Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

2 Theoretische und technische Grundlagen

2.1 Datenstrukturen

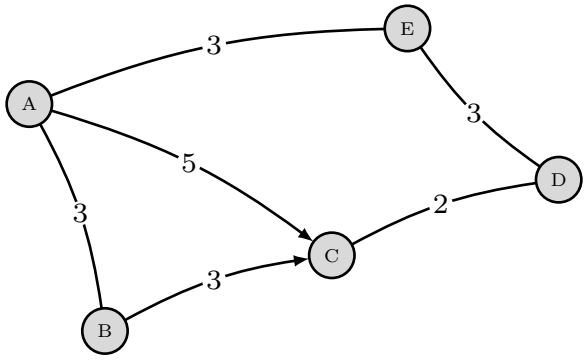
2.1.1 Graph

Um das kürzeste-Wege-Problem zu lösen, muss zunächst das reale Straßennetz in eine abstrakte Form gebracht werden. Hierzu wird das Straßennetz als Graph modelliert. Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge von Kanten E. Jede Kante $e = (u, v) \in E$ verbindet zwei Knoten $u, v \in V$. Knoten bilden Kreuzungen ab und Kanten Straßensegmente zwischen zwei Kreuzungen. Ein Graph kann als gerichtet oder ungerichtet definiert werden. Bei einem ungerichteten Graphen sind die Kanten bidirektional und können in beide Richtungen durchlaufen werden, während bei einem gerichteten Graphen die Kanten nur in eine Richtung durchlaufen werden können.

Der Graph in dieser Arbeit ist gerichtet und gewichtet. Die Kantenrichtung spiegelt dabei die Richtung des Verkehrs wieder, d. h. sie gibt an, ob eine Straße in einer bestimmten Richtung befahren werden darf oder nicht. Zusätzlich wird auf jede Kante eine Kostenfunktion angewandt, die dann ein Gewicht $w(e)$ festgelegt. In der Routenplanung ist normalerweise nicht die kürzeste Strecke von Interesse, sondern die Strecke mit der kürzesten Reisezeit. Diese lässt sich aus der Länge des Straßensegments S und der maximal erlaubten Geschwindigkeit V auf dieser Straße $t = \frac{V}{S}$ in Sekunden berechnen.

Es gibt generell zwei Möglichkeiten, einen Graphen als Datenstruktur darzustellen. Eine davon ist die Verwendung von Adjazenzlisten. Dabei wird für jeden Knoten $u \in V$ eine Liste der benachbarten Knoten bzw. ausgehenden Kanten $e(u, v) \in E$ gespeichert. Die Summe der Länge aller Adjazenzlisten in einem gerichteten Graph entspricht der Anzahl an Kanten $|E|$, bzw. $2|E|$ in einem ungerichteten Graph. Der Speicherverbrauch für die Adjazenzliste ist damit $\Theta(|V| + |E|)$ [2].
Die zweite Möglichkeit ist die Verwendung einer Adjazenzmatrix. Hier wird eine zweidimensionale Matrix der Größe $|V| \times |V|$ verwendet, bei der die Zeilen und Spalten den Knoten entsprechen. Der Eintrag an Position (i, j) in der Matrix gibt an, ob eine Kante zwischen den Knoten i und j existiert. Unabhängig von der Anzahl an Kanten ist der Speicherverbrauch $\Theta(|V|^2)$ [2]. Ein Beispiel für beide Darstellungen anhand eines einfachen Graph befindet sich in Abbildung 1.

Für diese Arbeit werden Adjazenzlisten verwendet, da zum einen Straßennetze dünne Graphen sind ($|E| << |V|^2$), und damit die Speichereffizienz gegenüber einer Adjazenzmatrix deutlich effizienter ist. Zum anderen ist für es wichtig, für den Aufbau der Contraction Hierarchies (CHs) und der Suche, schnell auf alle Nachbarn eines Knotens zuzugreifen zu können. Der Zugriff auf eingehende Kanten ist allerding nur sehr aufwendig möglich, wird aber für den Aufbau der Kontraktionshierarchien benötigt. Das Problem lässt sich jedoch lösen indem auch Adjazenzlisten für alle eingehenden Kanten eines Knotens angelegt werden. Damit erhöht sich der Speicherverbrauch auf $\Theta(|V|+2|E|)$, was aber immer noch deutlich effizienter ist als eine Darstellung als Adjazenzmatrix.



