ige Chance, dass sich zwei Knoten miteinander verbinden, was unmöglich ist. Die tatsächliche Formel für p_n kann wie folgt hergeleitet werden: Ist p die Wahrscheinlichkeit, dass bei einem Versuch eine Kante zwischen zwei Knoten entsteht, so ist $1-p$ die Gegenwahrscheinlichkeit, also die Wahrscheinlichkeit, dass keine Kante erzeugt wird. Somit ist $(1-p)^n$ die Wahrscheinlichkeit, dass bei keinem von n Versuchen eine Kante erstellt wird. Um die Wahrscheinlichkeit zu errechnen, dass die zwei Knoten bei mindestens einem der n Versuche miteinander verbunden werden, muss man ein weiteres mal die Gegenwahrscheinlichkeit berechnen, indem man $(1-p)^n$ von 1 subtrahiert. Man erhält die allgemeine Formel $p_n = 1-(1-p)^n$. Setzt man $n = 2$ in diese Formel ein, findet man heraus, dass $p_2 = 1-(1-p)^2$. Würde man also z.B. $p = 0.5$ mit dem fehlerhaften Code verwenden, so wäre die tatsächliche Kantenwahrscheinlichkeit $p_2 = 1-(1-0.5)^2 = 1-0.5^2 = 1-0.25 = 0.75 = 75\%$, anstatt den eigentlich gewollten 50%. [EZ:Web38]

Im Gegensatz zu `randomizeUndirectedEdges` kommen bei `randomizeDirectedEdges` `for-each`-Schleifen zum Einsatz, da der Index für das zufällige Erzeugen von gerichteten Kanten nicht benötigt wird, weil die Reihenfolge der Knotenpaare bei gerichteten Graphen einen Unterschied macht, weshalb keine Kante mehr als eine Chance bekommt, zu entstehen. Damit Schlingen vermieden werden, muss nur überprüft werden, ob `source != destination`, ansonsten wird keine Kante erzeugt. Aus demselben Grund beginnt `j` in `randomizeUndirectedEdges` bei `i + 1`, anstatt bei `i`.

```java
package pathfinding.service;

import lombok.Builder;
import lombok.Getter;
import lombok.Setter;
import pathfinding.graphs.Graph;

import java.util.List;
import java.util.Random;
import java.util.function.DoubleUnaryOperator;

/**
 * Helpful utility class for randomizing edges between vertices of a graph
 *
 * @param <T> the type of the value each vertex holds
 */
```

```java
@Builder
@Getter
@Setter
public class GraphRandomizer<T> {

    private static final Random RANDOM = new Random();

    /**
     * The probability of an edge being created between two vertices
     */
    @Builder.Default
    private double edgeProbability = 0.5;

    /**
     * The minimum value of the random value for the weight of an edge (
        inclusive)
     */
    @Builder.Default
    private double minRandomWeight = 1;

    /**
     * The maximum value of the random value for the weight of an edge (
        exclusive)
     */
    @Builder.Default
    private double maxRandomWeight = 1;

    /**
     * The function used to map the random weight to the actual weight
     */
    @Builder.Default
    private DoubleUnaryOperator weightMapping = Math::round;

    /**
     * The vertices of the graph
     */
    @Builder.Default
    private List<T> vertices = List.of();

    /**
     * Generates random edges between the vertices of a directed graph
     *
     * @return a new directed graph with randomized edges
     */
    public Graph<T> randomizeDirectedEdges() {
        var graph = createFilledGraph(true);

        for (var source : vertices) {
            for (var destination : vertices) {
                if (source != destination && random() < edgeProbability) {
                    graph.addEdge(source, destination, randomWeight());
```

```java
                }
            }
        }

        return graph;
    }

    /**
     * Generates random edges between the vertices of an undirected graph
     *
     * @return a new undirected graph with randomized edges
     */
    public Graph<T> randomizeUndirectedEdges() {
        var graph = createFilledGraph(false);

        for (int i = 0; i < vertices.size(); i++) {
            for (int j = i + 1; j < vertices.size(); j++) {
                if (random() < edgeProbability) {
                    var source = vertices.get(i);
                    var destination = vertices.get(j);
                    graph.addEdge(source, destination, randomWeight());
                }
            }
        }

        return graph;
    }

    private Graph<T> createFilledGraph(boolean directed) {
        var graph = new Graph<T>(directed);
        graph.addVertices(vertices);
        return graph;
    }

    private double random() {
        return RANDOM.nextDouble();
    }

    private double randomWeight() {
        double random = RANDOM.nextDouble(
                minRandomWeight,
                maxRandomWeight
        );

        return weightMapping.applyAsDouble(random);
    }

}
```

Listing 1.3: In *Listing 1.2* verwendeter GraphRandomizer

1.3 Klassische Pathfinding-Algorithmen

Pathfinding-Algorithmen können anhand verschiedenster Kriterien bewertet werden. Ein Algorithmus, der in einem Szenario gut funktioniert, muss nicht unbedingt in einem anderen Kontext die beste Wahl sein. Die Auswahl des für den Anwendungsfall richtigen Pathfinding-Algorithmus hängt vor allem von den folgenden drei Faktoren ab:

- **Laufzeit:** Die Zeitdauer, bis ein bzw. der kürzeste Pfad gefunden wurde.
- **Speicherbedarf:** Der Speicher, der im Prozess der Wegfindung eingenommen wird.
- **Skalierbarkeit:** Die Fähigkeit eines Algorithmus, effizient mit großen und komplexen Graphen umzugehen.

