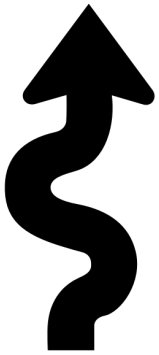

DHBW Karlsruhe

Wissenschaftliches Arbeiten

Wie Kartendienste zum Ziel finden

Johannes Quast

12. Juli 2020

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Ich glaube jeder von uns kennt die folgende Situation:
Man möchte mit dem Auto oder Verkehrsmittel seiner Wahl an einen bestimmten Ort gelangen, an dem man vorher noch nie gewesen ist. Die Lösung dieses Problems heißt heutzutage in den meisten Fällen „Google Maps“ oder trägt den Namen einer anderen populären Kartenapp. Kaum hat man eine beliebe Adresse in ein Suchfeld eingegeben, bekommt man bereits nach wenigen Sekunden eine Route vorgeschlagen.

In Zeiten der Digitalisierung, in der solche Apps für uns selbstverständlich geworden sind und klassische Navigationsmethoden wie z.B. Atlanten oder Karten des Straßennetzes beinahe ausgestorben sind, fragt sich kaum jemand, wie solche Routenplaner eigentlich funktionieren. In dieser Ausarbeitung möchte ich genau auf diese Hintergründe eingehen und versuchen **möglichst anschaulich** zu erklären, wie eine solche Routenberechnung eigentlich funktioniert und welche einfachen Möglichkeiten es gibt, in großen Straßennetzen zu navigieren.

1.2 Vorwort

Bevor man sich mit der Routenplanung in einem Straßennetz im speziellen beschäftigt, sollte man dieses Problem auf ein etwas allgemeineres reduzieren: Die Wegfindung von einem Punkt X zu einem Punkt Y. Es sollte natürlich selbsterklärend sein, dass Punkt X und Y meistens keine direkte Verbindung zueinander haben, sondern die Route über zahllose Zwischenpunkte gehen wird.

Um dieses Problem zu lösen gibt es sehr viele Ansätze, aber nicht alle sind praktikabel aufgrund verschiedener Limitationen der jeweiligen Ansätze. Auf den folgenden Seiten werden wir einige Möglichkeiten betrachten, sodass am Ende klar sein sollte, welche Möglichkeiten für eine Routenplanung in einem Straßennetz in Frage kommen.

2 Grundlagen

2.1 Graphen

Fast genauso wichtig wie das Verständnis für den Algorithmus ist ein grober Überblick darüber, auf welchen Daten bzw. Datenstrukturen unser Algorithmus arbeiten wird. Die Datenstruktur, die für diesen Anwendungszweck in Frage kommt nennt sich *Graph*. Vereinfacht gesagt besteht ein Graph aus Knoten (z.B. Punkt X oder Y) und Kanten (Verbindungen zwischen den Knoten). [Wik20c]

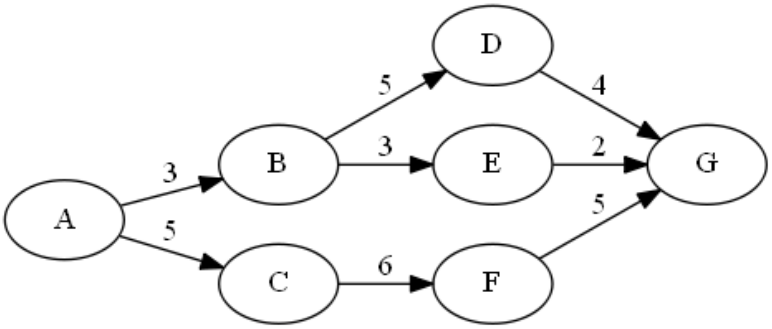

Abbildung 2.1: Ein einfacher Graph mit Knoten (A,B,C,...) und Kanten

Wie man in der Abbildung sehen kann, sind alle Kanten zusätzlich mit Zahlen versehen. Diese geben meistens eine Art *Gewichtung* o.ä. an. Im Kontext der Wegfindung können diese Zahlen zum Beispiel den Aufwand angeben, den jemand benötigt um z.B. von Punkt A zu B zu gelangen. In der realen Welt kann dieser Aufwand beispielsweise mit der Länge der Strecke oder Fahrzeit verbunden werden.

