Gymnasium
Mariengarden

Benedikt Schwering

Facharbeit

Kürzeste-Wege-Algorithmen

in Graphen und Implementation des Dijkstra-Algorithmus in Python

Kurs: Informatik Grundkurs

Fachlehrer: Herr Stroick

Datum der Themenausgabe: 26.02.2019

Die Arbeit wurde abgegeben am 27.05.2019

Die vorliegende Arbeit erhält die Note:

Inhaltsverzeichnis

1 Einleitung

Ich habe mich dazu entschlossen, meine Facharbeit im Grundkurs Informatik über das Thema „Kürzeste Wege“ zu schreiben, da ich es sehr spannend finde, wie solche ermittelt werden und im Alltag Anwendung finden. Wäre man nicht dazu in der Lage, mithilfe von Algorithmen den kürzesten Weg zwischen Zwei Punkten bestimmen zu können, sind wesentliche Bestandteile der modernen Infrastruktur nicht oder nur bedingt möglich.
Wie würde ein Navigationssystem arbeiten, wenn es nicht die Möglichkeit gäbe, für den Autofahrer den kürzesten Weg zu seinem Ziel bestimmen zu können? Kommt der Autofahrer überhaupt an sein Ziel?
Aber nicht nur dort ist die Bestimmung des kürzesten Weges notwendig, sondern beispielsweise auch bei Datenübertragungen im Internet. Denn es ist nicht jede Route gleich schnell, da die Leitungen unterschiedliche Geschwindigkeiten unterstützen und durch andere Verwender mitbenutzt werden können.
Außerdem finde ich es sehr faszinierend, wie sich zur Zeit die Wichtigkeit dieses Algorithmus verändert und in Zukunft verändern wird. Man kann sich vorstellen, dass diese Technik bald in autonomen Fahrzeug genutzt wird. Dadurch wird schnell klar, dass die Ansprüche in Punkten der Sicherheit und Effizienz sehr hoch gesetzt sind, da sonst beispielsweise Menschenleben gefährdet werden können. Somit ist es wichtig, dass der Dijkstra-Algorithmus präzise und zuverlässig funktioniert.
Lässt sich dieses Verfahren überhaupt bei solchen Dingen nutzen, oder ist es dafür zu fehleranfällig?

2 Definitionen

2.1 Graph

Ein Graph besteht aus einer Menge von Knoten, die auch als Ecken oder im Englischen als vertices bezeichnet werden, und einer Menge von Kanten, die auch als Linien oder edges im Englischen bezeichnet werden. Jeder Kante werden genau zwei Knoten zugeordnet, wenn einer Kante zwei mal der gleiche Knoten zugeordnet wird, spricht man von einer Schleife, im Englischen auch loop genannt.
Man unterscheidet zwischen gerichteten und ungerichteten Graphen. Die Kanten erhalten in einem gerichteten Graphen, anders als in einem ungerichteten Graphen, eine Richtung. [Steffen, Eckhard]
Kanten und Knoten können auch mit Gewichten versehen werden (d.h. rationalen oder reellen Zahlen), dann spricht man von Kanten- bzw. Knotengewichten.
Besondere Graphentypen sind Multigraphen, bei denen mehrere Kanten an einem Knoten

vorkommen, und die Hypergraphen, bei denen eine Kante mehr als zwei Knoten miteinander verbindet. [Steinfeld, Thomas]

2.2 Weg

Ein Weg ist eine offenen Kantenfolge, in der alle Knoten verschieden sind. [Hirner, Helmut] Die Länge eines Weges ergibt sich durch die Addition der Kantengewichte. Der Startknoten erhält das Gewicht 0, die folgenden Knoten erhalten als Gewicht die Länge des Weges zu dem Startknoten.

2.3 Kürzester Weg

Wird ein Knoten durch eine zweite Kantenfolge erreicht und ist die Summe der Kantengewichte niedriger als das aktuelle Knotengewicht, so wird das Knotengewicht überschrieben, da ein kürzerer Weg gefunden wurde.

3 Dijkstra-Algorithmus

3.1 Pseudocode

```
def dijkstra(graph, startknoten):

    erstelle Knotensatz Q;

    for Knoten v in graph:
        distanz[v] = unendlich;
        vorgaenger[v] = undefiniert;
        fuege v in Q hinzu;

    distanz[startknoten] = 0;

    while Q:
        u = Knoten in Q mit minimaler distanz[u];
        entferne u aus Q;

        for Nachbarn v in u:
            neueDistanz = distanz[u] + laenge(u,v);

            if neueDistanz < distanz[v]:
                distanz[v] = neueDistanz;
                vorgaenger[v] = u;

return distanz, vorgaenger
```

[Saurel, Sylvain]

3.2 Erklärung

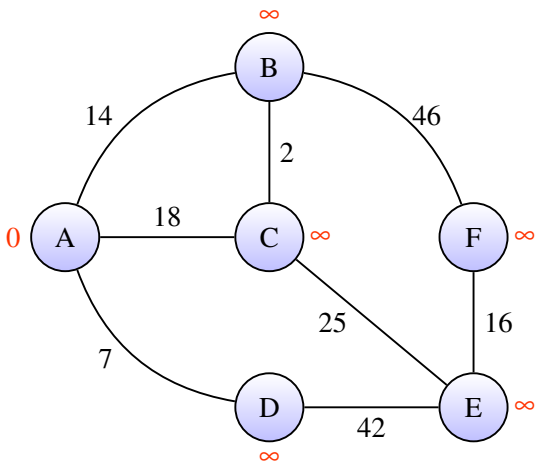
Die Idee des Dijkstra-Algorithmus besteht darin, die Länge des kürzesten Wegs zwischen Startknoten und Endknoten zu bestimmen.
Die Vorgehensweise erfolgt iterativ, das bedeutet, dass zuerst von einem sehr langen Weg, also einem schlechten Ergebnis, ausgegangen wird, welcher dann in jedem Schritt dem idealsten Weg näher kommt.
Im Folgenden erkläre ich den Dijkstra-Algorithmus anhand des Pseudocodes.
Die Funktion `Dijkstra` hat die Parameter `Graph` und `Startknoten`. Der Startknoten ist der Knoten, von dem aus der kürzeste Weg zu jedem anderen Knoten des Elements `Graph` berechnet werden soll.

