Dynamische und verteilte Abwandlungen des Floyd-Warshall Algorithmus

Seminar: Verkehr und Logistik

Inhalt

Einleitung

In dieser Arbeit soll es um den Floyd-Warshall Algorithmus und mögliche dynamische Abwandlungen davon gehen. Die Arbeit ist in zwei Teile gegliedert. Der erste Teil beschäftigt sich mit dem Floyd-Warshall Algorithmus in seiner reinen Form sowie einer Analyse dieses Algorithmus.

Der zweite Teil beschäftigt sich mit innovativen Verfahren zur Dynamisierung.

Floyd-Warshall Algorithmus

Das Grundproblem

Das Grundproblem welches hier behandelt werden soll beschäftigt sich mit der Berechnung von kürzesten Wegen W in einem Graphen.

Dieses Grundproblem kann in 3 Subprobleme unterteilt werden[1]:

1. Kürzeste Wege zwischen zwei Ecken
2. Kürzeste Wege zwischen einer Ecke und allen anderen Ecken
3. Kürzeste Wege zwischen allen Paaren von Ecken

Der Floyd-Warshall Algorithmus, der später noch genauer beschrieben wird, löst das Problem 3.

Der Algorithmus

Zwei kurze Sätze zu den beiden Personen nach denen der Algorithmus benannt wurde. Robert W. Floyd, 1936 – 2001, war ein Informatiker, der unter anderem mit einem Paper über Programmverifikation, „Assigning Meanings to Programs“, einen wichtigen Beitrag zur später entwickelten Hoare Logik beitrug.

Stephen Warshall, 1935 – 2006, wurde bekannt, weil er die Korrektheit des Algorithmus zum Berechnen des transitiven Abschlusses eines Graphen beweisen konnte.

Bevor der eigentliche Algorithmus beschrieben wird, soll das Grundprinzip erläutert werden, welches dem Algorithmus zugrunde liegt. Es ist das Prinzip der dynamischen Programmierung.

Dynamisches Programmieren[2] hatte in seiner ursprünglichen Form nicht mit der Generierung von Computerprogrammen zu tun, sondern beschrieb eine Methode, mathematische Optimierungsprobleme zu lösen. Da heutzutage fast alle Probleme durch ein entsprechendes Computerprogramm gelöst werden, ist die dynamische Programmierung eng mit der Implementierung von Algorithmen verbunden.

Mithilfe der dynamischen Programmierung lassen sich Probleme lösen, die rekursiv beschrieben werden können. Die rekursive Beschreibung eines Problems führt dazu, dass es in kleinere Subprobleme aufgeteilt werden kann. Diese können anschließend einzeln für sich gelöst werden.

---

1 Turau, Volker: Algorithmische Graphentheorie, 2. 2004, S. 243

2 http://ddi.cs.uni-potsdam.de/Personen/marco/DynProgrammieren.pdf
http://wwwcs.upb.de/cs/ag-monien/LEHRE/SS06/DuA/22.pdf
http://www.algorithmist.com/index.php/Dynamic_Programming

Die optimale Lösung ergibt sich dann aus der optimalen Lösungen aller Subprobleme.

Zwei Varianten führen zu einer Lösung:

1. Top-Down-Lösungsweg:

   Es werden alle Subprobleme gelöst und jedes Ergebnis wird in einer Tabelle abgespeichert. Bevor ein Subprolem gelöst wird, wird in dieser Tabelle geprüft, ob dieses Ergebnis möglicherweise schon vorher berechnet wurde.

2. Bottom-Up-Lösungsweg:

   Es werden die kleinsten Subprobleme gelöst. Die Ergebnisse werden nun genutzt um, die nächstgrößeren Subprobleme zu lösen.

Eine weitere Anwendung dieser Methode ist das Erstellen einer Fibonacci-Reihe.

Nachdem nun das Grundprinzip des Algorithmus dargestellt worden ist, soll im folgenden Teil der Algorithmus selbst beschrieben werden.

Zunächst sind noch einige wichtige Grundbestandteile für den Algorithmus festzulegen.

- Es wird ein Graph G benötigt. Dieser Graph G , darf keine negativen Kreise aufweisen.
- Weiter wird eine Distanzmatrix D_j benötigt. Eine Adjazenzmatrix lässt sich in eine Distanzmatrix überführen, indem einfach die Kantenbewertungen in die Tabelle eingetragen werden.
- Eine Vorgängermatrix A_j . Eine Vorgängermatrix speichert in der *i*-ten Zeile das Vorgängerfeld für einen Pfad mit der Startecke *i*.