1.3.1 Hilfsklassen

PathfindingAlgorithm

Um die Implementierung mehrerer verschiedener Algorithmen zu vereinfachen und zu flexibilisieren gibt es das generische Interface `PathfindingAlgorithm<T>`, dessen Definition in *Listing 1.4* zu sehen ist. Es schreibt zwei Methoden vor: Die erste heißt `findAnyPath` und erwartet als Parameter ein `T start`, ein `T end` sowie einen `Graph<T> graph`. Der Parameter `start` stellt den Knoten dar, von dem der Pfad ausgehen soll, `end` den Knoten, zu dem er führen soll und `graph` den Graphen, in dem irgendein Pfad zwischen `start` und `end` gefunden werden soll. Die Methode `findAnyPath` gibt ein `Optional<List<T>>` zurück, wobei die `List<T>` eine Knotenfolge ist, welche den Pfad repräsentiert. Sie ist in einem `Optional` verpackt, für den Fall dass es zwischen Start- und Endknoten keinen Pfad gibt. Defaultmäßig gibt `findAnyPath` den Rückgabewert der zweiten Methode, `findShortestPath`, mit denselben Übergabeparametern zurück, da die meisten Pathfinding-Algorithmen nur diese Methode implementieren. Die Methode `findShortestPath` erwartet dieselben Parameter wie `findAnyPath` und hat denselben Rückgabetypen. Die beiden Methoden ähneln sich sehr, `findShortestPath` ist jedoch dafür zuständig, nicht nur irgendeinen Pfad zwischen zwei gegebenen Knoten zu berechnen, sondern den kürzesten. Die Methode `findAnyPath` existiert, da manche Algorithmen auch auf Arten implementiert werden können, welche nicht unbedingt den optimalen Pfad finden und dadurch performanter sind. Somit kann man also sagen, dass bei sinnhafter Implementierung die Laufzeit von `findAnyPath` $\leq$ der Laufzeit von `findShortestPath` ist.

```java
package pathfinding.algorithms;

import pathfinding.graphs.Graph;

import java.util.List;
import java.util.Optional;

public interface PathfindingAlgorithm<T>{

    /**
     * Searches any path between two given vertices in a graph.
     *
     * @param start the vertex the path starts at
     * @param end the vertex the path ends at
     * @param graph the graph in which a path is to be found
     * @return Any path between the two vertices, or <code>Optional.empty</code> if no path exists.
     */
    default Optional<List<T>> findAnyPath(T start,
                                          T end,
                                          Graph<T> graph) {
        return findShortestPath(start, end, graph);
    }

    /**
     * Searches the shortest path between two given vertices in a graph.
     *
     * @param start the vertex the path starts at
     * @param end   the vertex the path ends at
     * @param graph the graph in which the shortest path is to be found
     * @return The shortest path between the two vertices, or <code>Optional.empty</code> if no path exists.
     */
    Optional<List<T>> findShortestPath(T start,
                                       T end,
                                       Graph<T> graph);


}
```

Listing 1.4: `PathfindingAlgorithm.java`

PathTracer

Die Klasse **`PathTracer<T>`** aus *Listing 1.5* ist eine Hilfsklasse, mit der aus einer gegebenen Map von Knotenvorgängern ein Pfad rekonstruiert werden kann. Sie hat nur eine Instanzvariable, die **`Map<T, T> predecessors`** und zwei Methoden, **`safeTrace`** und `unsafeTrace`.

Die `safeTrace`-Methode verwendet intern ein `LinkedHashSet<T>`, und das aus zwei Gründen:

1. `Set`, um mehrfach vorkommende Knoten zu erkennen und dadurch festzustellen, dass man in einem Zyklus gefangen ist.
2. `LinkedHashSet`, damit die Einfügereihenfolge beibehalten wird und der Pfad am Ende fehlerfrei rekonstruiert werden kann.

Zu Beginn wird der aktuelle Knoten in das Set eingefügt, der zu diesem Zeitpunkt noch der Endknoten des Pfades ist. In einer `while`-Schleife fügt man solange den Vorgänger des aktuellen Knoten zum Set hinzu und setzt danach den aktuellen Knoten auf den Vorgänger, bis man am Startknoten angelangt ist. Ist ein Knoten bereits im Set vorhanden, wird eine `CycleException` geworfen, *siehe Listing 1.6*. Da der Pfad in umgekehrter Reihenfolge in das Set eingefügt wird, wird der Pfad am Ende umgedreht und als `List<T>` zurückgegeben, damit einzeln auf Elemente an beliebigen Indizes zugegriffen werden kann.