Als erstes wird ein leerer `Knotensatz Q` erstellt. Dann wird jedem Knoten die Entfernung `unendlich` zugeordnet, da die kürzeste Entfernung noch nicht bekannt ist. Außerdem wird so sichergestellt, dass die Distanz `unendlich` beträgt, wenn kein einziger Weg vom Startknoten zu diesem Knoten möglich ist. Es werden auch alle Vorgänger, die später auf dem schnellsten Weg zum Startknoten führen, auf `undefiniert` gesetzt. Auch wird jeder Knoten zum `Knotensatz Q` hinzugefügt.
Darauf wird dem Startknoten die Distanz `0` zugeordnet, da vom Startknoten aus dieser sofort erreicht ist.
In der nächsten Schleife wird jeder Knoten aus `Q` betrachtet. Die Variable `u` bekommt den Knoten mit der minimalen Distanz zugeordnet. Das ist im ersten Durchlauf der Startknoten. Nun wird `u` aus `Q` entfernt, damit der aktuelle Knoten nicht noch einmal betrachtet wird.
Danach werden alle Nachbarn von `u` betrachtet und nacheinander als `v` bezeichnet. Es wird die Distanz von `u` zu `v` als Alternativwert gespeichert. Nun wird verglichen, ob `v` schon ein kleineres Gewicht hat. Ist die aktuelle Distanz kleiner, so bleibt der Wert bestehen und es wird keine Änderung vorgenommen. Ist der Alternativwert allerdings kleiner, so wird dieser die neue Distanz von `v`, da die Strecke über den Knoten `u` nun kürzer ist. In diesem Fall wird auch der Vorgänger von `v` überschrieben und gleich `u` gesetzt.
Nach dem Durchlaufen der beiden Schleifen wird das Distanz-Array und das Vorgänger-Array zurückgegeben.

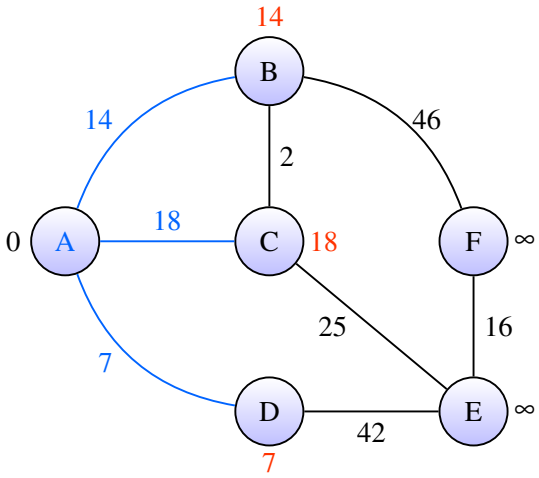
3.3 Beispiel

Beispielsweise soll der kürzeste Weg von Knoten A zu Knoten F bestimmt werden.
Änderungen am Graph werden in orange und das aktuell betrachtet Element wird in blau hinterlegt.

Zuerst werden die Knotengewichte und somit auch die Distanzen aller Knoten auf unendlich gesetzt, nur das Gewicht vom Startknoten A wird auf 0 gesetzt, da die Distanz zu sich selbst gleich null ist.

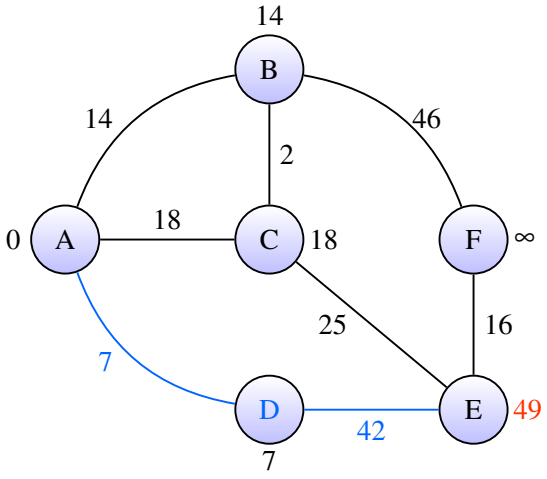


Als nächstes werden alle Knoten betrachtet, angefangen bei dem Knoten mit dem geringsten Gewicht. Hier ist das Startknoten A.
Von A werden alle Nachbarn gesucht, das sind die Knoten B, C und D. Nun wird der neue Weg zu diesen Knoten berechnet, dazu wird das Gewicht von A mit dem Kantengewicht zu den jeweiligen Nachbarn addiert. Für B ist die neue Distanz gleich 14, für C gleich 18 und für D gleich 7.

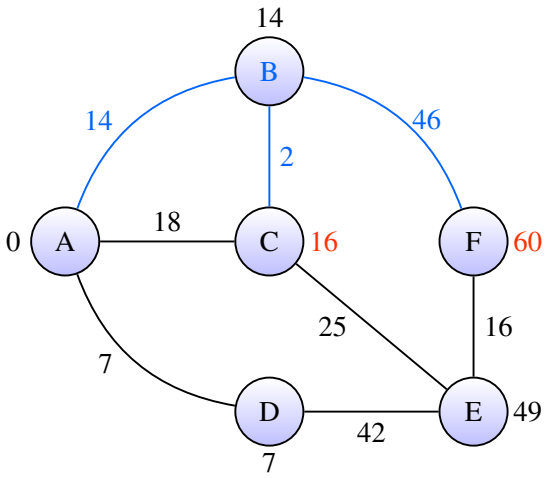


Bevor die neuen Distanzen an den Knoten geschrieben werden, muss verglichen werden, ob der neue Weg wirklich schneller ist als der vorherige. Da alle Gewichte zum Start gleich unendlich gesetzt wurden, sind die neuen Wege auf jeden Fall kürzer. Also werden die Kantengewichte überschrieben.

