Informatik
Hauptcampus

HOCHSCHULE TRIER

Funktionsprinzipien und Anwendungen von Algorithmen zur Pfadplanung

Bearbeiter 1: Mohammed Salih Mezraoui
Bearbeiter 2: David Gruber
Bearbeiter 3: Marius Müller

Gruppe: 7

Ausarbeitung zur Vorlesung Wissenschaftliches Arbeiten

Trier, 17.07.2022

Kurzfassung

In dieser wissenschaftlichen Arbeit werden Algorithmen zur Pfadplanung dargestellt und analysiert. Es wird anhand der Anwendung in Geoinformationssystemen und bei mobilen Robotern aufgezeigt, wie der Algorithmus von Dijkstra eingesetzt wird und welche Stärken und Schwächen damit einhergehen. Da die Fragestellungen, die mit Algorithmen zur Pfadplanung bearbeitet werden, immer komplexer werden, steigen auch die Anforderungen an Zeit- und Raumkomplexität. Daher werden in dieser Arbeit die wichtigsten Optimierungsstrategien für Algorithmen zur Pfadplanung dargestellt. In einer experimentellen Analyse konnte gezeigt werden, dass die Laufzeit des Algorithmus von Dijkstra durch den Einsatz von Preprocessing und heuristischer Funktionen um mehrere Größenordnungen gesenkt werden kann.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

In dieser wissenschaftlichen Arbeit werden aufeinander aufbauend Algorithmen vorgestellt, die in einem Programm mit bestimmten Eingaben und Umgebungen wie z.B. Graphen verwendet werden, um von einem gegebenen Start ein oder mehrere Ziele zu finden und dabei durch verschiedene Bewertungskriterien den besten Weg zu bestimmen [Esr22]. Mit diesen Algorithmen kann ein Programm durch eine Sequenz von Aktionen sein Ziel erreichen. Diesen Prozess nennt man Suche [RN10, 108,109].

Es werden verschiedene Pfadsuchalgorithmen, wie der Dijkstra-Algorithmus und der A* Algorithmus, welcher im Verkehrleitsystem von Google Maps verwendet wird, präsentiert [MKL19].

Das Thema *Funktionsprinzipien und Anwendungen von Algorithmen zur Pfadplanung* hat, damals wie heute, einen wichtigen Stellenwert. Ob bei der Netzwerkroutenplanung oder bei KI-Spielern in Computerspielen: Pfadsuchalgorithmen sind so relevant wie nie [FGK+21]. Weitere Beispiele für die Anwendung von Pfadsuchalgorithmen sind Planung von öffentlichen Verkehrsmitteln und Robotik. Es werden beispielsweise Pakete in Logistikzentren durch Roboter organisiert und Routen über verschiedene Logistikzentren durch Pfadsuchalgorithmen bestimmt. Viele Bereiche im Alltag verwenden im Hintergrund Pfadsuchalgorithmen um den (kosten)günstigsten Weg zu finden. Dabei ist es wichtig, dass diese effizient, akkurat und schnell sind, damit die Hauptsysteme genügend Ressourcen übrig haben, um wie gewünscht zu funktionieren [FGK+21].

In dieser wissenschaftlichen Arbeit wird die Funktionsweise von wichtigen Pfadsuchalgorithmen erklärt, angefangen mit den uninformierten Suchalgorithmen wie der Breiten- und Tiefensuche, welche einen ersten Einblick in die Thematik geben und die Rahmenbedingungen und Problemumgebungen veranschaulichen sollen. Vom Bellman-Ford-Algorithmus, der nahezu intuitiv den kürzesten Weg liefert und bei dem auch negative Kantenlängen möglich sind [MS20], bis hin zu durch Heuristiken optimierte und informierte Pfadsuchalgorithmen wie A*, und wie sie in heutiger Zeit eingesetzt werden, wird hier berichtet. Die genannten Algorithmen werden teilweise durch Pseudocode- und Anwendungsbeispiele veranschaulicht.

2

Algorithmen zur Pfadplanung

In diesem Kapitel werden die Algorithmen *Dijkstra* und *Bellman-Ford* zur Pfadplanung beschrieben.

2.1 Einleitung

Die Pfadplanung ist ein nichtdeterministisches Problem mit polynomialer Laufzeit ("NP"), bei dem es darum geht, einen Pfad zu finden, der in einem System den Ausgangspunkt mit dem Ziel verbindet. Der zu wählende Weg (die beste Route) wird durch Beschränkungen und Bedingungen bestimmt [KSDS21].

In der Informatik, insbesondere in der Graphentheorie, ist das Problem des kürzesten Weges bekannt. Der kürzestmögliche Weg von einer Quelle zu einem Ziel hat die geringsten Längenanforderung.

Die Dijkstra- und Bellman-Algorithmen für den kürzesten Weg sind in einer Vielzahl von Bereichen und Anwendungen weit verbreitet. Beispiele für diese Anwendungen sind Routing-Protokolle für Netzwerke, Routenplanung, Verkehrssteuerung, Pfadfindung in sozialen Netzwerken, Computerspiele und Navigationssysteme [PM18].

2.2 Uninformierte Suche

Als Einstieg in die Thematik der Pfadplanung wird die Uninformierte Suche beschrieben. Uninformierte Algorithmen werden auch „blind“ genannt, weil bei ihrer Suche auf keine zusätzlichen Informationen (wie z.B. Gewichtungen) zurückgegriffen wird [RN10, 80-82].

Nach [RN10, 80-82] gibt es 4 Hauptkriterien nach denen Pfachsuchalgorithmen bewertet werden können:

- *Vollständigkeit*: Liefert der Algorithmus immer eine Lösung, wenn es eine Lösung gibt?
- *Optimalität*: Liefert der Algorithmus einen optimalen Weg?
- *Zeitkomplexität*: Wie lange dauert es, bis der Algorithmus einen Weg gefunden hat?
- *Raumkomplexität*: Wie viel Speicher wird benötigt, um die Lösung zu finden?