2.2 Prioritätenliste (priority queue)

Diese Datenstruktur ist wichtig für viele Algorithmen, die den kürzesten Weg finden möchten. Wie der Name bereits vermuten lässt, handelt es sich hierbei um eine Warteschlange, die bestimmte Objekte speichern kann. Diese werden nach einer bestimmten *Priorität* geordnet.

Abbildung 2.2: Objekte mit Fahrzeit und Entfernung zwischen zwei Städten

Würde man diese 3 Objekte nun nacheinander in eine Prioritätenliste einfügen, dann würden diese sich automatisch an die richtige Position in der Liste einreihen.

Abbildung 2.3: Objekte in Prioritätenliste: Niedrigere Entfernung = höhere Priorität

3 Algorithmen

Auf den folgenden Seiten möchte ich mich auf zwei sehr bekannte Algorithmen beschränken, da diese ein sehr gutes Verständnis dafür liefern, wie eine Routenberechnung implementiert werden kann und wie die Wegfindung vorgeht.

3.1 Dijkstra

Der erste ist der Dijkstra Algorithmus, auf dem auch A* basiert. Er arbeitet auf einem Graphen, um die kürzeste Route zwischen einem gegebenen Startknoten und einem bestimmten Zielknoten zu ermitteln. Jede Kante im Graphen muss dabei eine nicht negative Gewichtung aufweisen.

3.1.1 Allgemeine Funktionsweise

1. Bevor die eigentliche Wegfindung beginnt, wird die Datenstruktur initialisiert. Dabei wird jedem Knoten in unserem Graphen folgende zwei Eigenschaften zugeordnet: Entfernung und Vorgänger. Die Entfernung gibt dabei die Distanz zum Startknoten an. Es ist an dieser Stelle zu erwähnen, dass der Begriff „Entfernung“ hier stark von der verwendeten Kantengewichtung abhängt, denn nicht immer ist die Einheit dieser Werte ein Weg o.ä. Alle Knoten haben eine initiale Entfernung von ∞, da am Anfang der Suche noch nicht sichergestellt ist, ob ein bestimmter Knoten vom Startknoten überhaupt erreichbar ist. Der Startknoten selbst bekommt die Entfernung 0 zugeordnet. [Wik20b]

2. In diesem und den folgenden Schritten kommt die zuvor erklärte *Prioritätenliste* zum Einsatz. Alle Knoten werden in dieser Liste gespeichert und aufsteigend nach der Entfernung sortiert.

   a) Nun beginnt man, für alle Nachbarknoten des aktiven Knotens (am Anfang ist dies der Startknoten) die Entfernungen zu berechnen.

$$Enterfnung_{Knoten_{Nachbar}} = Enterfnung_{Knoten_{Aktiv}} + Kantengewicht$$

Diese berechnete Entfernung wird nur gesetzt, wenn die bereits gesetzte Entfernung des Nachbarknotens größer als die neue ist. Sollte dies der Fall sein, wird dieser Wert aktualisiert und das Attribut *Vorgänger* wird auf den aktiven Knoten gesetzt. [Wik20b]

b) Der momentan aktive Knoten wird nun der Liste der *besuchten Knoten* hinzugefügt. Dies hat den einfachen Hintergrund, dass der Algorithmus nicht mehrmals den selben Knoten untersucht, oder ein Zirkel entsteht (z.B. wenn der Algorithmus auf einen Kreisverkehr trifft).

c) Jetzt beginnt die Schleife erneut bei Schritt a. Allerdings mit dem wichtigen Unterschied, dass der aktive Knoten mit dem ersten Knoten aus der *Prioritätenliste* ersetzt wird. [Wik20b] Kurz zur Erinnerung: Der 1. Knoten dieser Warteschlange ist der Knoten, der momentan die geringste Entfernung zum Startknoten ausweist. Die Schleife **terminiert**, wenn der aktive Knoten gleich dem gesuchten Zielknoten entspricht.