Auf welche Weise löst der Floyd-Warshall Algorithmus nun das Problem der kürzesten Wege zwischen allen Paaren von Ecken in einem Graphen?
Es soll der kürzester Weg W von $i \in V$ nach $k \in V$ nur mit Ecken aus *{1,...,j}* $j \in V$ berechnet werden.

Hierbei sind nun zwei Fälle zu unterscheiden:

1. Der Weg W verwendet nicht die Ecke *j*.
2. Der Weg W verwendet die Ecke *j*.

Für Fall 1 würde ein Weg W folgendermaßen aussehen:

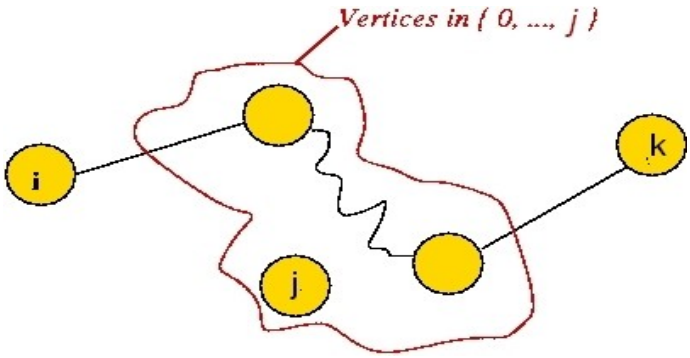

Abbildung 1: Alle Knoten ausser dem Knoten j

Da der Weg W nicht die Ecke j benutzt, muss im folgenden der Weg zwischen i und k nur noch aus Ecken aus der Menge *{ 1,..., j-1 }* bestehen. Die Ecke j wird bei der weiteren Wegberechnung nicht mehr berücksichtigt.

Fall 2 ist der interessantere der beiden und würde folgendermaßen aussehen:

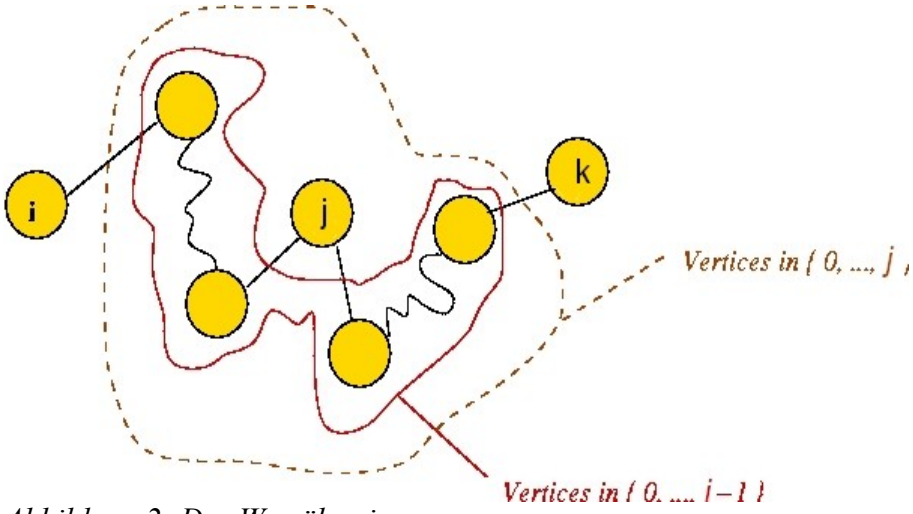

Abbildung 2: Der Weg über j.

Es sind jetzt zwei neue Wege entstanden. Der erste Weg W_1 ist der Weg von i nach j. Der zweite Weg W_2, ist der Weg von j nach k. Nun kann jeder einzelne Weg für sich selbst optimiert werden.

Beide Wege dürfen deshalb voneinander getrennt optimiert werden, da der Graph keine negativen Kreise besitzt und damit das Optimalitätsprinzip[3] Wirkung zeigt. Das Optimalitätsprinzip sagt aus: Wenn es einen kürzesten Weg von i nach j gibt und es einen kürzesten Weg von j nach k gibt, ist der Weg von i nach k über j auch der kürzeste Weg.