(a)

	A	B	C	D	E
A	0	3	5	0	3
B	3	0	3	0	0
C	0	0	0	2	0
D	0	0	2	0	3
E	3	0	0	3	0

(b)

A →	B	C	E
B →	A	C	
C →	D		
D →	C	E	
E →	A	D	

(c)

Abbildung 1: Der oben gezeigte gerichtete Graph (a) dargestellt als Adjazenzmatrix(b) oder Adjazenzliste (c). Viele Einträge der Matrix sind 0, da der Graph dünn ist, d. h. $|E|$ um einiges kleiner ist als $|V^2|$.

2.1.2 Vorrangwarteschlangen

Eine Vorrangwarteschlange (eng. Priority Queue) (PQ) ist eine Datenstruktur, in der nur auf das Element mit der höchsten Priorität zugegriffen werden kann. Eine PQ unterstützt in der Regel die folgenden Operationen:

1. Einfügen (Push): Ein Element wird mit seiner zugehörigen Priorität in die Warteschlange eingefügt. Das Element wird entsprechend seiner Priorität platziert.

2. Entfernen (Pop): Das Element mit der aktuell höchsten Priorität wird aus der Warteschlange entfernt und zurückgegeben.

PQs werden allen nachfolgenden Suchalgorithmen verwendet, daher hat die Implementierung der Warteschlange einen großen Einfluss auf die Laufzeit der Algorithmen. In der Arbeit wird die Implementierung als binärern Min-Heap verwendet mit einer Zeitkompexität für das Einfügen von $\theta(1) \sim$ und für das Entfernen von $\theta(\log n)$.

2.2 Kürzeste-Wege-Algorithmen

Grundsätzlich arbeiten kürzeste-Wege-Algorithmen daran, den kürzesten Weg zwischen einem Startknoten s und einem Zielknoten t in einem gewichteten Graph zu finden. Der kürzeste Weg P bezieht sich dabei auf den Weg mit dem geringsten Gesamtgewicht $dist(s,t)$, der sich aus der Summe der Kosten zum Überqueren der einzelnen Kanten ergibt. Neben dem Point-to-Point-kürzeste-Wege-Problem existieren noch weitere Varianten, wie das One-to-Many-Problem, bei dem der kürzeste Weg von einem Knoten s zu allen anderen Knoten im Graphen gesucht wird, oder das Many-to-Many-Problem, bei dem jeder kürzeste Weg zwischen einer Knotenmenge S und einer Knotenmenge T gesucht wird [1]. In dieser Arbeit wird sich hauptsächlich auf das Point-to-Point-Problem fokusiert.

2.2.1 Grundlegende Technik

Der Algorithmus von Dijkstra löst das One-to-Many-Problem auf einem gewichteten gerichten Graphen. Dabei ist zu beachten dass alle Kanten $e \in E$ nicht-negativ gewichtet sind, also $e(u,v) \geq 0$.
Der Algorithmus besteht aus einer Initialisierungsphase, in der die Kosten *dist* für alle Knoten auf den Wert ∞ gesetzt werden. Um am Ende nicht nur die Kosten, sondern auch den Weg zu erhalten wird zusätzlich der direkte Vorgängerknoten für jeden Knoten in P gespeichert. In folgender Implementierung wird eine Min-Vorrangwarteschlange Q verwendet, in der zu Beginn der Startknoten s mit Kosten 0 eingefügt wird. In der Hauptschleife wird nun solange ein Knoten $u \in V - S$ aus der Warteschlange entnommen, bis diese leer ist oder der Zielknoten erreicht wurde. Wenn ein Nachbarknoten v von u noch nicht besucht wurde, werden die Kosten $dist(v)$ aktualisiert, falls der Weg über u kürzer ist. Der Vorgänger $P(v)$ wird ebenfalls aktualisiert und der Knoten v in die Warteschlange eingefügt. Wenn der Zielknoten gefunden wurde, dann wird der kürzeste Weg mithilfe der gemerkten Vorgängerknoten ausgehend von t rekonstruiert und zurückgegeben. Durch Weglassen des Abbruchskrieterium in Zeile 8–10

kann der Algorithmsu von einer Point-to-Point suche zu einer One-to-Many Suche erweitert werden, sodass jeder Knoten im Graph traversiert wird.