3.1.2 Beispiel

Für das Beispiel verwenden wir den 1. Beispielgraphen. Der Startknoten in diesem Fall ist A, der Zielknoten G. Der momentan aktive Knoten wird blau markiert, die bereits besuchten Lila. Orange sind die Nachbarknoten, für die gerade die Entfernung berechnet wurde (Vergleich Schritt c. der allgemeinen Funktionsweise). Weiterhin werden die Einträge der Prioritätenliste visualisiert, indem jeweils der Name des Knoten selbst, die Entfernung zum Startknoten und über welchen Vorgängerknoten man den betrachteten Knoten erreichen kann, dargestellt wird.

1. Schritt

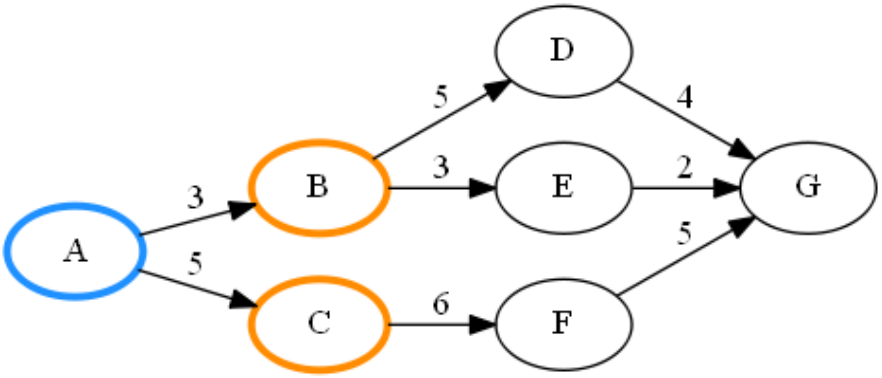

Abbildung 3.1: Graph während Schritt 1: A ist aktiver Knoten, B und C Nachbarknoten

B	
3	A

C	
5	A

Abbildung 3.2: Prioritätenliste nach Schritt 1

Der Knoten A ist in diesem Fall der Startknoten und hat die Nachbarknoten B und C. Für beide Nachbarn wird nun die Entfernung berechnet, ausgehend von A.

$$Entfernung_B = Entfernung_A + 3 = 0 + 3 = 3$$
$$Entfernung_C = Entfernung_A + 5 = 0 + 5 = 5$$

2. Schritt

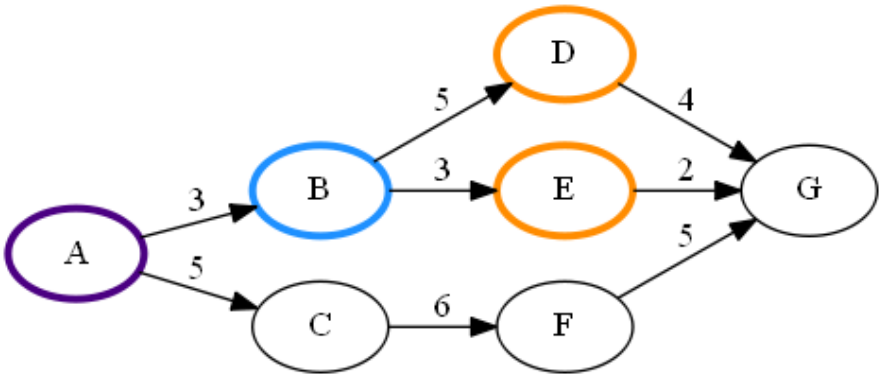

Abbildung 3.3: B ist aktiver Knoten. Für D und E wird Entfernung berechnet. A bereits betrachtet.