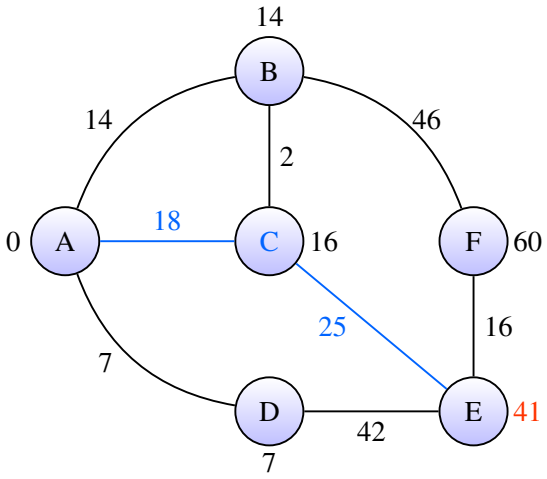
Danach wird wieder der Knoten betrachtet, der das kleinste Gewicht hat und noch nicht betrachtet wurde. Das ist nun D.
D hat die Nachbarn E und A. Die neue Distanz zu Knoten E ist 49 und da das Gewicht vorher auf unendlich gesetzt wurde, wird die 49 das neue Knotengewicht.
Es wird auch wieder der Weg von D nach A berechnet, dieser ist 7. Da 0 kleiner ist als 7 wird das Gewicht von A nicht überschrieben.



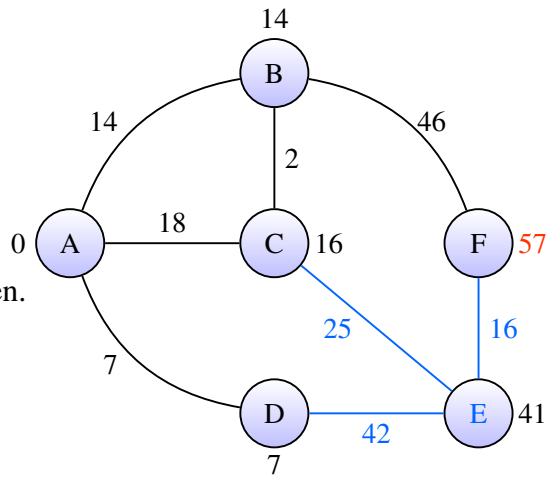
Der nächste Knoten ist B. Von B gehen drei Pfade zu A, C und F. F hat noch die Distanz unendlich, darum wird ihr das neue Gewicht 60 zugeordnet.
Der neue Weg von B zu C ist insgesamt gleich 16. Nun ist der neue Weg kleiner als der bereits vorhandene Wert, also wird die 18 von der 16 überschrieben.
Der Rückweg zu A ist natürlich länger, darum wird hier keine Änderung vorgenommen.



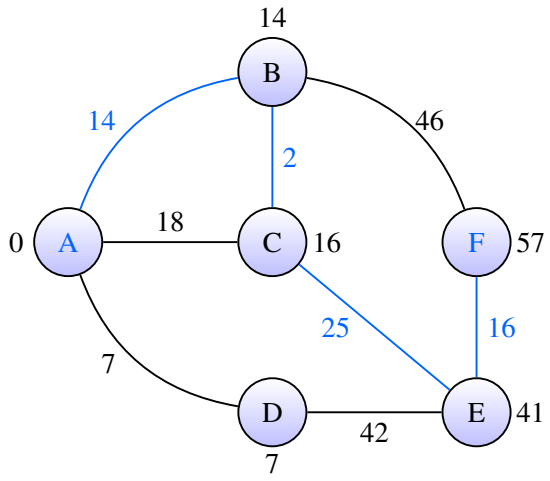
C hat die Nachbarn E und A, dessen neue Distanz wieder kleiner ist als das aktuelle Knotengewicht. Also bekommt E die neue Länge 41.
Der Rückweg von C nach A ist länger, also lassen wir ihn aus.



Der vorletzte Knoten ist E. Von E aus gehen Pfade nach D, C und F.
Die neuen Wege nach C und D sind nicht kürzer als die vorhandenen, also wird nur der Weg nach F betrachtet.
41 und 16 ist zusammen 57 und somit kleiner als 60. Die neue Distanz wird also überschrieben.



Jetzt ist nur noch F übrig. Da F der Zielknoten ist, ist es nicht nötig die Nachbarn zu betrachten. Das ist so, weil immer der Knoten mit dem kleinsten Gewicht untersucht wird. Ist nun beispielsweise ein Knoten noch nicht betrachtet worden, so ist sein Gewicht größer als der gesamte Weg zum Ziel, also kann von diesem Knoten aus das Ziel nicht mit einer kürzeren Strecke erreicht werden.



3.4 Beweis

Startknoten ist der Knoten v. Wir haben eine Menge D, in welcher die Knoten enthalten sind, für die der kürzeste Weg von v aus schon bestimmt ist.
Von einem Knoten w dessen Abstand von v $d(v,w)$ ist, können wir einen Knoten x, welcher noch nicht in der Menge D ist enthalten ist, mit den minimalen Kosten von v erreichen. Wir wählen diesen Knoten. Formalisiert gilt für alle Paare $w' \in D/W$ und $x' \in (V/D)/X$.

$$d(w,x) \leq d(w',x')$$

Würden wir nun einen Knoten $y \in V$ wählen, so würde gelten:

$$d(v,w) + d(w,x) \leq d(v,y) + d(y,x)$$

Wenn $y \in D$ ist $d(v,y) + d(y,x)$ größer als $d(v,x)$, da $d(v,x)$ schon am kleinsten ist, weil in der Menge D schon alle kürzesten Pfade bestimmt sind.
Es ist unmöglich, dass $y \notin D$, da so der Weg $d(v,y) + d(y,x)$ garantiert größer ist, da über einen anderen Knoten aus D gegangen werden muss und $d(v,x)$ minimal ist. [Tornau, Christian]

3.5 Hinweise zur Implementierung

3.5.1 Grundlegender Aufbau der Implementierung

Die Implementation umfasst zwei Methoden. Die erste Methode `dijkstra(graph, start, ende)` berechnet den kürzesten Weg von `start` nach `ende` im Graphen `graph`. Die Methode gibt den Startpunkt `start`, den Endpunkt `ende`, die Distanz zwischen Start und Ende `distanz` und eine Liste mit den vorherigen Knoten `vorgaenger`.
Die zweite Methode `ausgabe(start, ende, distanz, vorgaenger)` gibt den berechneten Wert in einfacher Form in der Konsole aus. Dazu werden neben dem Startpunkt, dem Endpunkt und der Distanz auch die einzelnen Zwischenschritte ausgegeben.
Der restliche Teil der Pythondatei besteht aus der Initialisierung des Graphens und dem Aufruf der beiden, oben beschriebenen Methoden.