2.2.1 Breitensuche

Die Breitensuche gehört zu den uninformierten Suchalgorithmen. Der erste Schritt der Breitensuche ist es, wie in Abb 2.1 zu sehen ist, alle verbundenen Knoten ersten Grades des Wurzelknotens zu besuchen [RN10, 80-82]. Erst dann wird die nächste Ebene besucht. Dies wird Ebene für Ebene im Baum wiederholt, bis alle Knoten besucht wurden. Die Breitensuche ist vollständig und bei z.B gleichverteilten Pfadkosten auch optimal [RN10, 80-82].

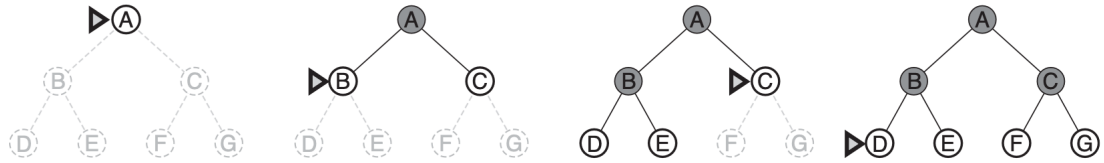

Abb. 2.1: Breitensuche in einem Binärbaum. Unbesuchte Knoten werden hellgrau dargestellt, besuchte Knoten dunkelgrau [RN10, 82].

2.2.2 Tiefensuche

Die Tiefensuche gehört ebenfalls zu den uninformierten Suchalgorithmen. Das Vorgehen bei der Tiefensuche ist es, wie in Abb. 2.2 zu sehen ist, zuerst alle Unterknoten eines direkten Nachbarn (z.B Knoten B) des Wurzelknotens A zu besuchen, bis dieser keine unbesuchten Unterknoten mehr hat [RN10, 85,86]. Erst dann wird der nächste Nachbar, in Abb. 2.2 Knoten C, in der ersten Ebene besucht, bis keine unbesuchten Knoten mehr vorhanden sind oder der Zielknoten gefunden wurde [RN10, 85,86]. Es gibt 2 Versionen der Tiefensuche: Die Version, die in einem Graphen den kürzesten Weg zu einem Ziel sucht, und die Version, die diesen in einer Baumstruktur sucht [RN10, 85,86]. Nur die Version, welche in einem Graphen sucht, ist in endlicher Raumkomplexität vollständig, die Version mit Baumstruktur nicht [RN10, 85,86].

Die iterative Vertiefungssuche die klassische Tiefensuche wiederholt und in Verbindung mit einer steigenden Tiefenbegrenzung aus, bis ein Ziel gefunden wird [RN10, 108,109].

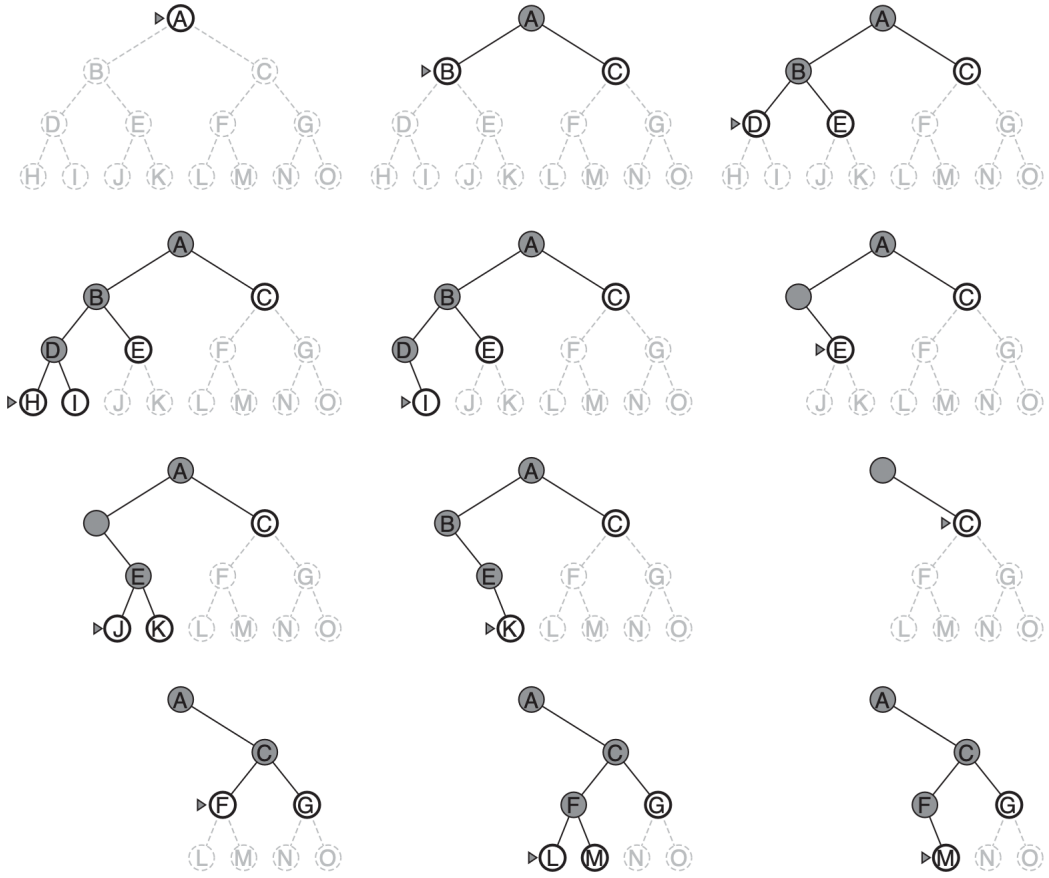

Abb. 2.2: Tiefensuche in einem Binärbaum. A ist der Startknoten, M der Zielknoten; unentdeckte Knoten sind hellgrau. Bereits entdeckte Knoten, ohne weitere Unterknoten, werden entfernt [RN10, 86].