Die Methode `unsafeTrace` bietet eine effizientere Alternative zu `safeTrace`, da als Datenstruktur anstatt eines `LinkedHashSet`, welches für seine Ineffizienz berühmt-berüchtigt ist, eine `ArrayList` verwendet wird. Es wird jedoch nicht auf Duplikate überprüft und somit könnte man sich in einem Zyklus verfangen, daher das *unsafe* im Namen. Da viele Pathfinding-Algorithmen so implementiert sind, dass Zyklen vermieden werden, kann `unsafeTrace` in den meisten Fällen ohne Bedenken eingesetzt werden.

```java
package pathfinding.service;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import pathfinding.exceptions.CycleException;

import java.util.ArrayList;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Map;

/**
 * Utility class for reconstructing paths from a given map of predecessors.
 *
 * @param <T> the type of the nodes in the map
 */
@AllArgsConstructor
@Getter
```

```java
@Setter
public class PathTracer<T> {

    private Map<T, T> predecessors;

    /**
     * Traces the path from start to end in a given map of predecessors.
     * <p>
     * This method is less safe than {@link #safeTrace(T, T)}
     * because it doesn't check for cycles in the path and it is possible
     * to get stuck in an infinite loop. However, it is more efficient
     * because it uses an {@link ArrayList} as data structure.
     *
     * @param start the node the path starts at
     * @param end   the node the path ends at
     * @return a list of nodes that form the path from start to end
     */
    public List<T> unsafeTrace(T start, T end) {
        var path = new ArrayList<T>();
        var current = end;
        path.add(current);

        while (!current.equals(start)) {
            current = predecessors.get(current);
            path.addFirst(current);
        }

        return path;
    }

    /**
     * Traces the path from start to end in a given map of predecessors.
     * <p>
     * This method is more safe than {@link #unsafeTrace(T, T)}
     * because it checks for cycles in the path. However, it is
     * less efficient because it uses a {@link LinkedHashSet}.
     *
     * @param start the node the path starts at
     * @param end   the node the path ends at
     * @return a list of nodes that form the path from start to end
     * @throws CycleException if the end node is unreachable because of a
         cycle
     */
    public List<T> safeTrace(T start, T end) {
        var path = new LinkedHashSet<T>();
        var current = end;
        path.add(current);

        while (!current.equals(start)) {
            current = predecessors.get(current);

```

```java
            if (!path.add(current)) {
                throw new CycleException(
                        "The end node is unreachable because of a cycle."
                );
            }
        }

        return path.reversed()
                .stream()
                .toList();
    }

}
```

Listing 1.5: `PathTracer.java`

CycleException

Die RuntimeException `CycleException` aus *Listing 1.6* sollte dann geworfen werden, wenn das Programm in einem Zyklus feststeckt und sich nicht mehr befreien kann. Das kann im durchaus passieren: Enthält ein Pfad einen Zyklus und wird von einem `PathTracer` rekonstruiert, ist diese Fehlermeldung garantiert.

```java
package pathfinding.algorithms;

/**
 * An exception thrown when the program detects that it
 * got stuck in a cycle during the traversal of a graph.
 */
public class CycleException extends RuntimeException {

    public CycleException() {
        super();
    }

    public CycleException(String message) {
        super(message);
    }

}
```

Listing 1.6: `CycleException.java`

1.3.2 Breitensuche

Die Breitensuche, abgekürzt mit BFS[2], ist ein Suchalgorithmus für Graphen, der unter Umständen auch dazu verwendet werden kann, den kürzesten Pfad zwischen zwei Knoten zu finden. Damit der Algorithmus in allen Fällen den Pfad mit der geringsten Gesamtgewichtung findet, muss der Graph, auf dem er angewendet wird, folgende Kriterien erfüllen:

1. Er ist ungewichtet oder alle Kanten haben die gleiche Gewichtung
2. Er ist nicht zyklisch

Ist das erste Kriterium nicht erfüllt, findet BFS den Pfad mit der geringsten Knotenanzahl, unabhängig von deren individuellen Gewichtungen. Das zweite Kriterium ist nur dann wichtig, wenn man keine Liste von Knoten führt, bei denen man bereits war, da man ansonsten in eine Endlosschleife geraten könnte. Speichert man jedoch die bereits behandelten Knoten ab, ist es völlig egal, ob der Graph zyklisch ist, oder nicht.

Die Breitensuche hat ihren Namen von ihrer Funktionsweise: Für jeden Knoten in einer Queue, welche anfangs nur den Startknoten enthält, wird überprüft, ob er mit dem Zielknoten übereinstimmt. Außerdem werden all seine Nachbarn, die noch nicht besucht wurden und noch nicht in der Queue sind, in diese hinzugefügt. Da Queues auf dem FIFO[3]-Prinzip basieren, wird also in die Breite gesucht.

Eine Implementierung von BFS ist in *Listing 1.7* zu sehen. Die Streaming-API von Java kommt hier zum Einsatz, um den Code einfach und verständlich zu gestalten. Als Queue wird in diesem Beispiel die Klasse `ArrayDeque` verwendet, wobei Deque für *Double-ended queue steht*. Deques können somit sowohl als Queue als auch als Stack dienen. In der `ArrayList<T> visited` werden alle bereits besuchten Knoten abgespeichert, damit man keine Knoten öfter als einmal besucht und man nicht in Endlosschleifen gerät. In der `HashMap<T, T> predecessors` wird für jeden Knoten abgespeichert, bei welchem Knoten man war, als er in die Queue eingefügt wurde, damit man am Ende den Pfad wiederherstellen kann. Der `boolean found` ist am Ende der Methode wichtig, um feststellen zu können, ob die `while`-Schleife abgebrochen hat, weil der Zielknoten gefunden wurde, oder weil es keine unbesuchten Knoten mehr gibt. Die `while` -Schleife wird solange ausgeführt, bis keine Elemente mehr in der Queue vorhanden sind. Anfangs ist nur der Startknoten in der Queue, woraufhin all seine Nachbarknoten an das Ende dieser hinzugefügt werden. Bei der nächsten Iteration werden dann die Nachbarn der Nachbarn hinzugefügt und so weiter, bis man am Zielknoten angelangt