Algorithm 1 Dijkstra Point-To-Point

```
1:  function DIJKSTRA(G=(V,E), s,t)
2:      Q ← ∅
3:      dist(s) = 0
4:      P ← ∅                                  ▷ Vorgängerknoten für Pfadrekonstruktion
5:      Q ← Q ∪ {s}                                        ▷ Prioritätswarteschlange
6:      while Q ≠ ∅ do
7:          u =← pop(Q)
8:          if u = t then
9:              return shortest_path(P,t)
10:         end if
11:         for each ausgehende Kante e(u,v) ∈ Adj[u] do
12:             tentative_distance ← dist(u) + w(u,v)
13:             if tentative_distance < dist(v) then
14:                 dist(v) ← tentative_distance
15:                 P(v) ← u
16:                 Q ← Q ∪ {v}
17:             end if
18:         end for
19:     end while
20: end function
```

Die Laufzeit des Algorithmus hängt hauptsächlich von der Implementierung der Prioritätswarteschlange ab. Mit der verwendeten Implementierung ergibt sich eine Laufzeit von $\Theta(|E| + |V| \log |V|)$, wenn Adjazenzlisten verwendet werden [2].

2.2.2 Bidirektionale Suche

Eine Variante des klassischen Dijkstra-Algorithmus ist die Erweiterung um Bidirektionalität. Im Gegensatz zum herkömmlichen Algorithmus, der nur in einer Richtung vom Startknoten zum Zielknoten arbeitet, führt der bidirektionale Dijkstra-Algorithmus eine Suche von beiden Knoten gleichzeitig durch. Die Rückwärtssuche traversiert dabei den transponierten Graphen, d. h. die Richtung der Kanten sind invertiert.
Der Algorithmus verwendet zwei Prioritätswarteschlangen, eine für die Vorwärtsrichtung und eine für die Rückwärtsrichtung. Der Ablauf des Algorithmus ist in Abbildung 2 illustriert.

1. Initialisierung: Jede Prioritätswarteschlange wird mit dem Startknoten bzw. dem Zielknoten initialisiert. Die vorläufigen Werte werden auf 0 für den Startknoten bzw. auf

unendlich für den Zielknoten gesetzt.

2. Expansionsschritt: In jedem Schritt wird der Knoten mit dem kleinsten vorläufigen Wert von beiden Warteschlangen ausgewählt. In der Schlange mit dem kleineren Knoten wird eine Dijkstra-Iteration ausgeführt.

3. Überprüfung des Treffpunkts: Bei jedem Expansionsschritt wird überprüft, ob der aktuelle Knoten in beiden Richtungen erreicht wurde. Wenn ein Knoten in beiden Richtungen erreicht wurde, gibt es einen potenziellen Pfad von Start zu Ziel.

4. Terminierung: Der Algorithmus terminiert, wenn beide Prioritätswarteschlangen leer sind oder ein Treffpunkt gefunden wurde.

5. Kürzester Weg: Für jeden Knoten u der von beiden Seiten erreicht wurde (u besitzt eine endliche Distanz von s und t), wird die Summe $dist(s,t) = dist(s,u) + dist(t,u)$ berechnet. Der kürzeste Weg verläuft über den Knoten, bei dem die Summe minimal ist.

6. Rückverfolgung des kürzesten Pfades: Wenn ein Treffpunkt u gefunden wurde, kann der kürzeste Pfad durch Rückverfolgung der Vorgängerknoten von Startknoten bis zum Treffpunkt in der Vorwärtsrichtung und von Zielknoten bis zum Treffpunkt in der Rückwärtsrichtung rekonstruiert werden.