2.3 Dijkstra Algorithmus

Der Dijkstra-Algorithmus (benannt nach seinem Entdecker E.W. Dijkstra) ist ein bekannter Algorithmus auf dem Gebiet der optimalen Pfadwahl und wird verwendet, um den kürzesten Pfad von einem Startpunkt in einem Graphen zu einem Zielpunkt zu finden [Jav13].

2.3.1 Funktionsprinzip

Das Grundkonzept des Algorithmus ist wie folgt:

In einem ersten Schritt wird der Startknoten festgelegt und der Abstand zwischen ihm und den anderen Knoten des Graphen berechnet. Gibt es keine Kante, die diesen Knoten mit dem Startpunkt verbindet, ist der Abstand unendlich; gibt es eine Kante, die diesen Knoten mit dem Startpunkt verbindet, ist der Abstand n; und das niedrigste Gewicht (wenn es mehrere Kanten gibt) ist n.

Der zweite Schritt besteht darin, den Punkt mit dem kürzesten Abstand zum Startpunkt zu ermitteln und zu speichern. Die zuvor ermittelte Entfernung wird mit derjenigen verglichen, die über den soeben beiseite gelegten Scheitelpunkt für alle verbleibenden Scheitelpunkte ermittelt wurde, und nur der kleinste der beiden Werte wird beibehalten. Dieser Vorgang wird so lange wiederholt, bis es keine Scheitelpunkte mehr gibt oder bis der Ankunftsscheitelpunkt gewählt ist [ZG19].

Der Dijkstra-Algorithmus arbeitet mit der Zuweisung einiger vorläufiger Entfernungswerte und versucht, diese schrittweise zu verbessern. Der Pseudocode des Algorithmus ist in der Abbildung 2.3 dargestellt [HG12].

Im Folgenden wird die Umsetzung des unten stehenden Pseudocodes erläutert, angelehnt an [AIS+20].

Beim Dijkstra-Algorithmus ist die Route unbekannt. Die Knoten werden als temporär (t) oder permanent (p) eingestuft.

- Schritt1: die Entfernung des Quellknotens wird der Wert Null zugewiesen [$distance(source) = 0$], und die Entfernung der anderen Knoten wird auf unendlich gesetzt [$distance(x) = Infinity$].
- Schritt 2: Suche des Knotens x mit der kürzesten Entfernung $d(x)$. Wenn $d(x)$ unendlich ist oder es keine temporären Knoten gibt, wurde der Knoten x als permanent markiert, was bedeutet, dass sich $d(x)$ und sein übergeordneter Wert nicht ändern werden.
- Schritt 3: Für jeden temporären Knoten mit dem Label y, der an x angrenzt, wird der folgende Vergleich durchgeführt:

wenn

$$d(x) + w(x, y) < d(y) \tag{2.1}$$

dann

$$D(y) = d(x) + w(x, y) \tag{2.2}$$

Wenn die Entfernung des gekennzeichneten Knotens $d(y)$ größer ist als die Entfernung des gekennzeichneten Knotens $d(x)$ plus Verbindungsgewicht $w(x, y)$, wird die Entfernung des gekennzeichneten Knotens y gemäß den Gleichungen 2.1 und 2.2 aktualisiert in $D(y)$.

```
1  function DIJKSTRA (Graph, source)
2      create vertex set D
3      for each vertex v in Graph:
4          distance[v] ← INFINITY
5          previous[v] ← UNDEFINED
```

Listing 2.1: Dijkstra Algorithmus Pseudocode

Algorithm 1 Dijkstra Algorithm

```
1: function DIJKSTRA (Graph, source)
2:     create vertex set D
3:     for each vertex v in Graph:
4:         distance[v] ← INFINITY
5:         previous[v] ← UNDEFINED
6:         add v to D
7:     distance[source] ← 0
8:
9:     while D is not empty do:
10:        u ← in D with min distance[u]
11:        remove u from D
12:        for each neighbour v of u:
13:        alt ← distance[u] + length (u, v)
14:        if alt < distance[v]
15:            distance[v] ← alt
16:            previous[v] ← u
17:    end while
18:    return distance [], previous []
19: end function
```

Abb. 2.3: Dijkstra Algorithmus Pseudo-Code, angelehnt an [AIS+20].

2.3.2 Vor- und Nachteile

Der Dijkstra-Algorithmus hat zwei wesentliche Vorteile: Er kann alle optimalen Pfade finden und die Trefferquote dafür liegt bei 100%. Der zweite Vorteil ist, dass er die verbleibenden unerwünschten Knoten nicht besucht, wenn der beabsichtigte Zielknoten erreicht ist [ZG19, AIS+20].

Der Hauptnachteil des Algorithmus besteht darin, dass er eine blinde Suche durchführt, was eine erhebliche Menge an Zeit und Ressourcen vergeudet. Ein weiterer Nachteil ist, dass er nicht mit negativen Kanten umgehen kann, was zu azyklischen Graphen führt, und daher nicht immer den kürzesten Weg findet [MS20].

2.4 Bellman-Ford Algorithmus

In 1958 veröffentlichte Richard Bellman den Bellman-Ford-Algorithmus, einen Graphen-Suchalgorithmus zur Ermittlung des kürzesten Pfades [AIS+20, SSNH18].