[2] Breadth-First Search
[3] First In - First Out

ist. Am Ende wird der Pfad mithilfe der **`unsafeTrace`**-Methode der **`PathTracer`**-Klasse wiederhergestellt, *siehe Listing 1.5*. Es wird **`unsafeTrace`** bevorzugt, da es viel performanter ist als **`safeTrace`** und nicht auf Zyklen überprüft werden muss. Diese Aufgabe wird bei der Breitensuche schon von der **`ArrayList<T> visited`** übernommen. *siehe Kapitel 1.3.1*.

```java
package pathfinding.algorithms;

import pathfinding.graphs.Graph;
import pathfinding.service.PathTracer;

import java.util.*;

/**
 * Iterative implementation of the Breadth-First Search algorithm.
 *
 * @param <T> the type of the nodes in the graph to be searched
 */
public class BreadthFirstSearch<T> implements PathfindingAlgorithm<T> {

    @Override
    public Optional<List<T>> findShortestPath(T start,
                                              T end,
                                              Graph<T> graph) {
        var queue = new ArrayDeque<T>();
        var visited = new ArrayList<T>();
        var predecessors = new HashMap<T, T>();
        boolean found = false;
        queue.add(start);

        while (!queue.isEmpty()) {
            var current = queue.poll();
            visited.add(current);

            if (current.equals(end)) {
                found = true;
                break;
            }

            graph.getNeighbors(current)
                    .keySet()
                    .stream()
                    .filter(neighbor -> !visited.contains(neighbor))
                    .filter(neighbor -> !queue.contains(neighbor))
                    .forEach(neighbor -> {
                        queue.offer(neighbor);
                        predecessors.put(neighbor, current);
                    });
        }

        if (found) {
```

```
            var pathTracer = new PathTracer<>(predecessors);
            return Optional.of(pathTracer.unsafeTrace(start, end));
        } else {
            return Optional.empty();
        }
    }

}
```

Listing 1.7: BFS-Algorithmus in Java

1.3.3 Tiefensuche

Die Tiefensuche, abgekürzt mit DFS[4], ist ein weiterer Suchalgorithmus für Graphen, welcher auch für Pathfinding eingesetzt werden kann. Mit dem standardmäßigen DFS-Algorithmus findet man jedoch nicht den kürzesten Pfad, sondern irgendeinen. Vorteil ist, dass der Algorithmus schnell ist, er hat also eine vergleichsweise kurze Laufzeit, da er nicht darauf ausgelegt ist, den optimalen Pfad zu finden. Der offensichtliche Nachteil ist dadurch aber, dass der gefundene Pfad sehr lange sein kann und im Großteil der Fälle nicht der kürzeste ist.

Die Tiefensuche hat ihren Namen ebenfalls von ihrer Funktionsweise, die sehr ähnlich zur Breitensuche ist: Für jeden Knoten in einem Stack, welcher anfangs nur den Startknoten enthält, wird überprüft, ob er mit dem Zielknoten übereinstimmt. Es werden all seine Nachbarn, die noch nicht im Stack sind, in diesen hinzugefügt. Da Stacks auf dem LIFO[5]-Prinzip basieren, wird also in die Tiefe gesucht. Der einzige logische Unterschied zu BFS ist also die Verwendung eines Stacks anstatt einer Queue.

Iterativ

Eine iterative Implementierung von DFS ist in *Listing 1.8* in der Methode `findAnyPath` zu sehen. Die Streaming-API wird hier erneut angewendet. Wie bei BFS wird hier eine `ArrayDeque` als Datenstruktur verwendet, mit dem Unterschied, dass sie hier als Stack verwendet wird, anstatt als Queue. Es wird nicht die Klasse `Stack` verwendet, da diese thread-safe und aufgrund ihrer `synchronized`-Methoden sehr langsam ist. Da die gezeigte Implementierung single-threaded ist, stellt die Verwendung von Klassen und Methoden, welche nicht thread-safe sind, kein Problem dar.