3.5.2 Heap Queue

Als Datenstruktur zum Speichern der einzelnen Knoten wird eine Heap Queue, in Python `heapq` genannt, verwendet.
Diese Queue macht das Speichern des Knotennamens in Verbindung mit dem Knotengewicht leichter. Außerdem verweist das root-Element ([0]) immer auf den Eintrag mit dem geringsten Gewicht. Elementen ohne Gewicht wird automatisch das Gewicht unendlich zugeordnet. [Python-Dokumentation]
Diese Eingenschaften eignen sich perfekt für Implementationen mit binären Bäumen und Graphen.
Konkret sieht die Heap Queue im 3. Schritt des Beispiels wie folgt aus:

```
     [7, [D, [A, 7], [E, 42]]]
[14, [B, [A, 14], [C, 2], [F, 46]]]
[18, [C, [A, 18], [B, 2], [E, 49]]]
    [49, [E, [D, 42], [F, 16]]]
    [∞, [F, [B, 46], [E, 16]]]
```

Es ist zu erkennen, dass als nächstes Element der Knoten `D` mit dem kleinsten Gewicht von `7` aufgerufen wird. Die anderen Elemente sind nach ihrem Gewicht aufsteigend geordnet. Am Ende des Heaps steht der Knoten `F` mit dem größten Knotengewicht ∞.

3.5.3 Bedienung

Zum Starten die Datei `/code/dijkstra.py` mit dem Befehl `python dijkstra.py` aufrufen.
Der Graph kann im Code angepasst werden. Beispielsweise bedeutet `'A': [['B', 14], ['C', 18], ['D', 7]]`, dass der Knoten A die Nachbarn B mit der Länge 14, den Nachbarn C mit der Länge 18 und den Nachbarn D mit der Länge 7 hat.
Möchte man einen anderen Weg berechnen, als der standardmäßige Weg von `A` nach `F`, so muss man die Parameter im Aufruf der `dijkstra(G, 'A', 'F')` Methode ändern.

4 Bellman-Ford-Algorithmus

4.1 Pseudocode

```
def bellmandord(graph, startknoten):

    for Knoten v in graph:
        distanz[v] = unendlich;
        vorgaenger[v] = undefiniert;

    distanz[startknoten] = 0;

    for Knoten n - 1:
        for benachbarte Knoten u, v in Kanten:
            if (distanz[u] + gewicht[u, v]) < distanz[v]:
                distanz[v] = distanz[u] + gewicht [u, v]
                vorgaenger[v] = u

    for Knoten u, v in Kanten:
        if (distanz[u] + gewicht[u, v]) < distanz[v]:
            STOPP: "Negativen Zyklus gefunden!"

return distanz, vorgaenger
```

[Chumbley, Alex; Moore, Karleigh]

4.2 Erklärung

Der Bellman-Ford-Algorithmus verfolgt, wie der Dijkstra-Algorithmus, die Bestimmung des kürzesten Weges.
Außerdem geht er auch iterativ vor, er geht also von der unidealsten Lösung aus und verbessert seine Berechnung in jedem Durchlauf.
Anders als bei der Erklärung von Dijkstra, beschreibe ich den Bellman-Ford-Algorithmus ohne auf Einzelheiten des Pseudocodes einzugehen.
Der Ausgangszustand ist, wie auch bei Dijkstra, die schlechteste Prognose des kürzesten Wegs, also bekommen alle Knoten den Wert unendlich, nur der Startknoten erhält den Wert 0, da man, von ihm selbst aus gesehen, ihn natürlich mit keinem Aufwand erreicht.
Als nächstes werden alle Kanten, in willkürlicher Reihenfolge betrachtet.
Für jede Kante wird nun überprüft, ob das Gewicht des Anfangsknoten addiert mit dem Kantengewicht niedriger als das Gewicht des Zielknotens ist. Wenn dies eintrifft, wurde eine Abkürzung gefunden, also wird das Gewicht des Zielknotens aktualisiert.

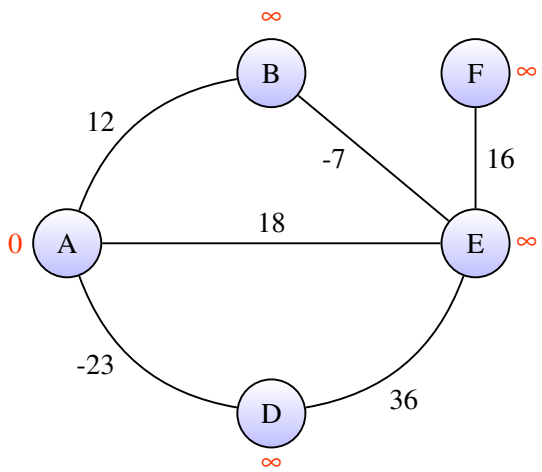
Diesen Durchlauf durch alle Kanten bezeichnet man als Phase.
Es reicht allerdings nicht immer nur ein Durchlauf aus, um alle kürzesten Wege zu bestimmen. Nun haben wir lediglich den kürzesten Abstand zwischen zwei Knoten bestimmt, die nur durch eine Kante miteinander verbunden werden.
Jetzt stellt sich aber die Frage, wie viele Phasen nötig sind, um von jedem Knoten aus den kürzesten Weg zu jedem anderen Knoten zu wissen. Logisch nachgedacht kommt man auf die Lösung, dass der längste, relativ gesehen möglichst kurze, Weg auf jeden Fall weniger Kanten haben muss als Knoten im kompletten Graph. Also brauchen wir genau eine Phase weniger als Knoten im gesamten System vorhanden sind.