2.4.1 Eigenschaften

Um kürzeste Wege auf gerichteten Graphen zu finden, nutzt der Bellman-Ford-Algorithmus die Relaxation. Relaxation bedeutet, dass geprüft wird, ob es möglich ist, den Weg zu dem Knoten, auf den die Kante zeigt, zu verkürzen, und wenn dies der Fall ist, wird der Weg zum Knoten durch den entdeckten Weg ersetzt. Der Bellman-Ford Algorithmus kann auch mit negative Kantengewichten umgehen. Wenn es Zyklen mit negativem Gewicht gibt, wird der Algorithmus sie erkennen (so dass es keine Lösung gibt) [VB14].

Die Vorteile dieses Algorithmus sind unter anderem, dass es sich um einen dynamischen Algorithmus handelt, dass er negative gerichtete Kanten (und auch positive) berechnen kann, dass er die Kosten für den Aufbau des Netzes reduzieren kann, indem er den kürzesten Weg von einem Knoten zum anderen findet, und dass er die Anzahl der aufgebauten Router-Pfad reduzieren kann [AIS+20].

2.4.2 Funktionsprinzip

Angelehnt an [AIS+20] wird der Bellman-Ford-Algorithmus wie folgt ausgeführt:

- Schritt 1: Zuweisung des Abstandswertes des Startpunktes s zu null ($distance[s] = 0$) und des Abstandswertes der anderen Punkte zu $INFINITY$.
- Schritt 2: Wenn n die Anzahl der Knoten ist, wird jede Kante $(n - 1)$ Mal relaxiert.
- Schritt 3: Mit der N-ten Schleife wird geprüft, ob der Graph negative Zyklen aufweist.

Der Bellman-Ford-Algorithmus wird wie in Listing 2.2 unten dargestellt ausgeführt.

```
1  function bellmanFord (G, s)
2      for each vertex V in G:
3          distance[v] ← INFINITY
4          previous[v] ← NULL
```

Listing 2.2: Bellman-Ford-Algorithmus Pseudocode

Algorithm 2 Bellman-Ford Algorithm

```
1: function bellmanFord (G, S)
2:     for each vertex V in G
3:         distance[v] ← INFINITY
4:         previous[v] ← NULL
5:     distance[s] ← 0
6:     for each vertex V in G
7:         for each edge (u, v) in G
8:             alt ← distance[u] + length (u, v)
9:             if alt < distance[v]
10:                distance[v] ← alt
11:                previous[v] ← u
12:
13:  for each edge (u, v) in G
14:      if distance[u] + length (u, v) < distance(v)
15:          Error: Negative Cycle Exists
16: return distance [], previous []
```

Abb. 2.4: Bellman-Ford Pseudo-Code, angelehnt an[AIS+20].

3

Optimierungsstrategien

Die Anforderungen an die Laufzeit von Algorithmen zur Pfadplanung steigt auf Grund der immer komplexer werdenden Systemen stetig weiter an. In diesem Kapitel wird beschrieben, wie die bisher vorgestellten Algorithmen zur Pfadplanung für bestimmte Anwendungsgebiete optimiert werden können.

3.1 Bidirektionale Suche

Bei der bidirektionalen Suche wird ein Suchalgorithmus simultan aus zwei Richtungen laufen gelassen – vom Startknoten zum Zielknoten und umgekehrt. Der Suchalgorithmus wird bei diesem Vorgehen so modifiziert, dass die Abbruchbedingung dann eintritt, wenn beide Suchen denselben Knoten expandieren. Somit betrachtet jede der beiden Suchen nur die Hälfte des Graphen, was in einer Reduktion der Zeitkomplexität resultiert [RN10, 90,91]. Es ist jedoch zu beachten, dass die Raumkomplexität stark ansteigt, da in beiden Suchen eigene Priority-Queues verwaltet werden müssen. Außerdem ist es für viele Problemstellungen keinesfalls trivial, eine Suche rückwärts durchzuführen, da eine Methode zur Berechnung des Vorgängers eines Knotens gegeben sein muss [RN10, 90,91].

3.2 Informed Search

Ein Ansatz, um effizienter Lösungen für das Shortest Path Problem zu finden, ist die Informed Search Strategie, bei der problemspezifisches Wissen, das über die Definition des Problems hinausgeht, bei der Lösungsfindung berücksichtigt wird. Der nächste zu expandierende Knoten auf dem Pfad zum Zielknoten wird auf Basis einer Bewertungsfunktion $f(n)$ ausgewählt. Eine Komponente dieser Bewertungsfunktion ist eine heuristische Funktion $h(n)$, die die zu erwartenden Kosten des optimalen Pfades vom Knoten n zum Zielknoten berechnet [RN10, 92-102]. Im Falle des Straßennetzes könnte hierzu die Länge der Luftlinie zwischen dem Knoten n und dem Zielknoten verwendet werden [HNR68].

Die einfachste Umsetzung dieser Strategie ist, nur die heuristische Funktion bei der Bewertung von Knoten heranzuziehen, sodass $f(n) = h(n)$ gilt. Dieses Vorgehen wird auch Greedy Best-First Suche genannt, da in jedem Schritt versucht

wird, so nahe wie möglich an den Zielknoten zu gelangen [RN10, 92-102]. Auf diese Weise werden die Suchkosten, also die Anzahl der expandierten Knoten zwar minimiert, es kann jedoch nicht garantiert werden, dass die gefundene Lösung optimal ist [RN10, 92-102]. Eine elaboriertere Umsetzung der Informed Search Strategie ist der A* Algorithmus zur Berechnung des kürzesten Pfades zwischen zwei Knoten [RN10, 92-102]. A* basiert auf Dijkstra's Algorithmus und erweitert diesen um eine heuristische Funktion, um die Laufzeit zu reduzieren [PAG+20]. Die Bewertungsfunktion $f(n)$ für den A*-Algorithmus setzt sich zusammen aus den Kosten des optimalen Pfades vom Startknoten bis zum Knoten n, $g(n)$ und einer heuristischen Funktion $h(n)$, sodass gilt:

$$f(n) = g(n) + h(n) \tag{3.1}$$

Da verschiedene Heuristiken zur Konstruktion von $h(n)$ gewählt werden können, stellt A* streng genommen eine Familie von Algorithmen dar, wobei die Wahl einer Funktion $h(n)$ einen spezifischen Algorithmus der Familie selektiert [HNR68].