[4] Depth-First Search
[5] Last In - First Out

Um garantiert den kürzesten Pfad zu finden, kann eine modifizierte Version von DFS angewendet werden, welche in `findShortestPath` implementiert ist. Diese ist jedoch extrem ineffizient, da alle Pfade durchsucht werden und erst dann der mit der geringsten Gewichtung zurückgegeben werden kann. Hier wird keine Liste von bereits besuchten Knoten geführt, da Knoten mehrmals besucht werden müssen, falls der Algorithmus einen Pfad gewählt hat, der nicht optimal ist, dieser Pfad jedoch Knoten des kürzesten Pfades enthält. Außerdem könnte der Zielknoten dann nur einmal besucht werden, was sinnlos ist, da es dann nur einen Pfad geben könnte, der mit hoher Wahrscheinlichkeit nicht der kürzeste wäre. Es reicht nicht, für den Zielknoten eine Ausnahme zu machen und ihn nicht in die Liste von besuchten Knoten aufzunehmen, da es für andere Knoten ebenso wichtig ist, häufiger besucht werden zu können, falls sie Teil des optimalen Pfades sind. Eine elegante Lösung für dieses Problem ist nicht zu überprüfen, ob ein Knoten bereits besucht wurde, sondern ob er Teil des aktuellen Pfades ist. Dies kann mithilfe der `predecessors`-Map in Kombination mit der `PathTracer`-Klasse erzielt werden, indem man von jedem Knoten aus den Pfad zum Startknoten rekonstruiert. Mit der Zeile `.filter(neighbor -> !path.contains(neighbor))` werden dann alle Nachbarknoten herausgefiltert, die bereits im aktuellen Pfad enthalten sind. Danach werden mit `.filter(neighbor -> !stack.contains(neighbor))` noch alle Nachbarn verworfen, die schon im Stack sind und nur noch darauf warten, besucht zu werden. Auch hier ist wie bei BFS der `boolean found` am Ende der Methode dazu da, um feststellen zu können, aus welchem Grund die `while`-Schleife abgebrochen hat.

```java
package pathfinding.algorithms;

import pathfinding.graphs.Graph;
import pathfinding.service.PathTracer;

import java.util.*;

/**
 * Iterative implementation of the Depth-First Search algorithm.
 * <p>
 * For the recursive implementation, see {@link RecursiveDFS}.
 * <p>
 * The internal workings are much more complex in this implementation,
 * but it is also much more performance and memory efficient for
 * large graphs. It does not suffer from stack overflow errors,
 * unlike the recursive implementation.
 *
 * @param <T> the type of the nodes in the graph to be searched
 */
public class DepthFirstSearch<T> implements PathfindingAlgorithm<T> {

    // TODO: check if this is actually faster than findShortestPath
    @Override
    public Optional<List<T>> findAnyPath(T start,
                                          T end,
                                          Graph<T> graph) {
```

```java
        var predecessors = new HashMap<T, T>();
        var stack = new ArrayDeque<T>();
        var visited = new ArrayList<T>();
        boolean found = false;
        stack.push(start);

        while (!stack.isEmpty()) {
            var current = stack.pop();
            visited.add(current);

            if (current.equals(end)) {
                found = true;
                break;
            }

            graph.getNeighbors(current)
                    .keySet()
                    .stream()
                    .filter(neighbor -> !visited.contains(neighbor))
                    .filter(neighbor -> !stack.contains(neighbor))
                    .forEach(neighbor -> {
                        stack.push(neighbor);
                        predecessors.put(neighbor, current);
                    });
        }

        if (found) {
            var pathTracer = new PathTracer<>(predecessors);
            return Optional.of(pathTracer.unsafeTrace(start, end));
        } else {
            return Optional.empty();
        }
    }

    @Override
    public Optional<List<T>> findShortestPath(T start,
                                              T end,
                                              Graph<T> graph) {
        var predecessors = new HashMap<T, T>();
        var stack = new ArrayDeque<T>();
        var paths = new ArrayList<List<T>>();
        var pathTracer = new PathTracer<>(predecessors);
        stack.push(start);

        while (!stack.isEmpty()) {
            var current = stack.pop();
            var path = pathTracer.unsafeTrace(start, current);

            if (current.equals(end)) {
                paths.add(path);
                continue;
```

```
            }

            graph.getNeighbors(current)
                    .keySet()
                    .stream()
                    .filter(neighbor -> !path.contains(neighbor))
                    .filter(neighbor -> !stack.contains(neighbor))
                    .forEach(neighbor -> {
                        stack.push(neighbor);
                        predecessors.put(neighbor, current);
                    });
        }

        return paths.stream()
                .min(Comparator.comparingDouble(
                        graph::sumEdgeWeights
                ));
    }

}
```

Listing 1.8: Iterativer DFS-Algorithmus in Java

Rekursiv

Die Tiefensuche kann nicht nur iterativ implementiert werden, sondern auch rekursiv, *siehe Listing 1.9*. Rekursive Implementierungen haben den Vorteil, dass sie viel sauberer und einfacher zu verstehen sind, da keine zusätzlichen Variablen wie `stack`, `predecessors`, `visited` (für `findAnyPath`), `paths` (für `findShortestPath`) oder andere benötigt werden. Bei rekursiven DFS-Implementierungen macht man sich den internen Callstack zunutze und verwendet ihn als Stack für die traversierten Knoten und Pfade. So ergibt sich der Nachteil, dass bei der Anwendung in gigantischen Graphen Stackoverflows auftreten können. Man muss hier den zurückgelegten Pfad nicht bei jedem Schritt rekonstruieren, sondern kann ihn einfach beim nächsttieferen Methodenaufruf übergeben und immer den aktuellen Knoten ans Ende anfügen. Wichtig ist hierbei jedoch, dass jeder Rekursionsbranch ein eigenes Listenobjekt erhält und somit keine Änderungen in anderen Branches bewirken kann. Darum kümmert sich die Methode `append`: Diese fügt nicht nur ein Element an das Ende einer gegebenen Liste an, sondern sorgt auch dafür, dass die übergebene Liste unverändert bleibt und eine Kopie von dieser erzeugt und mit dem hinzugefügten Element zurückgegeben wird.

Die Zeile `.filter(neighbor -> !path.contains(neighbor))` sorgt hier, wie bei der iterativen Variante, dafür, dass ausschließlich Knoten behandelt werden, die nicht bereits Teil des zurückgelegten Pfades sind. Danach wird von jedem Nachbarn des aktuellen Startknoten ausgehend irgendein bzw. der kürzeste Pfad zum Zielknoten gesucht und anschließend alle `Optional.empty()`-Pfade entfernt und die übrigen aus ihrem `Optional` entpackt. Bei `findAnyPath` wird zum Schluss nur noch `Stream::findAny` aufgerufen, wohingegen bei `findShortestPath` stattdessen `Stream::min` mit dem Comparator `Comparator.comparingDouble(graph::sumEdgeWeights)` aufgerufen und somit der optimale Pfad zurückgegeben wird.

```java
package pathfinding.algorithms;

import pathfinding.graphs.Graph;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

/**
 * Recursive implementation of the Depth-First Search algorithm.
 * <p>
 * For the iterative implementation, see {@link DepthFirstSearch}.
 * <p>
 * The interal workings are much simpler to understand in
 * this implementation, but it is prone to stack overflow errors
 * for large graphs.
 *
 * @param <T> the type of the nodes in the graph to be searched
 */
public class RecursiveDFS<T> implements PathfindingAlgorithm<T> {


    @Override
    public Optional<List<T>> findAnyPath(T start,
                                         T end,
                                         Graph<T> graph) {
        return findAnyPath(start, end, graph, List.of(start));
    }

    private Optional<List<T>> findAnyPath(T start,
                                          T end,
                                          Graph<T> graph,
                                          List<T> path) {
        if (start.equals(end)) {
            return Optional.of(path);
        }

        return graph.getNeighbors(start)
                .keySet()
                .stream()
                .filter(neighbor -> !path.contains(neighbor))
                .map(neighbor -> findAnyPath(
                        neighbor,
                        end,
```

```java
                        graph,
                        append(path, neighbor)
                ))
                .filter(Optional::isPresent)
                .map(Optional::get)
                .findAny();
    }

    @Override
    public Optional<List<T>> findShortestPath(T start,
                                              T end,
                                              Graph<T> graph) {
        return findShortestPath(start, end, graph, List.of(start));
    }

    private Optional<List<T>> findShortestPath(T start,
                                               T end,
                                               Graph<T> graph,
                                               List<T> path) {
        if (start.equals(end)) {
            return Optional.of(path);
        }

        return graph.getNeighbors(start)
                .keySet()
                .stream()
                .filter(neighbor -> !path.contains(neighbor))
                .map(neighbor -> findShortestPath(
                        neighbor,
                        end, graph,
                        append(path, neighbor)
                ))
                .filter(Optional::isPresent)
                .map(Optional::get)
                .min(Comparator.comparingDouble(graph::sumEdgeWeights));
    }

    private List<T> append(List<T> list, T element) {
        var appended = new ArrayList<>(list);
        appended.add(element);
        return appended;
    }

}
```