C		E		D	
5	A	6	B	8	B

Abbildung 3.4: Prioritätenliste nach Schritt 2

Da der Knoten B in der Prioritätenliste nach Schritt 1. der erste war, wurden die Schritte nun bei diesem wiederholt.

$$Entfernung_D = Entfernung_B + 5 = 3 + 3 = 6$$
$$Entfernung_E = Entfernung_B + 5 = 3 + 5 = 8$$

3. Schritt

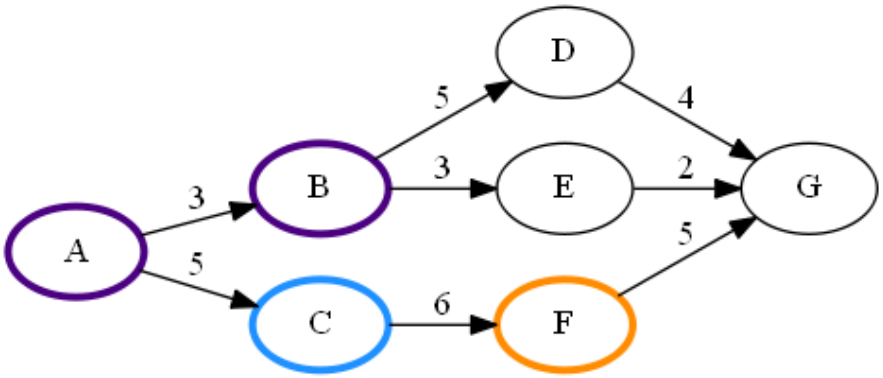

Abbildung 3.5: C ist aktiver Knoten. F wird berechnet. A und B bereits betrachtet

E	
6	B

D	
8	B

F	
11	C

Abbildung 3.6: Prioritätenliste nach Schritt 3

Obwohl C die höchste Anfangsdistanz zum Startknoten aufweist, ist er in der Gesamtdistanz zu A nach dem 2. Schritt an die erste Stelle der Prioritätenliste gerutscht, da der Algorithmus mit den gegebenen Daten nicht wissen kann, dass der Graph sowohl zwischen C und F, als auch zwischen F und G sehr hohe Werte für die Gewichtung aufweist.

$$Entfernung_F = Entfernung_C + 6 = 5 + 6 = 11$$

4. Schritt

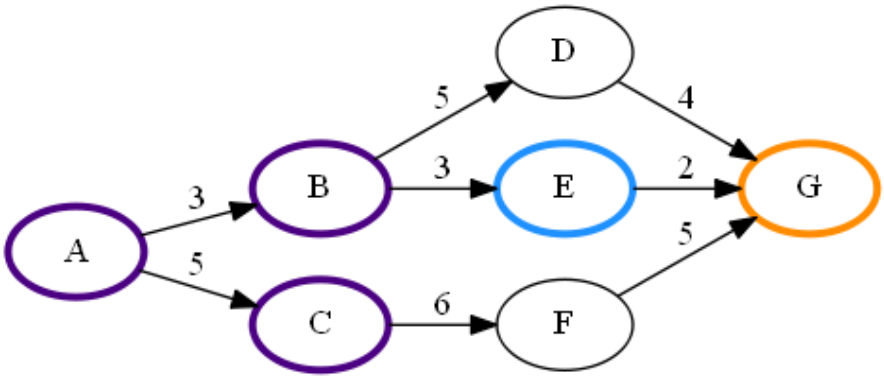

Abbildung 3.7: E ist aktiver Knoten. G wird berechnet. A, B, C bereits betrachtet.

Für diesen letzten Schritt stand der Knoten E als erster in der Prioritätenliste. In diesem Fall hat E nur einen Nachbarknoten: den Zielknoten G. An diesem Punkt bricht die Schleife ab, nachdem die Entfernung und der Vorgänger bei G gesetzt worden sind.

$$Entfernung_G = Entfernung_E + 2 = 6 + 2 = 8$$

Über genau diese Vorgängerbeziehung zwischen den Knoten kann man nun den kürzesten Weg rekonstruieren. In diesem Beispiel wäre der kürzeste Pfad A $\rightarrow$ B $\rightarrow$ E $\rightarrow$ G.