Hart, Nilsson und Raphael, die in [HNR68] die A*-Suche 1968 zum ersten Mal beschrieben haben, konnten nachweisen, dass A* vollständig und optimal ist, wenn die gewählte Heuristik zulässig und konsistent ist. Das heißt, dass unter den angegebenen Voraussetzungen für die Heuristik immer ein Pfad vom Start- zum Zielknoten gefunden wird (sofern dieser existiert) und dass dieser Pfad in jedem Fall optimal ist. Des Weiteren konnte gezeigt werden, dass A* optimal effizient ist – es kann also keinen anderen optimalen Algorithmus geben, der garantiert weniger Knoten expandiert als A* [RN10, 92-102].

3.3 Preprocessing

Eine weitere Optimierungsstrategie ist das Preprocessing, also die Vorverarbeitung des Graphen. Dabei wird gefordert, dass die Raumkomplexität der vorverarbeiteten Daten linear in der Größe des zu bearbeitenden Graphen ist, da in der Realität oft mit sehr großen Graphen gearbeitet wird. [GH05].

3.3.1 ALT-Algorithmen

Eine Familie von Algorithmen, die auf Preprocessing basieren, sind die von Goldberg und Harrleson in [GH05] vorgestellten ALT-Algorithmen. ALT ist ein Akronym für A* search, *Landmarks* und *Triangle Inequality*[1].

In der Preprocessing-Phase des Algorithmus, wird eine kleine (konstante) Anzahl von Landmarken im Graphen ausgewählt, von denen aus dem kürzesten Pfad zu allen anderen Knoten bestimmt wird. Die so bestimmte Distanz wird in Kombination mit der Dreiecksungleichung als Heuristik für die Bewertungsfunktion der A* Suche verwendet.

[1] Dreiecksungleichung

Die Dreiecksungleich besagt in diesem Fall, dass die Distanz zwischen einem Knoten n und einem Zielknoten s in jedem Fall größer oder gleich der Differenz der Distanz zwischen n und einer Landmarke l und der Distanz zwischen s und der Landmarke l ist.

$$dist(n, s) \geq dist(l, n) - dist(l, s) \tag{3.2}$$

Diese Differenz stellt eine untere Schranke für die Distanz zwischen n und s dar und kann somit als Heuristik verwendet werden [GH05].

3.3.2 Reach-Based Pruning

Reach-Based Pruning [2] ist eine von Gutman in [Gut04] vorgestellte Preprocessing-Strategie zur Vereinfachung von Graphen. Der *Reach* r ist hierbei eine Metrik für einen Graphen G, die als Modifikation für den Dijkstra Algorithmus verwendet werden kann.

Sei P ein Pfad in G von einem Startknoten s zu einem Zielknoten t und v ein Knoten auf diesem Pfad P. Sei zudem $dist(v, w, P)$ die Distanz zwischen den Knoten v und w auf dem Pfad P. Dann ist

$$r(v, P) = \min\left(dist(s, v, P), dist(v, t, P)\right) \tag{3.3}$$

der *Reach* von v auf P. Zudem ist der *Reach* von v in G, $r(v, G)$ definiert als das Maximum aller $r(v, Q)$ für alle kürzesten Pfade Q in G.

Gutman konnte beweisen, dass ein Knoten v nur dann vom Dijkstra Algorithmus betrachtet werden muss, wenn

$$r(v, G) \geq \underline{dist(s, v)} \quad \vee \quad r(v, G) \geq \underline{dist(v, t)} \tag{3.4}$$

gilt, wobei $\underline{dist(v, w)}$ eine untere Schranke für die Distanz zwischen zwei Knoten v und w darstellt.

Die einfachste Möglichkeit die *Reaches* aller Knoten zu bestimmen, ist alle kürzesten Pfade eines Graphen zu bestimmen und die Bedingung 3.4 anzuwenden. Effizientere Vorgehen wurden in [Gol07] und [Gut04] beschrieben.

[2] englischer Ausdruck für das Beschneiden von Bäumen. In der Informatik wird *Pruning* oft als Ausdruck für das Vereinfachen von Graphen verwendet [Wik].

3.4 Experimentelle Analyse

Um die Auswirkung der hier vorgestellten Optimierungsstrategien zu veranschaulichen, wurde von Goldberg in [Gol07] die Laufzeit der folgenden Algorithmen verglichen:

- ***B***: Bidirektionaler Dijkstra Algorithmus
- ***ALT***: Algorithmus aus der ALT-Familie
- ***RE***: Implementierung der Reach-Based Pruning Strategie
- ***REAL***: Algorithmus mit zwei Preprocessing-Stufen (ALT und RE)

Als Input wurde das Straßennetz der San Francisco Bay Area mit 330024 Knoten und 793681 Kanten verwendet und jeder dieser Algorithmen wurde auf 10.000 zufällig gewählten Paaren von Knoten angewendet. Dabei wurden die Laufzeit der Preprocssing-Phase und der Query-Phase der Algorithmen gemessen[3] und in Tabelle 3.1 dargestellt.