Im folgenden Abschnitt soll der konkrete Algorithmus erläutert werden. Zuvor wurde das Prinzip beschrieben, auf welche Weise der Algorithmus arbeitet. Um festzustellen, ob der Weg W von i nach k über j kürzer ist als der direkte von i nach k, muss folgende Bedingung überprüft werden:

$$D[i,k] = min(D[i,k], D[i,j] + D[j,k])$$

Es wird an dieser Stelle sowohl geprüft als auch zugewiesen. In der Distanzmatrix an der Stelle i,k soll nun das Minimum des Weges von i nach k direkt und von i nach k über j stehen.
Nachdem das Prinzip und die zu überprüfende Bedingung erklärt ist, wird nun der eigentliche Algorithmus in Pseudo-Code-Notation angegeben:

```
for j := 1 to n do
    for i := 1 to n do
        for k := 1 to n do
        if D[i,k]  > D[i,j] + D[j,k] then
            D[i,k] := D[i,j] + D[j,k]
```

Wie nun zu erkennen ist, wird für alle j einmal geprüft, ob dieses j auf dem Weg zwischen allen i und k liegt. Da noch die Vorgängermatrix zur Verfügung steht, kann durch eine kleine Veränderung auch gleich der Weg eingetragen werden:

```
for j := 1 to n do
    for i := 1 to n do
    for k := 1 to n do
        if D[i,k]  > D[i,j] + D[j,k] then
            D[i,k] := D[i,j] + D[j,k]
            V[i,k] := V[j,k]
```

An der Stelle i,k in der Vorgängermatrix wird einfach der Startknoten für den Weg von j nach k eingetragen. Es heißt: Um von i nach k zu kommen, muss der Knoten j passiert werden.

3 Turau, Volker: Algorithmische Graphentheorie, 2. 2004, S. 244 f.

Das Beispiel

Im folgenden Abschnitt soll die Arbeitsweise des Floyd-Warshall Algorithmus anhand eines Beispiels gezeigt werden.[4]

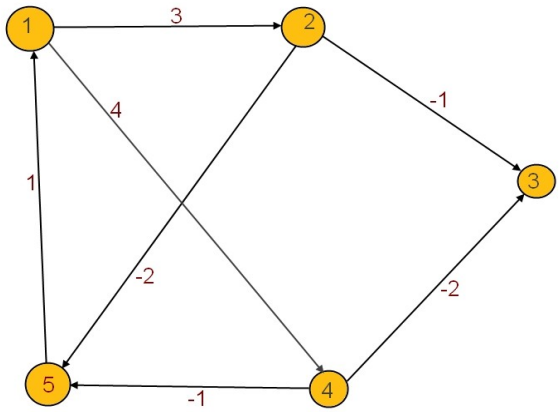

Abbildung 3: Beispielgraph

Aus dem Beispielgraphen aus Abbildung 3 folgen die initialisierte Distanzmatrix und die initialisierte Vorgängermatrix. Die Distanzmatrix entspricht nach der Initialisierung der Adjazenzmatrix. In der Vorgängermatrix sind nach der Initialisierung nur Wege zu direkten Nachbarn eingetragen.

$$D_0 = \begin{pmatrix} 0 & 3 & \infty & 4 & \infty \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & \infty & \infty & \infty & 0 \end{pmatrix}$$

Tabelle 1: Initialisierte Distanzmatrix

$$V_0 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Tabelle 2: Initialisierte Vorgängermatrix

Nach dem ersten Schritt des Algorithmus sind einige Veränderungen eingetragen worden. Diese sind Fett markiert.

„Nach dem ersten Schritt“ bedeutet, dass der Knoten 1 als j gewählt wurde und dann für alle Paare von Ecken geprüft wird, ob der Weg über j kürzer ist, als der direkte schon bekannte Weg von i nach k.

4 Turau, Volker: Algorithmische Graphentheorie, 2. 2004, S. 279

$$D_1 = \begin{pmatrix} 0 & 3 & \infty & 4 & \infty \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & \mathbf{4} & \infty & \mathbf{5} & 0 \end{pmatrix}$$

Tabelle 3: Distanzmatrix nach Schritt 1

$$V_1 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & \mathbf{1} & 0 & \mathbf{1} & 0 \end{pmatrix}$$

Tabelle 4: Vorgängermatrix nach Schritt 1

Nach dem zweiten Schritt sind wieder einige Veränderungen eingetreten. Diese Veränderungen sind folgendermaßen zu deuten:

Der Algorithmus hat festgestellt, dass der Weg über den Knoten 2 kürzer ist als der direkte Weg von i nach k. Z.B wurde festgestellt, dass der Weg vom Knoten 1 zum Knoten 3 über den Knoten 2 führt.