Listing 1.9: Rekursiver DFS-Algorithmus in Java

1.3.4 Dijkstra-Algorithmus

> *What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame.* [EZ:Web47, Edsger W. Dijkstra, 2001]

Der Algorithmus von Dijkstra wurde im Jahr 1956 von Edsger W. Dijkstra entworfen und drei Jahre später veröffentlicht. Er findet in vielen verschiedenen Bereichen Anwendung und das nicht nur bei der Navigation im Straßenverkehr, sondern zum Beispiel auch bei Routing-Protokollen für Computernetzwerke wie OSPF[6], welches auf dem Dijkstra-Algorithmus aufbaut. [EZ:Web48, EZ:Web49]

Die Gesamtkosten $f(n)$ vom Start- zum Zielknoten durch den Knoten n sind durch

$$f(n) = g(n)$$

definiert. $g(n)$ sind hier die tatsächlichen Kosten vom Startknoten zum Knoten n. Ein wichtiges Detail des Dijkstra-Algorithmus ist, dass er nicht nur den kürzesten Pfad zwischen zwei Knoten findet, sondern zu allen Knoten im Graphen von einem bestimmten Startpunkt aus. [EZ:Web08]

```java
package pathfinding.algorithms;

import pathfinding.graphs.Graph;
import pathfinding.service.PathTracer;

import java.util.*;
import java.util.stream.Collector;
import java.util.stream.Collectors;

/**
 * Implementation of the Dijkstra algorithm for finding the shortest path.
```

[6] Open Shortest Path First

```java
 * @param <T> the type of the vertices in the graph
 */
public class Dijkstra<T> implements PathfindingAlgorithm<T> {

    @Override
    public Optional<List<T>> findShortestPath(T start,
                                               T end,
                                               Graph<T> graph) {
        var distances = new HashMap<T, Double>();

        for (var vertex : graph.getVertices()) {
            distances.put(vertex, Double.POSITIVE_INFINITY);
        }

        distances.put(start, 0.0);
        var predecessors = new HashMap<T, T>();
        var queue = new PriorityQueue<T>(Comparator.comparingDouble(
            distances::get));
        queue.add(start);

        while (!queue.isEmpty()) {
            var current = queue.poll();

            graph.getNeighbors(current)
                    .forEach((neighbor, weight) -> {
                        var newDistance = weight + distances.get(current);

                        if (newDistance < distances.get(neighbor)) {
                            distances.put(neighbor, newDistance);
                            predecessors.put(neighbor, current);
                            queue.offer(neighbor);
                        }
                    });
        }

        var pathTracer = new PathTracer<>(predecessors);
        return Optional.of(pathTracer.unsafeTrace(start, end));
    }

}
```

Listing 1.10: Dijkstra-Algorithmus in Java

1.3.5 A*-Algorithmus

A* ist ein Algorithmus aus der Klasse der BeFS[7]-Algorithmen. Die geschätzten Gesamtkosten $f(n)$ vom Start- zum Zielknoten durch den Knoten n sind wie folgt definiert:

$$f(n) = g(n) + h(n)$$

$g(n)$ entspricht hier den tatsächlichen Kosten vom Startknoten zum Knoten n und $h(n)$ den geschätzten Kosten vom Knoten n zum Zielknoten, welche auch als *Heuristik* bezeichnet werden. Die Formel ähnelt der des Dijkstra-Algorithmus sehr, was daran liegt, dass der Dijkstra-Algorithmus als Sonderfall des A*-Algorithmus betrachtet werden kann. Bei Dijkstra gilt, im Gegensatz zu A*, immer $h(n) = 0$. [EZ:Web08, EZ:Web18]