Algorithmus	**Laufzeit Preprocessing**	**Laufzeit Query**
B	-	$30,49\,\text{ms}$
ALT	$5,7\,\text{s}$	$2,91\,\text{ms}$
RE	$45,4\,\text{s}$	$0,55\,\text{ms}$
REAL	$51,1\,\text{s}$	$0,28\,\text{ms}$

Tabelle 3.1: Messung der durchschnittlichen Laufzeit der Preprocessing- und Query-Phase der Algorithmen *B*, *ALT*, *RE* und *REAL* auf dem Straßennetz der San Francisco Bay Area.

Man kann erkennen, dass die Laufzeit der Algorithmen mit Preprocessing-Phase (*ALT*, *RE*, *REAL*) signifikant besser ist, als die der Algorithmen ohne Preprocessing-Phase (*B*). Es ist jedoch zu beachten, dass die Laufzeit der Preprocessing-Phase um einige Größenordnungen höher ist, als die Laufzeit des eigentlichen Query-Algorithmus.

[3] Die Laufzeitmessungen wurden auf einem Toshiba Tecra 5 Laptop mit 2GB RAM und Dual-Core 2 GHz Processor durchgeführt

4

Anwendungen

In diesem Kapitel wird beschrieben, wie Pfadplanung Algorithmen in realen Anwendungen eingesetzt werden können.

4.1 GIS(Geoinformationssystem)

Ein geografisches Informationssystem (GIS) ist ein System zum Erstellen, Verwalten, Analysieren und Kartieren verschiedener Arten von Daten. GIS verknüpft Daten mit einer Karte und integriert Standortdaten mit verschiedenen Arten von beschreibenden Daten. Dies ist die Grundlage für die Kartierung und Analyse in der Wissenschaft und in fast allen Branchen. Bei diesen Daten handelt es sich um Informationen über Objekte auf der Erde wie Städte, Eisenbahnstrecken, Flüsse usw [VB14].

Algorithmen für den kürzesten Weg werden in Kartenplattformen wie Google Maps verwendet, um den kürzesten Weg zwischen zwei Punkten zu ermitteln. Obwohl der Dijkstra-Standardalgorithmus in diesem Fall anwendbar zu sein scheint, dauert das Routing des Pfades vom Startpunkt zum Endpunkt in einer großen Datenmenge sehr lange [HAMTMA20].

Um bei der Berechnung des kürzesten Weges Zeit zu sparen und eine bessere Lösung zu erhalten. Beeinflusst von [HAMTMA20], der bessere Ansatz besteht darin, vor dem Start des Dijkstra-Algorithmus einen temporären Datensatz zu erstellen und ihn so zu behandeln, als wäre er der Graph, mit dem der Algorithmus arbeiten muss. Nachdem die Start- und Endknoten ermittelt wurden, wird dieser Datensatz erstellt. Der Datensatz kann anhand der Koordinaten des Start- und des Zielknotens aus den Hauptdaten ausgeschlossen werden, indem nur die Knoten ausgewählt werden, die sich innerhalb des von den beiden Knoten gebildeten Quadrats befinden. Es gibt andere Algorithmen, die in GIS verwendet werden können, um bessere Ergebnisse zu erzielen, wie z. B. A*.

4.2 Mobile Roboter

Mit der Weiterentwicklung der Internet-of-Things-Technologien in den letzten Jahren hat die Entwicklung intelligenter Städte in der Industrie große Aufmerksamkeit erlangt. Die Steuerung mobiler Roboter ist ein wichtiges Thema in intelligenten Städten und die Pfadplanung ist eine wichtige Komponente unter den mobilen Roboter-Technologien, die dem Roboter hilft, den Weg von einem Startpunkt zu einem Zielpunkt zu finden und dabei sicher und zuverlässig alle Hindernisse in einer statischen und dynamischen Umgebung während der Reise zu vermeiden [MW21, mZlL17].

4.2.1 Umgebungsmodell

Das Rolling-Window-Prinzip wird zur Hindernisvermeidung in einer unbekannten Umgebung verwendet, während die Algorithmen Dijkstra und A* in erster Linie für die Pfadplanung in einer bekannten Umgebung eingesetzt werden.
Da sowohl der Dijkstra- als auch der A*-Algorithmus Gitter-basierte Suchmethoden sind, sollte zunächst das Gittermodell der umgebenden Karte erstellt werden. Bei der Gittermethode wird die Karte in benachbarte Gitter gleicher Größe unterteilt. Die Größe des mobilen Roboters bestimmt die Größe des Gitters und beeinflusst die Suchgenauigkeit und Effizienz des Algorithmus [mZlL17].
Das Gittermodell einer Umgebungskarte ist in der Abbildung 4.1 unten dargestellt.

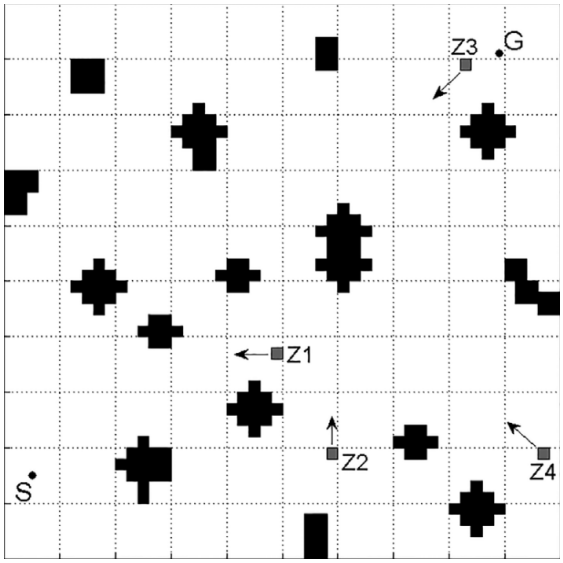

Abb. 4.1: Umgebungsmodell angelehnt an [mZlL17].