Es zeigt sich nun, dass dieser Weg 2 Wegeinheiten benötigt. Gleichzeitig wurde in die Vorgängermatrix folgendes eingetragen: um von dem Knoten 1 zum Knoten 3 zu gelangen, muss man den Knoten 2 passieren, da der Knoten 2 der Startknoten für den kürzesten Weg vom Knoten 1 zum Knoten 3 ist, ausgehend vom Knoten 1.

$$D_2 = \begin{pmatrix} 0 & 3 & \mathbf{2} & 4 & \mathbf{1} \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & 4 & \mathbf{3} & 5 & 0 \end{pmatrix}$$

Tabelle 5: Distanzmatrix nach Schritt 2

$$V_2 = \begin{pmatrix} 0 & 1 & \mathbf{2} & 1 & \mathbf{2} \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 1 & \mathbf{2} & 1 & 0 \end{pmatrix}$$

Tabelle 6: Vorgängermatrix nach Schritt 2

In den Schritten 3 und 4 werden keine weiteren Veränderungen festgestellt. Im Schritt 3 deswegen nicht, da es keine ausgehende Kante vom Knoten 3 gibt. In Schritt 4 wird nichts verändert, weil in Schritt 1 und 2 schon ein kürzester Weg gefunden worden ist.

$$D_3 = \begin{pmatrix} 0 & 3 & 2 & 4 & 1 \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & 4 & 3 & 5 & 0 \end{pmatrix}$$

Tabelle 7: Distanzmatrix nach Schritt 3

$$V_3 = \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 1 & 2 & 1 & 0 \end{pmatrix}$$

Tabelle 8: Vorgängermatrix nach Schritt 4

$$D_4 = \begin{pmatrix} 0 & 3 & 2 & 4 & 1 \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & 4 & 3 & 5 & 0 \end{pmatrix}$$

Tabelle 9: Distanzmatrix nach Schritt 4

$$V_4 = \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 1 & 2 & 1 & 0 \end{pmatrix}$$

Tabelle 10: Vorgängermatrix nach Schritt 4

Abschließend, also Schritt 5, ergeben sich die letzten Einträge in die Matrizen.

$$D_5 = \begin{pmatrix} 0 & 3 & 2 & 4 & 1 \\ \mathbf{-1} & 0 & -1 & \mathbf{3} & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \mathbf{0} & \mathbf{3} & -2 & 0 & -1 \\ 1 & 4 & 3 & 5 & 0 \end{pmatrix}$$

Tabelle 11: Distanzmatrix nach Schritt 5

$$V_5 = \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ \mathbf{5} & 0 & 2 & \mathbf{1} & 2 \\ 0 & 0 & 0 & 0 & 0 \\ \mathbf{5} & \mathbf{1} & 4 & 0 & 4 \\ 5 & 1 & 2 & 1 & 0 \end{pmatrix}$$

Tabelle 12: Vorgängermatrix nach Schritt 5

Nachdem jetzt alle 5 Knoten einmal ein j gewesen sind, terminiert der Algorithmus. Er hat nun alle kürzesten Wege zwischen allen paaren von Knoten berechnet. In der Distanzmatrix stehen die Längen der kürzesten Wege zwischen zwei Knoten. In der Vorgängermatrix stehen, nachdem der Algorithmus terminierte, alle Wege mit den jeweiligen Knoten, welche passiert werden müssen.

Die Analyse

Wie an dem obigen Beispiel leicht zu erkennen war, benötigt der Algorithmus n Durchläufe, damit jeder Knoten einmal ein j war. Für jeden dieser Durchläufe wurde für alle i und k einmal geprüft, ob j auf diesem Weg liegt. Daraus folgen weitere n * n Durchläufe. Da bei jedem Durchlauf auch noch Zuweisungen und Überprüfungen durchgeführt werden, führt das insgesamt zu:

$$O(n^3)$$

Der Speicherbedarf lässt sich genau bestimmen, da es insgesamt nur 2 Matrizen gibt, auf denen die ganze Zeit gearbeitet wird. Es ergibt sich ein Speicherbedarf von:

$$2 * n$$

Trotz der schlechten Laufzeiteigenschaften des Algorithmus sind zwei Vorteile erkennbar gegenüber anderen Ansätzen. Da keine komplexe Datenstruktur verwendet werden muss, lässt sich der Algorithmus einfach und schnell implementieren. Der zuvor dargestellte Pseudocode lässt sich fast einfach kopieren und dann compilieren.