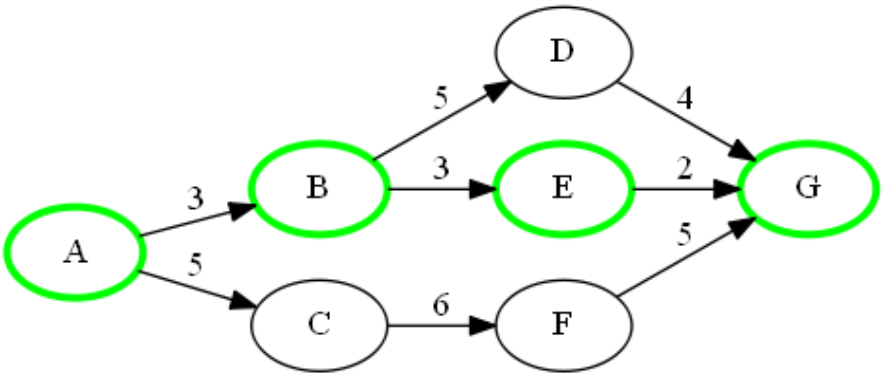

Abbildung 3.8: Kürzester Pfad

Würde man den Pfad über die Beziehungen selbst darstellen, würde dies folgendermaßen aussehen.

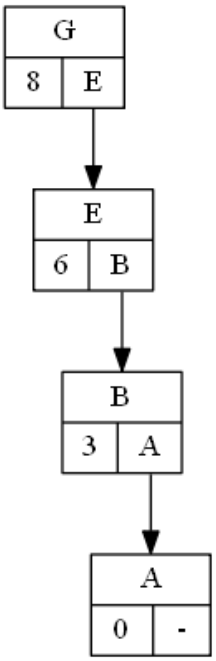

Abbildung 3.9: Kürzester Pfad über Vorgängerbeziehung

3.2 A*

3.2.1 Problemstellung

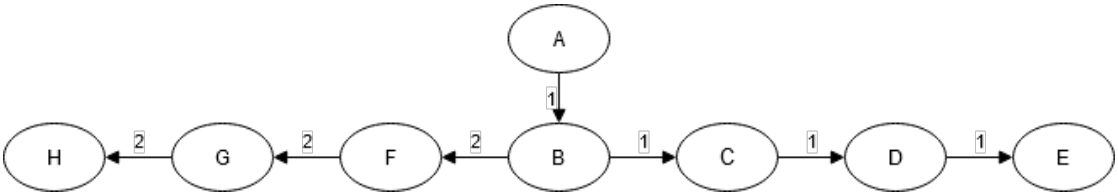

Abbildung 3.10: Veranschaulichung einer Kreuzung mit zwei abgehenden Straßen

Die Kreuzung aus Abbildung 3.10 besitzt von A kommend zwei abgehende Straßen. Die nach rechts führenden Streckenabschnitte besitzen alle eine Gewichtung von eins, da in diesem Szenario keine Behinderung für Verkehrsteilnehmer vorliegt. Die nach links führenden Streckenabschnitte weisen in diesem Fall einen leichten Stau auf und besitzen deshalb einen Gewichtung von zwei (höherer Aufwand um die Strecke zu befahren). [You] Der Startknoten ist A, der Zielknoten H.

Wendet man nun Dijkstra auf diesen Graphen an, dann wird man schnell feststellen, dass der Algorithmus den rechten Streckenabschnitten eher nachgehen wird, da dort jede Kante eine geringere Gewichtung aufweist, obwohl der Zielknoten in der genau entgegengesetzten Richtung liegt. Dieses Verhalten lässt sich dadurch erklären, dass Dijkstra nur die Entfernung zum Startknoten verwendet, um den Graphen zu durchsuchen, aber der Algorithmus dadurch kein Verständnis für Richtungen aufweist.

3.2.2 Funktionsweise

Um genau dieses Problem von Dijkstra zu lösen, verwendet der A* Algorithmus einen leicht anderen Ansatz. In der Prioritätenliste von A* wird als Priorität statt der Entfernung vom Startknoten (wie bei Dijkstra) folgende Heuristik verwendet. [Wik20a]

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

$g(x)$ ist die Entfernung des betrachteten Knotens vom Startknoten aus. Dieser Teil der Heuristik ist genau die Gewichtung, die auch Dijkstra verwendet. [Wik20a]