Aber wie wird nun der kürzeste Weg konkret konstruiert?
Dazu speichert der Algorithmus für jeden aktualisierten Knoten die Kante, mit der die Abkürzung gefunden wurde.
Am Ende kann man also so lange den Vorgängerknoten bestimmen, bis man am Startknoten angekommen ist. [Stolz, Richard]

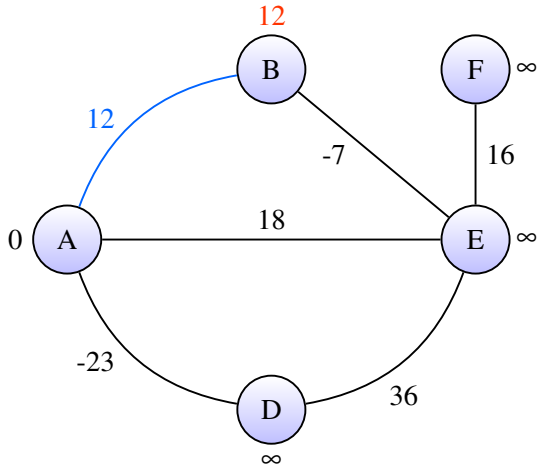
4.3 Beispiel

In diesem Beispiel soll der kürzeste Weg zwischen den Knoten A und F bestimmt werden. Änderungen am Graph werden in orange und die aktuell betrachtete Kante wird in blau kenntlich gemacht.
Ich werde zunächst eine komplette Phase ausführlich beschreiben. In den darauf folgenden Schritten gehe ich dann nur noch auf die Änderung durch die nächsten Phasen ein.

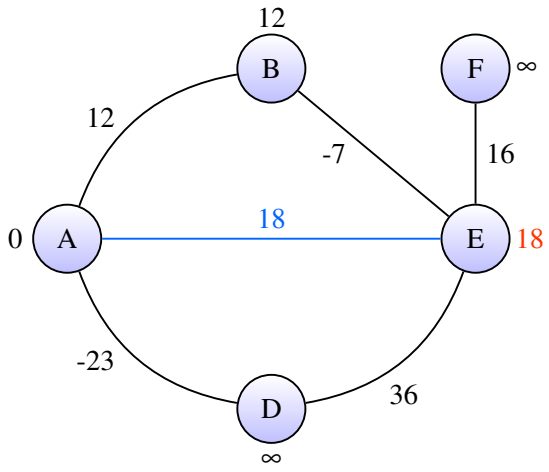
Bevor die erste Phase beginnt werden die Kosten aller Knoten auf unendlich gesetzt. Es wird also vom schlechtesten Fall ausgegangen. Allerdings ist die Entfernung vom Startknoten A zu sich selbst natürlich null.



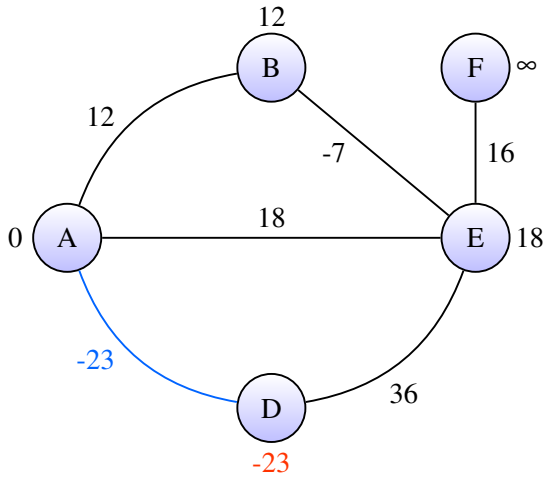
Die erste Phase betrachtet, wie jede andere Phase auch, jede Kante und vergleicht, ob durch diese Kante eine der beiden Endpunkte schneller erreicht werden könne.
In diesem Fall wird die Kante zwischen A und B betrachtet.
Das Gewicht von A ist 0 und 0 plus das Gewicht der Kante ist zusammen 12. Der berechnete Weg zu B ist also 12. Es muss noch verglichen werden, ob der Weg mit der Länge 12 wirklich eine Abkürzung ist. Da 12 kleiner als unendlich ist, wird das Gewicht von B mit 12 aktualisiert.



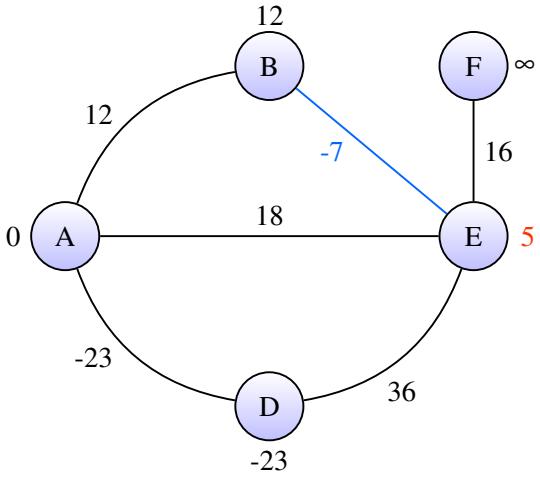
Als Nächstes wird die Kante zwischen A und E betrachtet. Sie hat das Gewicht 18 und durch sie kann der Knoten E mit dem Aufwand von 18 erreicht werden. Dieser neue Weg ist natürlich kürzer als ein unbekannter Weg mit der Länge von unendlich, somit wird das Gewicht von E aktualisiert.



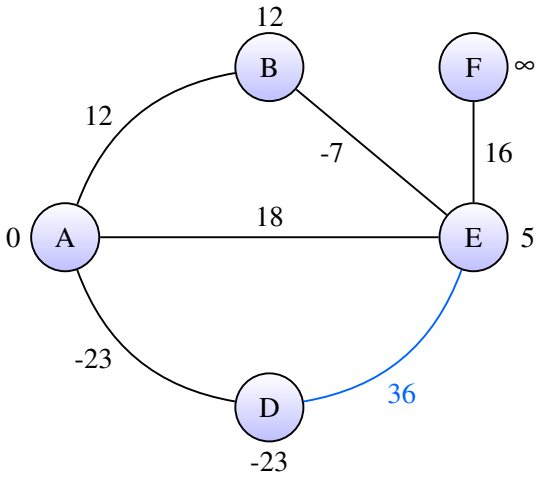
Die nächste Kante verbindet A und D. Addiert ergeben 0 und -23 in Summe -23. Diese neue Länge ist wiederum kleiner als unendlich und deswegen bekommt D das neue Knotengewicht -23.