Ein weiterer Vorteil des Floyd-Warshall-Algorithmus besteht darin, dass dieser auch einfach zu parallelisieren ist. Ein Beispiel als Pseudocode:

```
1. for k := 1 to n do
2.    for each Pij , where 1 < i, j < n do in parallel
3.       dij := min {dij , dik + dkj }
```

Quelle: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter43.html

Mit Pij ist in disem Fall der Prozessor gemeint, welcher die Bedingung

$\min \{d_{ij}, d_{ik} + d_{kj}\}$ überprüft.

Dies bedeutet, dass die inneren beiden Schleifen mit den jeweiligen Paarüberprüfungen nicht sequenziell ausgeführt werden, sondern dass so viele Paare überprüft werden wie Prozessoren vorhanden sind.

Wenn man diesen Algorithmus anhand von NVIDIAs CUDA[5] Technologie parallelisieren würde, könnte man mit einer handelsüblichen NVIDIA GeForce GTX 280 mehr als 30.000 Paare gleichzeitig prüfen.[6]

5 http://www.nvidia.de/object/cuda_what_is_de.html
6 NVIDIA CUDA C Programming Best Practice Guide, July 2009, S. 1

Entscheidender Nachteil des Algorithmus ist allerdings der, dass er ein statischer ist. Demzufolge müssen bei einem Kantenupdate alle Wege neuberechnet werden.

Für eine Anwendung in der Verkehrsnavigation ist das nicht zumutbar. Wie oben schon erwähnt, dauert eine Neuberechnung exponentiell lange. Wenn in einem Verkehrsnetz beispielsweise ein Kantenupdate auftritt -z.B ein Stau- müssen alle kürzesten Wege zwischen allen Paaren von Knoten neuberechnet werden.

Da diese Berechnung sehr lange dauert, hätte sich der Stau vermutlich schon aufgelöst, bevor dieses Hindernis in den kürzesten Wegen mitberücksichtigt würde.

Innovative Verfahren zur Dynamisierung

In diesem Kapitel sollen innovative Verfahren zur Dynamisierung vorgestellt werden, die das zuletzt erwähnte Problem lösen sollen. Dazu werden zwei verschiedene Verfahren vorgestellt.

Die erste vorgestellte Methode berechnet alle Wege zwischen allen Paaren von Knoten im Graphen. Diese werden gespeichert, dann werden verschiedene Algorithmen eingesetzt, um die kürzesten Wege zu berechnen und um Kantenupdates einzutragen.

Im zweiten Ansatz wurde ein neuer Algorithmus entwickelt. Dieser reduziert die zu betrachtenden Kanten im Graphen derart, dass die kürzesten Wege zwischen allen Paaren von Knoten im Graphen deutlich schneller berechnet werden können als mit dem Floyd-Warshall Algorithmus.

Der Ansatz von Jeffrey Miller

In diesem Abschnitt soll auf die Grundideen von Jeffrey Miller eingegangen werden. Abschließend sollen dann die von Jeffrey Miller gemessenen Ergebnisse dargestellt und einer Analyse unterzogen werden.

Zur Person Jeffrey Miller: Jeffrey Miller lebt in Anchorage/Alaska. An der dortigen Universität besetzt er eine Stelle als Assistent Professor. Seine Doktorarbeit, die er im Mai 2007 fertigstellte, beschäftigt sich mit der Analyse und Anwendung der im nachfolgenden vorgestellten Algorithmen.

Er unterscheidet drei Klassen von Algorithmen, welche das Problem „alle kürzesten Wege zwischen allen Paaren von Knoten in einem Graphen“ lösen.

1. Naive Algorithmen

   Als „naive Algorithmen“[7] bezeichnet Jeffrey Miller die erste Klasse von Algorithmen. Unter naiven Algorithmen versteht er den in dieser Arbeit vorgestellten Floyd-Warshall Algorithmus sowie den Johnson-Algorithmus. Wie schon oben erwähnt sind die „naiven Algorithmen“ nicht nutzbar, wenn eine dynamische Anwendung erstellt werden soll. Diese Klasse von Algorithmen ist viel zu langsam, um bei einem Kantenupdate alle kürzesten Wege neu zu berechnen.