1.4 Vergleich und Evaluation der Algorithmen (4 Seiten)

Damit man nicht bei jeder Pfadberechnung denselben Graphen und Algorithmus als Parameter übergeben muss, gibt es die generische Klasse `Pathfinder<T>`. Sie hat zwei Instanzvariablen, den `Graph<T> graph` und den `PathfindingAlgorithm<T> algorithm` . Will man die Methoden `findAnyPath` oder `findShortestPath` am an den Konstruktor von `Pathfinder<T>` übergebenen Graphen anwenden, müssen jetzt nur mehr Start- und Zielknoten übergeben werden, da immer der gespeicherte Graph verwendet wird. Sowohl der `Graph<T> graph` als auch der `PathfindingAlgorithm<T> algorithm` haben Getter und Setter, beide Variablen können also im Nachhinein noch verändert werden, falls nötig.

```java
package pathfinding.service;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import pathfinding.algorithms.PathfindingAlgorithm;
import pathfinding.graphs.Graph;

import java.util.List;
import java.util.Optional;

/**
 * Utility class so the graph doesn't have
```

[7] Best-first search (nicht zu verwechseln mit Breadth-first search)

```java
 * to be passed to the algorithm every time.
 * Also, the type T only has to be specified once,
 * as the objects not specifying it
 * can infer it from the object that specified it.
 */
@AllArgsConstructor
@Getter
@Setter
public class Pathfinder<T> {

    private Graph<T> graph;
    private PathfindingAlgorithm<T> algorithm;

    public Optional<List<T>> findAnyPath(T start, T end) {
        return algorithm.findAnyPath(start, end, graph);
    }

    public Optional<List<T>> findShortestPath(T start, T end) {
        return algorithm.findShortestPath(start, end, graph);
    }

}
```

Listing 1.11: `Pathfinder.java`

Da die Klasse **`Pathfinder<T>`** generisch ist, reicht es aus, den Typ **`T`** nur ein einziges mal zu spezifizieren, in diesem Fall bei der Erzeugung des Graphen, *siehe Listing 1.12*. Der Typ **`T`** des **`Pathfinder<T>`** kann hier durch Typinferenz bestimmt werden. Dasselbe gilt für den **`PathfindingAlgorithm<T>`** (hier **`BreadthFirstSearch<T>`**, *siehe Kapitel 1.3.2*), da er erst bei der Übergabe der Konstruktorparameter erzeugt wird. Als Beispiel wird hier der Graph aus *Abbildung 1.16* verwendet. Das **`print`**-Statement am Ende gibt wie erwartet den kürzesten Pfad von A zu D, **`Optional[[A, B, D]]`**, aus.

```java
var graph = new Graph<Character>();
graph.addEdge('A', 'B');
graph.addEdge('B', 'C');
graph.addEdge('C', 'D');
graph.addEdge('B', 'D');

var pathfinder = new Pathfinder<>(graph, new BreadthFirstSearch<>());
System.out.println(pathfinder.findShortestPath('A', 'D'));
```

Listing 1.12: Typinferenz

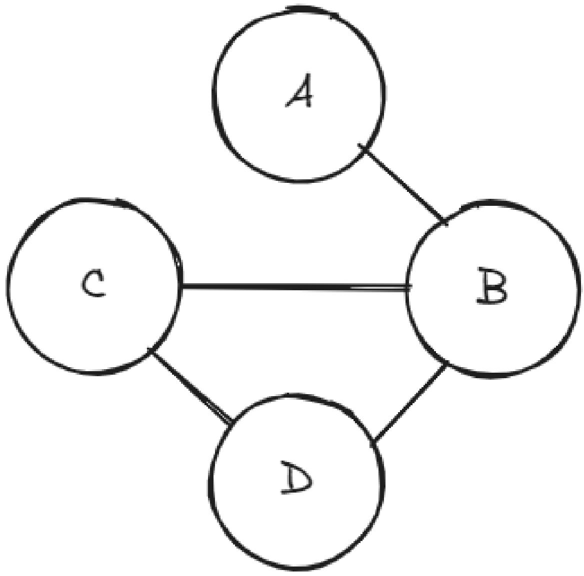

Abbildung 1.16: Graph mit zwei unterschiedlich langen Wegen von A zu D

1.4.1 Vergleichsparameter und Benchmarking (2 Seiten)

1.4.2 Zusammenfassung der Ergebnisse (2 Seiten)

1.4.3 Empfehlungen (2 Seiten)

1.5 Projektbezug (4 Seiten)

Bei unserem Projekt, bei dem es unsere Aufgabe ist, ein Modellauto selbstständig zwischen zwei Punkten navigieren zu lassen, ist Pathfinding ein wichtiges Konzept. Der Graph könnte mithilfe einer zweidimensionalen Punktwolke erstellt werden, die wir direkt aus einem LiDAR-Sensor auslesen. Die rohen LiDAR-Daten können unter anderem mittels SLAM[8] zu einem Graphen konvertiert werden.

[8] Simultaneous Localization and Mapping

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[EZ:Web01] https://www.ionos.at/digitalguide/online-marketing/web-analyse/pathfinding
Pathfinding: Wegfindung in der Informatik
03.11.2023

[EZ:Web02] https://www.geeksforgeeks.org/a-search-algorithm/
A*-Algorithm
28.10.2023

[EZ:Web03] https://arxiv.org/pdf/physics/0510162.pdf
Structural Properties of Planar Graphs of Urban Street Patterns
03.11.2023

[EZ:Web04] https://www.baeldung.com/cs/weighted-vs-unweighted-graphs
Weighted vs. Unweighted Graphs
28.10.2023

[EZ:Web05] https://happycoding.io/tutorials/libgdx/pathfinding
Pathfinding
04.11.2023

[EZ:Web06] https://studyflix.de/informatik/grundbegriffe-der-graphentheorie-1285
Grundbegriffe der Graphentheorie
13.11.2023

[EZ:Web07] https://blog.viking-studios.net/wp-content/uploads/2013/04/Pathfinding-Algorithmen-in-verschiedenen-Spielegenres.pdf
Pathfinding-Algorithmen in verschiedenen Spielegenres
14.11.2023