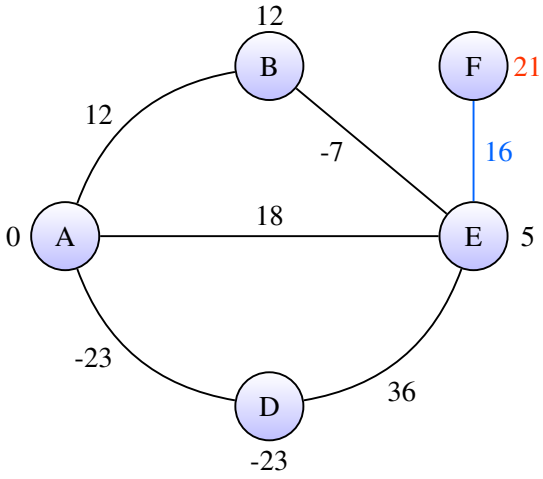
Nun ist die Strecke zwischen B und E an der Reihe. Die neue Gewichtung des Wegs lautet 5, da 12 plus -7 gleich 5 ist. Diese neue Länge ist kleiner als 18, somit wurde eine Abkürzung gefunden.

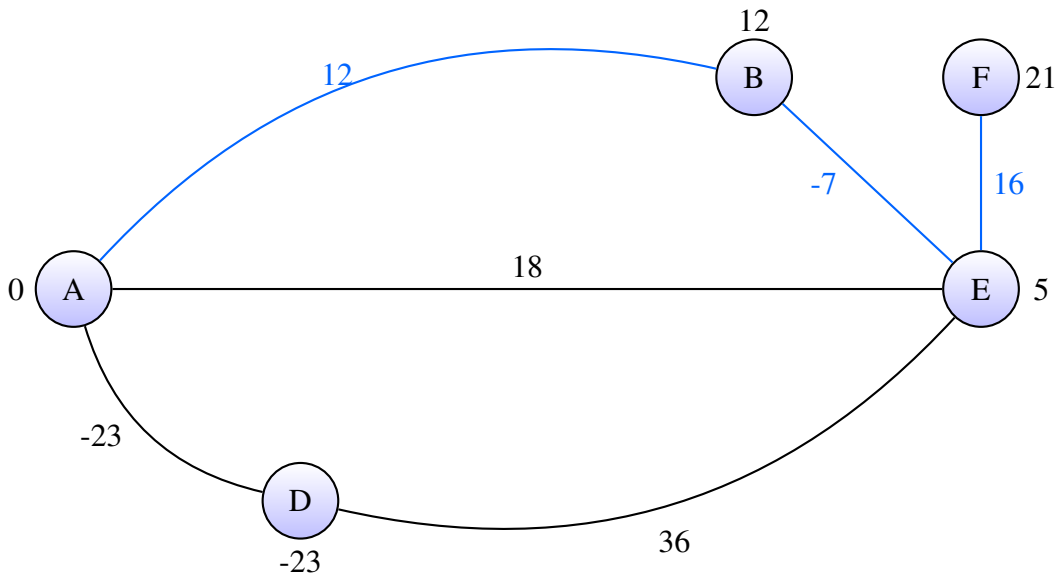


Die Strecke D zu E hat die Länge 36, addiert ergibt sich eine neue Länge von 13. Dieser neue Weg ist allerdings keine Abkürzung zu E, da E schon mit einem Aufwand von 5 erreicht werden kann. Das bedeutet, es findet keine Aktualisierung statt.



Die letzte Kante verbindet E und F miteinander. F hat noch das Gewicht unendlich, somit ist jeder neu berechnete Weg kürzer. Der neue Weg hat die Länge 21 und somit ist dies auch das neue Gewicht von F.





Prinzipiell wird dieser Phasenvorgang noch 3 mal wiederholt, da das System 5 Knoten hat. Aber in diesem Beispiel wurde nun schon für jeden Knoten der kürzeste Weg berechnet.

4.4 Hinweise zur Implementation

4.4.1 Grundlegender Aufbau der Implementierung

Die Implementation umfasst zwei Methoden. Angefangen mit der Hauptmethode `bellmanford (graph, startknoten)`, die zu allen Knoten im Graph die kürzeste Distanz errechnet. Außerdem werden auch die jeweiligen Vorgänger auf dem direkten Weg zum Startknoten mitberechnet.
Die Eingabe des Graphen erfolgt über eine Liste von Kanten. Das macht es leichter in der Programmierung, da jede Kante und nicht jeder Knoten betrachtet wird.
Die zweite Methode `ausgabe(start, ende, distanz, vorgaenger)` ist von der Implementation des Dijkstra-Algorithmus übernommen worden. Diese Methode gibt das Ergebnis in der Konsole aus.

4.4.2 Bedienung

Zum Starten die Datei `/code/bellmanford.py` mit dem Befehl `python bellmanford.py` aufrufen.
Der Graph kann im Code angepasst werden. Beispielsweise bedeutet `{'start': 'A', 'ziel': 'B', 'gewicht': 12}` , dass zwischen den Knoten A und B eine Kante mit der Länge 12 ist. Möchte man einen anderen Weg berechnen, als der standardmäßige Weg von `A` nach `F`, so muss man die beiden Werte der Variablen `startknoten = 'A'` und `endknoten = 'F'` ändern.