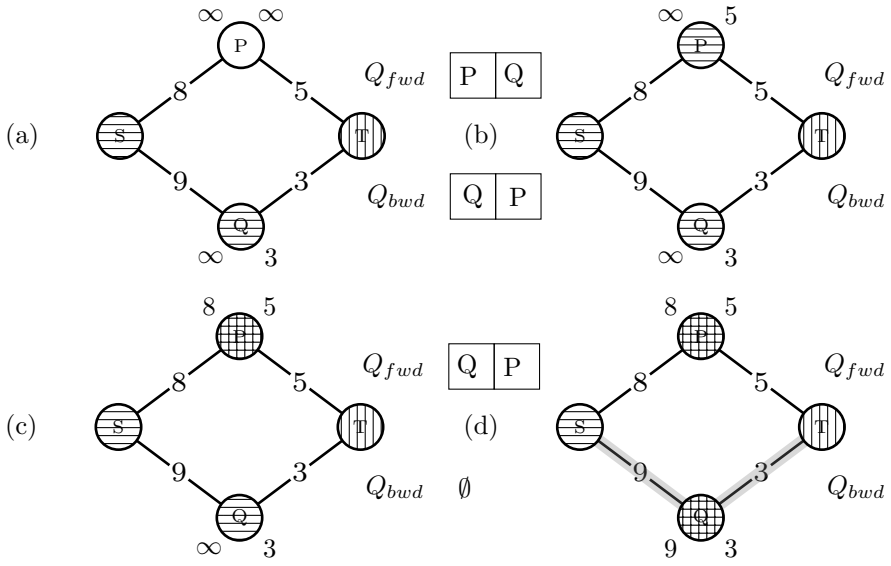


Abbildung 2: Bidirektionale Suche von S nach T: In Schritt (a) und (b) wird zuerst Knoten Q und P von der Rückwärtssuche relaxiert. In den Schritten (c) und (d) wird P und Q von der Vorwärtssuche relaxiert. Obwohl P der erste Knoten ist, der von beiden Suchen relaxiert wurde, ist der Weg über Q am kürzesten.

Der bidirektionale Dijkstra Algorithmus kann die Anzahl der untersuchten Knoten im Vergleich zum herkömmlichen Dijkstra reduzieren, insbesondere in großen Graphen. Durch die gleichzeitige Suche in beiden Richtungen kann er die Laufzeit verbessern, indem er die Anzahl der Expansionsschritte und die Anzahl der Knoten, die in Betracht gezogen werden, reduziert. In der Praxis wird der Suchraum etwa um die Hälfte reduziert (siehe Abbildung 3):

$$A = \pi r^2$$
$$A_{bidir} = 2\pi(\frac{r}{2})^2 = \frac{A}{2}$$

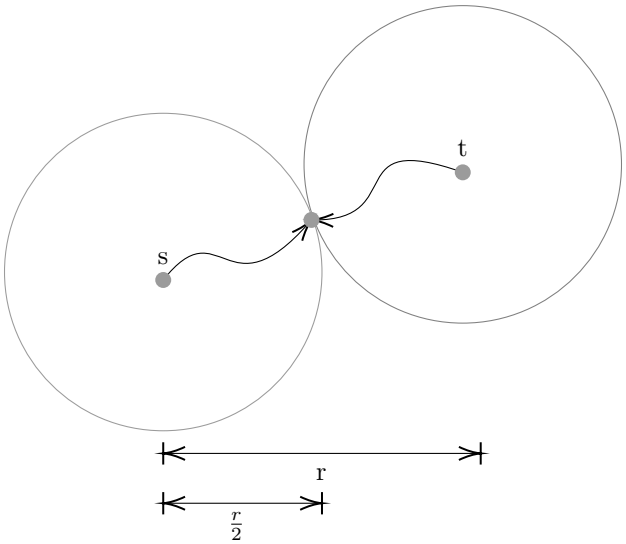


Abbildung 3: Suchraum einer bidirektioanlen Suche. Die Anzahl der Knoten die untersucht werden müssen, halbiert sich.

Korrektheitsbeweis

Lemma 2.1. *Für alle Knoten u die von beiden Seiten erledigt wurden, gilt:*
$$dist(s,t) = min\{dist(s,u) + dist(t,u)\}.$$

Beweis. Gegeben sei der Graph in Abbildung 4. Die Länge des kürzesten Pfads von s nach t wird als D bezeichnet. Der Knoten p ist der erste Knoten der von beiden Suchen erledigt wurde. Wenn gilt $dist(s,p) = dist(t,p)$, dann ist p garantiert auf dem kürzesten Weg.
Angenommen $dist(s,p)$ und $dist(t,p)$ sind ungleich D/2, dann muss entweder $dist(s,p) < D/2$ oder $dist(t,p) < D/2$ gelten. Dies wiederum bedeutet , dass alle Knoten mit einem kürzesten-Pfad-Wert, der kleiner oder gleich $D/2$ ist, bereits gesetzt wurden.