[EZ:Web08] https://yuminlee2.medium.com/a-search-algorithm-42c1a13fcf9f
A* Search Algorithm
14.11.2023

[EZ:Web09] http://gitta.info/Accessibilit/de/html/NetworkChara_learningObject1.html
Charakterisierung von Netzwerken
20.11.2023

[EZ:Web10] https://hyperskill.org/learn/step/5645
Weighted Graph
14.11.2023

[EZ:Web11] https://mathworld.wolfram.com/GraphCycle.html
Graph Cycle
20.11.2023

[EZ:Web12] https://www.geeksforgeeks.org/what-is-cyclic-graph/
What is Cyclic Graph
15.11.2023

[EZ:Web13] https://files.ifi.uzh.ch/cl/siclemat/lehre/hs07/ecl1/script/html/scriptse50.html
Graphen
16.11.2023

[EZ:Web14] https://mathworld.wolfram.com/CycleGraph.html
Cycle Graph
20.11.2023

[EZ:Web15] https://www.geeksforgeeks.org/degree-of-a-cycle-graph/
Degree of a Cycle Graph
16.11.2023

[EZ:Web16] https://en.wikipedia.org/wiki/Cycle_(graph_theory)
Cycle (graph theory)
16.11.2023

[EZ:Web17] https://de.wikipedia.org/wiki/Zyklus_(Graphentheorie)
Zyklus (Graphentheorie)
16.11.2023

[EZ:Web18] https://en.wikipedia.org/wiki/A*_search_algorithm
A* search algorithm
16.11.2023

[EZ:Web19] https://de.wikipedia.org/wiki/Bipartiter_Graph
Bipartiter Graph
20.11.2023

[EZ:Web20] https://de.wikipedia.org/wiki/Gerichteter_Graph
Gerichteter Graph
20.11.2023

[EZ:Web21] https://www.geeksforgeeks.org/implementing-generic-graph-in-java
Implementing Generic Graph in Java
20.11.2023

[EZ:Web22] https://de.wikipedia.org/wiki/Kantengewichteter_Graph
Kantengewichteter Graph
24.11.2023

[EZ:Web23] https://www.mathe-online.at/symbole.html
Mathematische Symbole und Abkürzungen
24.11.2023

[EZ:Web24] https://www.mathsisfun.com/sets/symbols.html
Set Symbols
24.11.2023

[EZ:Web25] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)
Graph (discrete mathematics)
24.11.2023

[EZ:Web26] https://de.wikipedia.org/wiki/Einfacher_Graph
Einfacher Graph
24.11.2023

[EZ:Web27] https://de.wikipedia.org/wiki/Schleife_(Graphentheorie)
Schleife (Graphentheorie)
24.11.2023

[EZ:Web28] https://en.wikipedia.org/wiki/Loop_(graph_theory)
Loop (graph theory)
24.11.2023

[EZ:Web29] https://de.wikipedia.org/wiki/Graph_(Graphentheorie)
Graph (Graphentheorie)
24.11.2023

[EZ:Web30] https://de.wikipedia.org/wiki/Idempotenz
Idempotenz
27.11.2023

[EZ:Web31] https://en.wikipedia.org/wiki/Adjacency_list
Adjacency list
27.11.2023

[EZ:Web32] https://www.scaler.com/topics/data-structures/graph-in-data-structure/
Graph in Data Structure
27.11.2023

[EZ:Web33] https://de.wikipedia.org/wiki/Vollständiger_Graph
Vollständiger Graph
27.11.2023

[EZ:Web34] https://en.wikipedia.org/wiki/Complete_graph
Complete graph
27.11.2023

[EZ:Web35] https://de.wikipedia.org/wiki/Knotengewichteter_Graph
Knotengewichteter Graph
28.11.2023

[EZ:Web36] https://de.wikipedia.org/wiki/Adjazenzmatrix
Adjazenzmatrix
28.11.2023

[EZ:Web37] https://en.wikipedia.org/wiki/Adjacency_matrix
Adjacency matrix
28.11.2023

[EZ:Web38] https://de.wikipedia.org/wiki/Reductio_ad_absurdum
Reductio ad absurdum
28.11.2023

[EZ:Web39] https://de.wikipedia.org/wiki/Kante_(Graphentheorie)
Kante (Graphentheorie)
29.11.2023

[EZ:Web40] https://www.educative.io/answers/what-is-an-adjacency-list
What is an adjacency list?
29.11.2023

[EZ:Web41] https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec11/lec11-8.html
Adjacency matrices
29.11.2023

[EZ:Web42] https://iq.opengenus.org/adjacency-matrix/
Adjacency Matrices Explained: A Representation of Graphs
29.11.2023

[EZ:Web43] https://people.engr.tamu.edu/djimenez/ut/utsa/cs1723/lecture16.html
Weighted Graphs
29.11.2023

[EZ:Web44] https://refactoring.guru/design-patterns/builder
Builder
30.11.2023

[EZ:Web45] https://networkx.org/
NetworkX
1.12.2023

[EZ:Web46] https://jgrapht.org/
JGraphT
1.12.2023

[EZ:Web47] en.wikipedia.org/wiki/Dijkstra's_algorithm
Dijkstra's algorithm
29.01.2024

[EZ:Web48] https://de.wikipedia.org/wiki/Open_Shortest_Path_First
Open Shortest Path First
06.02.2024

[EZ:Web49] https://en.wikipedia.org/wiki/Open_Shortest_Path_First
Open Shortest Path First
06.02.2024