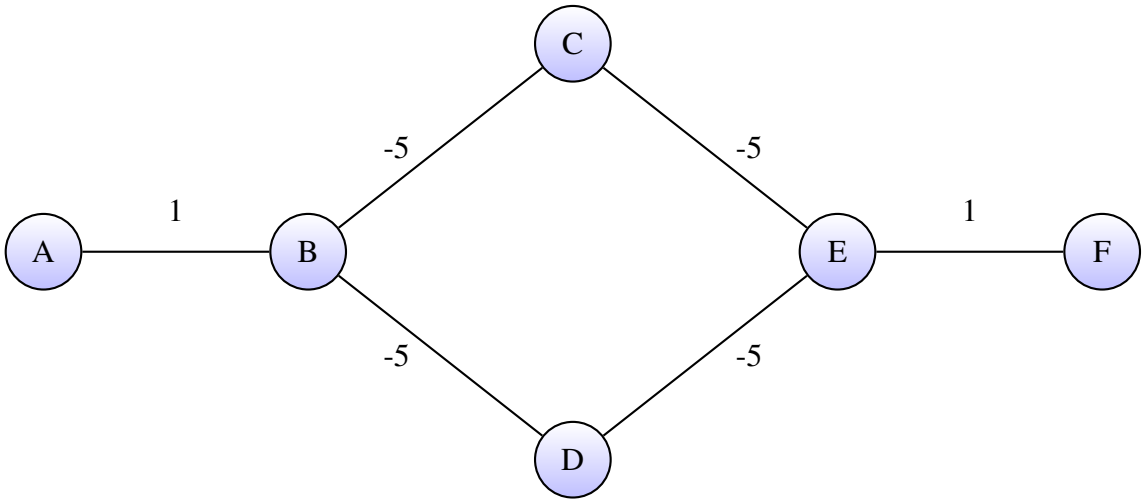
4.5 Vergleich zu Dijkstra

4.5.1 Vorteile

Im Gegensatz zum Dijkstra-Algorithmus kann der Bellman-Ford-Algorithmus auch negative Kantengewicht verarbeiten.
Dies klingt erst einmal nicht spektakulär und findet auf den ersten Blick auch keinen Platz in einer realen Anwendung.
Die Praktikabilität lässt sich am besten an einem Beispiel erläutern:
Fährt ein Taxifahrer beispielsweise zu einem Fahrgast, so entstehen für ihn Kosten, also ist das Gewicht für diesen Weg positiv. Hat er aber nun einen Fahrgast, so entstehen für ihn keine Ausgaben, ganz im Gegenteil, er verdient sogar an seinem Fahrgast. Somit ist das Kantengewicht für diese Strecke negativ. In Summe kann er also entweder Gewinn oder Verlust machen. [Stolz, Richard]

4.5.2 Nachteile

Was zugleich Vorteil ist, kann auch zum Nachteil werden. Denn nicht alle Graphen mit negativen Kantengewichten kann der Bellman-Ford-Algorithmus lösen.
Dazu ein Beispiel:



Dieses Beispiel stellt einen sogenannten negativen Kreis dar.
Betrachtet der Algorithmus nun eine der vier Kanten mit dem Gewicht `-5`, so wird er immer zu dem Nachbarn einen Weg, verkürzt um `5`, erkennen. Somit ist er quasi in einer Endlosschleife.
Für diese Fehleranfälligkeit wurde aber eine Lösung gefunden.

Wie oben festgestellt, braucht der Algorithmus genau eine Phase weniger als Knoten im System sind. Am Ende aller Durchläufe wird nun noch einmal jede Kante geprüft und gibt es mindestens eine, die zu einem Nachbarn eine Abkürzung herstellen kann, so erkennt der Algorithmus, dass es sich um einen negativen Kreis handelt. [Stolz, Richard]

5 Ausblick

5.1 Navigationssysteme

5.1.1 GPS

GPS steht im Allgemeinen für ”Global Positioning System”und heißt auf Deutsch so viel wie ”Globaler Positions Dienst”.
Der GPS-Dienst wurde offiziell am 17.06.1995 in Betrieb genommen und war ursprünglich für den Einsatz im US-amerikanischen Militär bestimmt.
Das System arbeitet mit Satelliten, die ein permanentes Signal mit ihrem aktuellen Standort und mit der aktuellen Uhrzeit senden. Dieses Signal wird von GPS-Empfängern empfangen und, da dank der Relativitätstheorie von Albert Einstein, bekannt ist, dass sich Licht, und somit auch andere elektromagnetische Wellen, mit einer Geschwindigkeit von $299.792.458\frac{m}{s}$ [Meßinger-Koppelt, Dr. Jenny] ausbreiten, somit kann der Standort auf bis zu einige Meter genau bestimmt werden.
Obwohl GPS mit modernen Satelliten arbeitet, die rund um die Erde fliegen, gibt es auch einige Schwächen. Beispielsweise sollte man sich nicht darauf verlassen, dass man mit GPS jederzeit an jedem Ort seinen Standort bestimmen kann. Anders als bei Handystrahlen gelangen GPS-Strahlen nicht durch dickere Materieschichten. Die perfekte Voraussetzung für einwandfreien GPS-Empfang ist eine ”freie Himmelssicht”. [Irle, Matthias]

5.1.2 Allgemeine Routenbestimmung

Allgemein unterscheiden moderne Navigationsgeräte zwischen der schnellsten und der kürzesten Route.
Dazu ein Beispiel: Eine Strecke von A nach B ist beispielsweise am kürzesten, wenn am viele Abkürzungen durch Feldwege und Bauernschaften nimmt. Allerdings ist der etwas längere Weg über die Autobahn trotzdem schneller.
Um dieses unnötige Befahren von langsamen Wegen zu vermeiden wurde die Berechnung des schnellsten Weges eingeführt.
Implementieren lässt sich diese Art der Route, indem man jedes Kantengewicht mit der auf dieser Straße geltenden zulässigen Höchstgeschwindigkeit multipliziert.

5.1.3 Cloudbasierte Verkehrserkennung und Echtzeitdaten

Jeder Nutzer von Google Maps stellt im Laufe der Zeit fest, dass im Gegensatz zu anderen Navigationssystemen, die berechnete Zeit, für zum Beispiel das Durchfahren eines Staus, ziemlich genau ist. Aber warum kann Google so genaue Prognosen aufstellen?
Das Geheimnis hinter dieser Technik ist die große Menge an Android-User. Denn jedes Handy, was das mobile Betriebssystem von Google installiert hat, sendet in regelmäßigen

Abständen seine Standortdaten an Google. Mit diesen Daten errechnet Google dann, wie viele Autos auf den jeweiligen Straßen sind und vor allem, wie schnell gerade gefahren werden kann.
Aber warum werde ich denn dann von Google gewarnt, wenn noch kein Stau vorhanden ist? Das liegt daran, dass Google die gesammelten Daten nicht nur in Echtzeit verarbeitet, sondern auch speichert und analysiert.
Die Algorithmen von Google Maps erkennen wiederkehrende Verkehrsereignisse, wie zum Beispiel Stau, der täglich durch den Berufsverkehr ausgelöst wird. So kann Google relativ genaue Stauprognosen erstellen, bevor der Stau sich gebildet hat.
Diese Technik bringt natürlich auch einen großen Nachteil mit sich. Denn Google weiß von jedem Gerät, wann es sich wo auf der Welt aufgehalten hat. Für Google eröffnen sich dadurch zwar neue Möglichkeiten, personalisierte Werbung zu schalten, aber auf der anderen Seite fühlen sich die Nutzer dadurch in ihrer Privatsphäre beschränkt.
Zur Privatsphäre äußert sich Google, in dem sie sagen, dass alle Daten vertraulich und anonymisiert verarbeitet werden. Ob man sich darauf verlassen kann, möchte ich an dieser Stelle stark anzweifeln. [Landesanstalt für Medien NRW]