$h(x)$ ist der Bestandteil, den A* zur Berechnung hinzufügt. Diese Funktion gibt die eine *Schätzung* an, wie viel es, ausgehend vom betrachteten Knoten im günstigsten Fall

„kostet“, um zum Ziel zu gelangen.[Wik20a]

$f(x)$ wird dem betrachteten Knoten zugeordnet und die gibt die *Gesamtkosten* an, um zum Ziel zu gelangen, wenn der Pfad durch den betrachteten Knoten verläuft. [Wik20a]

Setzt man $h(x)$ auf 0, dann verhält sich A* gleich dem Dijkstra Algorithmus. Für eine Routenfindung verwendet man häufig als $h(x)$ die Luftlinie zwischen dem betrachteten Knoten und dem Zielknoten.

Benutzt man nun für das Beispiel der Problemstellung statt dem Dijkstra den A* Algorithmus mit einer Luftlinie als Heuristik [Wik20a], dann würden die $h(x)$ für die Knoten des rechten Streckenabschnitts ansteigen und damit die Gesamtkosten, während die Schätzung für den linken Abschnitt immer kleiner wird. Somit umgeht man das beschriebe Problem für dieses konkrete Beispiel.

4 Mögliche Verbesserung

Die vorherigen Beispiele für die Routenberechnung waren in der Anzahl ihrer Knoten sehr begrenzt. Schaut man sich nun allerdings ein typisches Alltagsszenario an, dann wird schnell klar, dass ein solcher Algorithmus normalerweise auf einem Graphen mit tausenden Knoten arbeiten wird (z.B. Navigation in Großstädten oder größere Entfernungen).

In einem großen Straßennetz wie z.B. von Deutschland, wäre es eine Möglichkeit bei der Routenfindung über größere Distanzen, Autobahnen normalen Landstraßen der Bundesstraßen zu bevorzugen, da man so meistens deutlich schneller ans Ziel gelangt.

Auch könnte man eine Art Vorverarbeitung des Graphen vornehmen, indem man sehr wenig befahrene Straßen (z.B. Feldwege) von vornherein nicht berücksichtigt.

Weiterhin helfen zusätzliche Daten wie z.B. Geschwindigkeitslimits, Echtzeit Verkehrsdaten oder statistische Auswertung des Verkehrs zu bestimmten Wochentagen/Uhrzeiten bei der Wahl der schnellsten Route, da sich der Verkehr ständig verändert.

5 Fazit

A* und Dijkstra sind zwei sehr prominente und vielseitig einsetzbare Algorithmen für die Wegfindung in verschiedensten Situation. Natürlich gibt es noch weitere Algorithmen, die allerdings hier nicht weiter betrachtet wurden und ihre eigenen individuellen Stärken und Schwächen besitzen. Besonders in Situationen in denen ein Verständnis von *Richtungen* sehr wichtig ist, wie z.B. bei typischen Wegfindungen in Straßennetzen, eignet sich A* hervorragend. In anderen Anwendungsbereichen in denen dieses Kriterium keine hohe Priorität hat oder die Daten für eine solche Heuristik fehlen, ist Dijkstra die bessere Alternative.

Literatur

[Wik20a] Wikipedia. *A*-Algorithmus — Wikipedia, The Free Encyclopedia*. http://de.wikipedia.org/w/index.php?title=A*-Algorithmus&oldid=198870453. [Online; Aufgerufen am 11. Juli 2020]. 2020.

[Wik20b] Wikipedia. *Dijkstra-Algorithmus — Wikipedia, The Free Encyclopedia*. http://de.wikipedia.org/w/index.php?title=Dijkstra-Algorithmus&oldid=200375007. [Online; Aufgerufen am 10. Juli 2020]. 2020.

[Wik20c] Wikipedia. *Graphentheorie — Wikipedia, The Free Encyclopedia*. http://de.wikipedia.org/w/index.php?title=Graphentheorie&oldid=200172892. [Online; Aufgerufen am 11. Juli 2020]. 2020.

[You] *Dijkstra's Algorithm - Computerphile*. [Online, Aufgerufen am 08. Juli 2020]. URL: https://www.youtube.com/watch?v=GazC3A4OQTE.