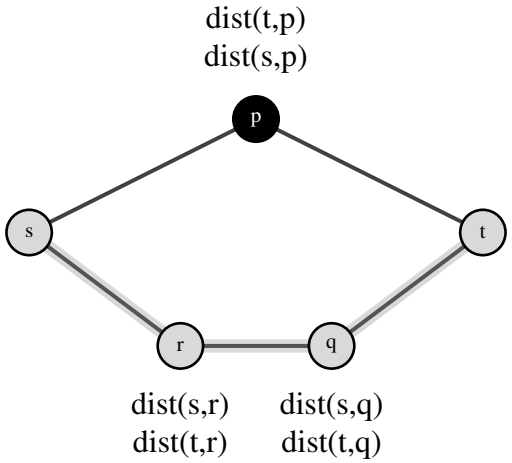


Abbildung 4: Graph für den Korrektheitsbeweis der bidirektionalen Suche

Knoten r und q liegen auf dem kürzesten Weg von s nach t und $dist(s,r) \leq D/2$ und $dist(t,q) \leq D/2$. Knoten r wurde von s erledigt und Knoten q von t. Dann wurde die Kante $e(r,q)$ bereits von beiden Seiten aus relaxiert uns somit haben r und q Distanzwerte von beiden Richtungen erhalten.
Die länge des kürzesten Weges von s nach t ist der kleinere Wert aus $dist(s,r) + dist(t,r)$ und $dist(s,q) + dist(t,q)$, welche in diesem Fall den gleichen Wert haben.
Somit ist der Knoten, der zuerst von beiden Seiten erledigt wurde nicht unbedingt auf dem kürzesten Weg, aber es wurde zumindest ein Knoten mit einer kürzeren Distanz gefunden, der auf dem kürzesten Weg liegt und einen Distanzwert von beiden Seiten erhalten hat. □

2.2.3 A*

Der A*-Algorithmus („A-Stern“) nach [5] ist eine Erweiterung und Verallgemeinerung des Dijkstra-Algorithmus. Das Gewicht jeder Kante wird mit einer Potentialfunktion ρ neu definiert, die vom Ziel t abhängt, sodass gilt:

Definition 2.1. $\bar{w}(u,v) = w(u,v) - \rho_t(u) + \rho_t(v)$

Kanten, die vom Ziel wegführen werden länger und Kanten die zum Ziel führen werden kürzer. Nachdem ein Potential gewählt wurde und alle Kanten neu gewichtet wurden, kann Dijkstra wie gewohnt auf dem neuen Graph ausgeführt werden. Durch ein gut gewähltes Potential wird die Suche in die Richtung des Ziels gelenkt und somit die Laufzeit verringert.

Abbildung 5 zeigt einen Graphen, in dem das Potential jedes Knotens als Höhe dargestellt ist. Knoten die weg vom Ziel führen haben ein höheres Potential und sind schwerer zu erreichen,

da mehr Arbeit aufgebracht werden muss, um nach oben zu klettern. Da die Bewegung nach unten bevorzugt wird, ist es daher wahrscheinlicher das Ziel früher zu treffen.

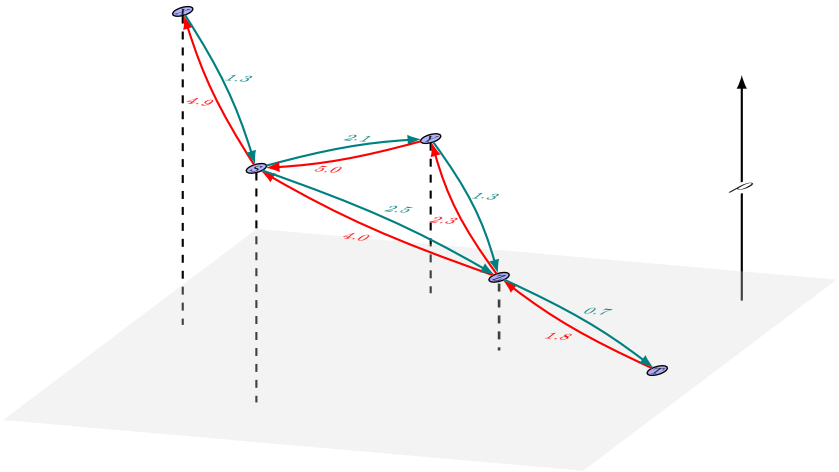


Abbildung 5: Graph nach Hinzufügen der Potentialfunktion ρ. Kanten die nach „oben“ verlaufen besitzen ein höheres Gewicht als Kanten die nach „unten“ verlaufen und sind dadurch weniger attraktiv.

Da der Dijkstra-Algoritmus nur mit nicht negativen Kantengewichten umgehen kann, muss ρ so gewählt sein, dass gilt:

Definition 2.2. In einem gewichteten Graph $G = (V, E)$ ist ein Potential ρ_t möglich, bei dem jede Kante $e(u,v) \in E$ die Bedingung $\rho_t(u) \leq w(u,v) + \rho_t(v)$ erfüllt. Damit wird außerdem für jeden Knoten $u \in V$ die Voraussetzung $\rho_t(u) \leq dist(u,t)$ impliziert.