5.1.4 Datenspeicherung

Nachdem ich einen kurzen Ausblick auf die Standort- und Routensysteme gegeben habe, möchte ich noch kurz darauf eingehen, wie die Kartendaten konkret auf den Navigationsgeräten gespeichert werden.
Eine Speicherung der einzelnen Knoten und Nachbarn, wie in meiner Implementation würde sehr schnell unübersichtlich werden. Um das Speichern übersichtlicher zu gestalten, werden verschiedene Formate verwendet.
Der Zugriff auf die Kartenmaterialien erfolgt via Datenbanken. Um den Zugriff zu vereinheitlichen, wird eine Abstraktionsschicht, im englischen ”DBAL - data base abstraction layer”genannt, eingeführt. [Kluge, Sebastian]
Übliche Dateiformate sind WMS (Web Map Service), WFS (Web Feature Service) und WMTS (Web Map Tile Service). [Open Data]
Durch diese Formate ist es möglich, nicht nur einzelnen Positionen mit einer Adresse zu verknüpfen, sondern auch andere, alltagsnahe Daten, wie zum Beispiel komplexe Straßennetzwerke, besondere Verkehrsführungen, Kartendarstellungen und besondere Orte (POI - Point Of Interest), zu speichern. [Kluge, Sebastian]

6 Fazit

Die Entwicklung der Algorithmen zur Bestimmung des kürzesten Weges zeigt, dass man heute nun sehr präzise bestimmen kann, welcher Weg der Kürzeste ist. Diese Tatsache bringt

eine hohe Effizienz hervor, welche in unserem Alltag einen besonderen Stellenwert hat, da man mit Zeiteinsparungen bei beispielsweise einem Transport einer Ware viele verschiedene Waren einem Zeitfenster transportieren kann, da jede einzelne Fahrt weniger lang dauert. Mit dieser Methodik lassen sich Verkehrslagen und anderweitige Besonderheiten genauestens bestimmen, was dafür sorgt, dass sie ein großes Zukunftspotenzial besitzt und daher auch für eine autonome Fahrt, wie ich sie in der Einleitung beschrieben habe, nutzbar gemacht werden kann. Dieses Potenzial kann den Alltag in der nahen Zukunft noch weiter vereinfachen und effizienter machen und so für die gesamte Bevölkerung ein smarteres Leben hervorrufen.
An dieser Stelle möchte ich noch erwähnen, dass ich die Facharbeit mithilfe von LaTeXverfasst habe. Es ist erstaunlich einfach, Graphen mit TikZ zu erstellen. Außerdem bin ich für weitere Funktionen, wie zum Beispiel den schnellen Zugriff auf die monospaced Font und die Möglichkeit, auf das Literaturverzeichnis zu verweisen, sehr dankbar.
Des weiteren habe ich die Programmiersprache Python verwendet, da man mit ihr Algorithmen übersichtlich und vor allem in kurzer Form umsetzen kann. Ich finde die große Anzahl an ”Built-In”Funktionen praktisch, da man sich somit mehr auf die eigentlichen Algorithmen konzentrieren kann, ohne sich große Gedanken über die Umsetzung machen zu müssen.

7 Quellen-/Literaturverzeichnis

Literatur

[Steffen, Eckhard] Graphentheorie
http://www.enzyklopaedie-der-wirtschaftsinformatik
de/lexikon/technologien-methoden/
Informatik--Grundlagen/Graphentheorie
Stand: 26.09.2014

[Steinfeld, Thomas] Graphentheorie
https://mathepedia.de/Graphentheorie.
html
Stand: 26.04.2019

[Hirner, Helmut] 7 Kantenfolge, Kantenzug, Weg
http://notizblog.digital/2012/10/
7-kantenfolge-kantenzug-weg/
Stand: 12.10.2012

[Saurel, Sylvain] Calculate shortest paths in Java by implementing Dijkstra's Algorithm
https://medium.com/@ssaurel/
calculate-shortest-paths-in-java-by-implementing-d
Stand: 03.06.2016

[Tornau, Christian] Dijkstra-Beweis
http://www.informatikseite.de/
algorithmen/node7.php
Stand: 29.04.2019

[Python-Dokumentation] Heap Queue
https://docs.python.org/2/library/heapq.
html
Stand: 12.05.2019

[Meßinger-Koppelt, Dr. Jenny] Lichtgeschwindigkeit
https://www.leifiphysik.de/optik/
lichtausbreitung/lichtgeschwindigkeit
Stand: 22.05.2019

[Irle, Matthias] Grundlagen der GPS-Technik
http://www.naviaktiv.de/index.php?cont=gpstechnik
Stand: 22.05.2019

[Landesanstalt für Medien NRW] Geheimnis Staumelder: Wie funktioniert eigentlich Google Maps?
https://www.handysektor.de/artikel/geheimnis-staumelder-wie-funktioniert-eigentlich-g
Stand: 23.05.2019

[Stolz, Richard] Der Bellman-Ford-Algorithmus
Lehrstuhl M9 der Technischen Universität München
2013
https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_de.html
Stand: 24.05.2019

[Kluge, Sebastian] Komponenten moderner Navigationssysteme - Software
https://www-m6.ma.tum.de/foswiki/pub/M6/Lehrstuhl/SebastianKluge/Moderne_Navigationssysteme.pdf
Seite 36
Stand: 13.04.2010

[Open Data] Geodienste - „Open Data“ - Rahmenkonzept
http://notes.leipzig.de/appl/laura/wp5/kais02.nsf/docid/B9F0508BE42227B7C1257C7F002F23F9/$FILE/V-ds-3615-anlage-1.pdf
Seite 18, 19
Stand: 15.04.2014

[Chumbley, Alex; Moore, Karleigh] Bellman-Ford Algorithm
https://brilliant.org/wiki/bellman-ford-algorithm/

8 Selbstständigkeitserklärung

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Borken, den 26.05.2019