4.2.2 Dijkstras Algorithmus und A*-Algorithmus

Die Idee ist, A*- und Dijkstra-Algorithmen miteinander zu kombinieren, um die Effizienz der Planung einer idealen kollisionsfreien Route für einen mobilen Roboter zu erhöhen. Bei der Neuplanung wird die Dijkstra-Methode zur Vorverar-

beitung der statischen Umgebungskartendaten angewendet, die optimale Routen vom Ziel G zu allen freien Zuständen plant und speichert. Wenn der Roboter auf seinem Weg zum Ziel auf ein bewegliches Hindernis trifft, wählt das Rolling-Window-Prinzip einen lokal optimalen Zustand als nächsten Zielzustand. Dann wird mit dem A*-Algorithmus eine lokal optimale Route von der Roboterposition zum lokalen Ziel neu geplant. Die neue Route kann den Roboter um Hindernisse herumführen [mZlL17].

4.3 Autonome Navigation

Pfadsuchalgorithmen werden ebenfalls im Bereich des Autonomen Fahrens, oder auch bei unbemannten Flugfahrzeugen verwendet, um sichere, effiziente, kollisionsfreie und kostengünstige Wege von Start zum Ziel zu führen, was die Wahl des richtigen Pfadsuchalgorithmus zu einer wichtigen Aufgabe macht. Es hängt unter Anderem die Geometrie des Fahrzeugs von dieser Wahl ab [KSDS21]. Mit der zunehmenden Verbreitung von autonomen Fahrzeugen, die immer mehr Wegfindung und -planung erfordert, sind Pfadsuchalgorithmen zu einem neuen Schwerpunkt der autonomen Steuerung geworden [KSDS21]. Da mobile Roboter in vielen Anwendungen eingesetzt werden, haben Forscher Methoden entwickelt, um die Anforderungen an mobile Roboter effektiv erfüllen zu können und einige Herausforderungen für die Umsetzung einer vollständig oder teilweise autonomen Navigation in unübersichtlichen Umgebungen zu bewältigen [KSDS21]. So wird die Wahl des richtigen Pfadplanungsalgorithmus von der kinematischen Bewegungsgestaltung des Roboters/Fahrzeugs, den zur Verfügung stehenden Rechenressourcen, sowie der sensorischen Ausstattung des Fahrzeugs bestimmt [KSDS21].

Die Leistung und Komplexität des verwendeten Algorithmus hängt vom Anwendungsfall ab [KSDS21].

Es gibt nicht *den perfekten Pfadsuchalgorithmus für Autonomes Fahren*, es finden aber viele verschiedene Algorithmen eine Anwendung in der autonomen Navigation.

5

Zusammenfassung und Ausblick

In den letzten Jahren hat die Pfadplanung immer mehr an Relevanz gewonnen, zum Beispiel wird beim autonomen Autofahren immer mehr Forschung im Bereich der Pfadplanungsalgorithmen betrieben [KSDS21].

- Bevor ein Programm mit der Suche nach der besten Lösung (dem besten Weg) beginnen kann, muss erst ein Ziel deklariert und das Problem (Umgebung) genau definiert werden [RN10, 108,109].
- Suchalgorithmen betrachten Zustände und Aktionen atomar, das heißt sie berücksichtigen keine interne Struktur, die sie besitzen könnten [RN10, 108,109].
- Sie werden nach folgenden Kriterien bewertet: Optimalität, Vollständigkeit, sowie Raum- und Zeitkomplexität [RN10, 80].
- Uninformierte Pfadsuchalgorithmen verfügen nur über eine grundlegende Problemdefinition und keine weiteren Metriken/Heuristiken. In dieser wissenschaftlichen Arbeit wurden folgende uninformierte Algorithmen vorgestellt:
  - Die Breitensuche (Kapitel 2.2.1) expandiert zuerst die flachsten Knoten. Sie ist vollständig, optimal für einheitliche Pfadkosten, hat jedoch eine exponentielle Raumkomplexität [RN10, 81].
  - Die Tiefensuche (Kapitel 2.2.2) expandiert zuerst den tiefsten nicht expandierten Knoten. Sie ist weder vollständig noch optimal, hat aber eine lineare Raumkomplexität [RN10, 85,86].
  - Die iterative Vertiefungssuche (Kapitel 2.2.2) ist eine Wiederholung der Tiefensuche mit zunehmender Tiefenbegrenzung, bis ein Ziel gefunden wird. Sie ist vollständig, optimal für die Kosten pro Schritt, hat eine vergleichbare Zeitkomplexität wie die Breitensuche und eine lineare Raumkomplexität [RN10, 85,86].
  - Der Greedy Dijkstra-Algorithmus (Kapitel 2.3) der zwar bei der blinden Suche Zeit vergeudet, aber dafür optimal ist und eine Trefferquote von 100% hat [KSDS21].
  - Die Optimierung durch eine bidirektionale Suche kann die Zeitkomplexität reduzieren, allerdings ist sie nicht immer für das Problem geeignet und kann viel Speicherplatz beanspruchen [RN10, 108,109].

- Informierte Suchmethoden basieren auf heuristischen Funktionen, die die Kosten einer Lösung schätzen können [RN10, 108,109].
  - Der Greedy Best-First-Search-Algorithmus (Kapitel 3) expandiert Knoten nach einem minimalen heuristischen Funktionswert. Er ist nicht optimal, aber dafür effizient [RN10, 108,109].
  - A*-Suche (Kapitel 3.2) expandiert Knoten mit minimalem Heuristischen Funktions- und Pfadkostenwerten. A* ist vollständig und optimal, vorausgesetzt die heuristische Funktion ist zulässig.
  - ALT-Algorithmen (Kapitel 3.3.1), die aufgebaut auf A* durch Preprocessing noch optimiertere Ergebnisse erzeugt.
  - Reach-based Pruning (Kapitel 3.3.2), welches den Dijkstra-Algorithmus um eine Metrik erweitert und dadurch optimiert.
- Wie ein, durch eine heuristische Funktion optimierter, informierter Pfadsuchalgorithmus leistungsbezogen abschneidet, hängt von der Qualität der heuristischen Funktion ab [RN10, 108,109].