Die Potentialfunktion ist damit eine optimistische Schätzung und untere Schranke. Sie schätzt den Weg von u nach t immer kleiner oder gleich dem tatsächlichen kürzesten Weg ein und verhindert damit, dass Kanten negative Gewichte erhalten können.
Da die Koordinaten der Knoten im Graphen eine Straßennetzes in der Regel bekannt sind, ist die euklidische bzw. Großkreisdistanz als Potential eine gute Wahl:

$$\rho_t(u) = \sqrt{(x_t - x_u)^2 + (y_t - y_u)^2} \tag{1}$$

Der einzige Unterschied zu Dijkstra findet sich in Zeile 7 (2) wieder. Hier wird nicht mehr das Element mit der niedrigsten Distanz extrahiert sonder das Element mit der niedrigsten Distanz plus dem Potential. Wenn für Potential die Nullheuristik gewählt wird, dann ist der Algorithmus identisch zu Dijkstra.

Algorithm 2 AStar Implementierung

```
1:  function ASTAR(G = (V,E), s, t, ρ_t)
2:      Q ← ∅
3:      P ← ∅                                   ▷ Vorgängerknoten für Pfadrekonstruktion
4:      dist(s) ← 0
5:      Q ← Q ∪ {s}
6:      while Q ≠ ∅ do
7:          u ← node in Q minimizing dist(node) + ρ_t(node)      ▷ Potential berüclsichtigen
8:          if u = t then
9:              return shortest_path(P,t)
10:         end if
11:         for v ∈ adj(u) do
12:             tentative_distance ← dist(u) + w(u,v)
13:             if tentative_distance < dist(v) then
14:                 dist(v) ← tentative_distance
15:                 P(v) ← u
16:                 Q ← Q ∪ {v}
17:             end if
18:         end for
19:     end while
20: end function
```

2.3 OpenStreetMap

OpenStreetMap (OSM) ist ein Projekt, das sich der Erstellung und Bereitstellung von freien geografischen Daten verschrieben hat. Es handelt sich dabei um Daten die von einer weltweiten Gemeinschaft von Freiwilligen erstellt und gepflegt wird [3].

Das grundlegende Konzept von OpenStreetMap basiert auf OpenData, was bedeutet, dass die Daten frei verfügbar und für jeden zugänglich sind. Im Gegensatz zu kommerziellen Kartenanbietern, die ihre Daten schützen und für den Zugriff hohe Gebühren verlangen, ermutigt OpenStreetMap Menschen dazu, ihre eigenen Daten beizutragen und von den vorhandenen Daten zu profitieren.

Das OpenStreetMap-Projekt verwendet eine Kombination aus verschiedenen Datenquellen, um eine detaillierte und umfassende Karte zu erstellen. Dazu gehören zum Beispiel Satellitenbilder, GPS-Tracks, Luftaufnahmen und auch öffentlich verfügbare geografische Daten. Die Daten werden von Freiwilligen erfasst, indem sie Straßen, Gebäude, Gewässer, Landnutzung und andere geografische Merkmale auf der Karte markieren oder Informationen darüber hinzufügen.

Die OpenStreetMap-Daten sind unter einer offenen Lizenz, der Open Data Commons Open

Database Lizenz (ODbL), verfügbar. Das bedeutet, dass die Daten frei verwendet, kopiert, modifiziert und weiterverbreitet werden können, solange die Lizenzbedingungen eingehalten werden [4]. Dadurch wird Entwicklern ermöglicht, die OpenStreetMap-Daten in ihre eigenen Anwendungen und Dienste zu integrieren.

Literatur

[1] Hannah Bast u. a. *Route Planning in Transportation Networks*. URL: http://arxiv.org/pdf/1504.05140v1.

[2] Thomas H. Cormen u. a. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009, S. 589–592, 661–662. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[3] OpenStreetMap Foundation. *About OpenStreetMap*. 2023. URL: https://www.openstreetmap.org/about (besucht am 16. 07. 2023).

[4] OpenStreetMap Foundation. *OpenStreetMap License*. 2023. URL: https://www.openstreetmap.org/copyright (besucht am 16. 07. 2023).

[5] Peter Hart, Nils Nilsson und Bertram Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), S. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.