Literaturverzeichnis

AIS+20. Abusalim, Samah, Rosziati Ibrahim, Mohd Saringat, Sapiee Jamel und Jahari Wahab: *Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization.* IOP Conference Series: Materials Science and Engineering, 917:012077, 09 2020.

Esr22. *Esri GIS Dictionary: Pathfinding.* online, Juli 2022. https://support.esri.com/en/other-resources/gis-dictionary/term/7f861382-d88c-4828-8272-c3da4bdc8fa6.

FGK+21. Foeada, Daniel, Alifio Ghifaria, Marchel Budi Kusumaa, Novita Hanafiahb und Eric Gunawanb: *A Systematic Literature Review of A* Pathfinding.* Elsevier B.V, 2021.

GH05. Goldberg, Andrew V. und Chris Harrelson: *Computing the shortest path: A search meets graph theory.* SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, Seiten 156–165, Januar 2005.

Gol07. Goldberg, Andrew V.: *Point-to-Point Shortest Path Algorithms with Preprocessing.* In: *Lecture Notes in Computer Science*, Seiten 88–102. Springer Berlin Heidelberg, 2007.

Gut04. Gutman, Ron: *Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks.* In: *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, New Orleans, LA, USA, Januar 2004.

HAMTMA20. Hamid Ali, Abed Alasadi, Azizand Mohammed Talib, Dhiya Mohammed und Abdulmaje Ahmed: *A Network Analysis for Finding the Shortest Path in Hospital Information System with GIS and GPS.* Journal of Network Computing and Applications (2020) 5: 10-22 Clausius Scientific Press, Canada, 2020.

HG12. Huang, Haosheng und Georg Gartner: *Collective intelligence-based route recommendation for assisting pedestrian wayfinding in the era of Web 2.0.* Journal of Location Based Services, 6:1–21, 03 2012.

HNR68. Hart, Peter, Nils Nilsson und Bertram Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107, 1968.

Jav13. Javaid, Adeel: *Understanding Dijkstra Algorithm.* SSRN Electronic Journal, 01 2013.

KSDS21. Karur, Karthik, Nitin Sharma, Chinmay Dharmatti und Joshua E. Siegel: *A Survey of Path Planning Algorithms for Mobile Robots.* Vehicles, 3(3):448–468, aug 2021.

MKL19. Mehta, Heeket, Pratik Kanani und Priya Lande: *Google Maps.* International Journal of Computer Applications, 178(8):41–46, may 2019.

MS20. Mukhlif, Fadhil und Abdu Saif: *Comparative Study On Bellman-Ford And Dijkstra Algorithms.* Int. Conf. Comm. Electric Comp. Net., 2020.

MW21. Myung, Hyun und Yang Wang: *Robotic Sensing and Systems for Smart Cities.* Sensors, 21(9), 2021.

mZlL17. Zhang, Hong mei und Ming long Li: *Rapid path planning algorithm for mobile robot in dynamic environment.* Advances in Mechanical Engineering, 9(12):1687814017747400, 2017.

PAG+20. Peralta, Federico, Mario Arzamendia, Derlis Gregor, Daniel G. Reina und Sergio Toral: *A Comparison of Local Path Planning Techniques of Autonomous Surface Vehicles for Monitoring Applications: The Ypacarai Lake Case-study.* Sensors, 20(5):1488, mar 2020.

PM18. Panda, Madhumita und Abinash Mishra: *A Survey of Shortest-Path Algorithms.* International Journal of Applied Engineering Research, 13(9):6817, 2018.

RN10. Russell, Stuart J. und Peter Norvig: *Artificial Intelligence: A Modern Approach.* Prentice Hall, Third Auflage, 2010.

SSNH18. Sulaiman, Oris, Amir Siregar, Khairuddin Nasution und Tasliyah Haramaini: *Bellman Ford algorithm - in Routing Information Protocol (RIP).* Journal of Physics: Conference Series, 1007:012009, 04 2018.

VB14. Vaibhavi, Patel und Prof.Chitra Baggar: *A survey paper of Bellman-ford algorithm and Dijkstra algorithm for finding shortest path in GIS application.* International Journal of P2P Network Trends and Technology (IJPTT), 2014.

Wik. *Wikipedia - Online Lexikon.* `de.wikipedia.org/wiki/Pruning`.

ZG19. Zhou, Minhang und Nina Gao: *Research on Optimal Path based on Dijkstra Algorithms.* Proceedings of the 3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019), 01 2019.

Arbeitsverteilung

Teilnehmer 1: Mohammed Salih Mezraoui
Inhalte:

- Kapitel 2.1: Was ist Pfadplanung?
- Kapitel 2.3: Dijkstra Algorithmus
- Kapitel 2.4: Bellman-Ford-Algorithmus
- Kapitel 4.1: GIS(Geoinformationssystem)
- Kapitel 4.2: Mobile Roboter

Teilnehmer 2: David Gruber
Inhalte:

- Kapitel 1: Einleitung und Problemstellung
- Kapitel 2.2: Uninformierte Suche
- Kapitel 4.3: Autonome Navigation
- Kapitel 5: Zusammenfassung und Ausblick

Teilnehmer 3: Marius Müller
Inhalte:

- Kapitel 3: Optimierungsstrategien