2. „dynamic algorithms class“[8]

   Unter der „dynamischen Klasse“ versteht Jeffrey Miller den Ansatz den Demetrescu und Italiano verfolgen. Dieser Ansatz soll im letzten Subkapitel noch ausführlicher beschrieben werden. Jeffrey Miller schreibt, dass diese Klasse von Algorithmen die „naiven Algorithmen“ verbessert.

3. „pre-computed algorithms class“

   Diese Klasse beschreibt den Ansatz, den er verfolgt. Die Idee dieser Algorithmen wird folgendermaßen beschrieben: Es werden alle Wege zwischen allen Paaren von Knoten im Graphen vorausberechnet und abgespeichert.

7 Miller, Jeffrey. "Dynamically Computing Fastest Paths for Intelligent Transportation Systems." IEEE Intelligent Transportation Systems Magazine, Volume 1, Number 1, Spring 2009. S. 3

8 Miller, Jeffrey. "Dynamically Computing Fastest Paths for Intelligent Transportation Systems." IEEE Intelligent Transportation Systems Magazine, Volume 1, Number 1, Spring 2009. S. 3

Jeffrey Miller unterscheidet in dieser letzten Klasse wieder 3 verschiedene Algorithmen, welche alle für eine spezielle Operation optimiert sind. Bevor die eigentlichen Updateoperationen oder das Auffinden der kürzesten Wege gestartet werden kann, müssen alle Wege einmal in $O(n^2 * E!)$ vorausberechnet werden. E ist die Menge der Kanten im betrachteten Graphen.

Daraus ergibt sich dann eine Unterklasse, die er „pre-computed-constant-query-class“[9] nennt. Diese soll den kürzesten Weg in den vorberechneten Wegen in konstanter Zeit, also $O(1)$ finden.

Diese Unterklasse hat aber einen Nachteil. Um ein Kantenupdate einzupflegen, benötigt diese Unterklasse $O(n^2 * h)$.

h steht hier für die Anzahl der Wege, die für den bestimmten Graphen berechnet wurden.

In der zweiten Unterklasse, die von Jeffrey Miller als „pre-computed-constant-update-class“[10] bezeichnet wird, soll ein Kantenupdate in $O(1)$ eingepflegt werden.

Auch in dieser Unterklasse gibt es wieder einen Nachteil. Der kürzeste Weg für ein bestimmtes Paar von Knoten wird in $O(n * h)$ gefunden.

Die letzte Unterklasse, die Jeffrey Miller beschreibt, hat nun das Ziel, nur die Vorteile der beiden zuletzt erwähnten Klassen zu vereinen. Dabei sollen natürlich die Schwächen der Klassen abgemildert werden. Jeffrey Miller hat nun eine Aufwandsabschätzung durchgeführt, die folgende Ergebnisse geliefert hat:

- Kantenupdates in: $O(n^2 * Anzahl\ Pfade)$
- Kürzester Weg gefunden in: $O(Anzahl\ Pfade)$

9 Miller, Jeffrey. "Dynamically Computing Fastest Paths for Intelligent Transportation Systems." IEEE Intelligent Transportation Systems Magazine, Volume 1, Number 1, Spring 2009. S. 3f

10 Miller, Jeffrey. "Dynamically Computing Fastest Paths for Intelligent Transportation Systems." IEEE Intelligent Transportation Systems Magazine, Volume 1, Number 1, Spring 2009. S. 3f

Nachdem alle Klassen vorgestellt wurden, sollen im folgenden Abschnitt Testergebnisse dargestellt werden.

Als Testgraph wurde das Highway-Netz in Los-Angeles Downtown gewählt. Knoten sind in diesem Graphen Highway-Auf- und Abfahrten. Eine Kante repräsentiert den Weg zwischen zwei Auf- bzw. Abfahrten. Es gibt 97 Knoten und 101 Kanten. Auf diesem Graphen wurden 10.000 Updates und 10.000 Anfragen ausgeführt und anschließend die Geschwindigkeiten der einzelnen Algorithmenklassen verglichen.

Algorithmen	Pre-computation	Kantenupdate	Schnellsten Weg finden
Naive	0	13229	0
Demetrescu & Italiano	0	3358	18
Pre-computed-constant-update	31545	0	7
Pre-computed-constant-query	31545	2091	0
Pre-computed-hybrid	31545	9	0

Tabelle 13: Zeiten in Millisekunden.[11]

Die Tabelle zeigt, dass die Algorithmen von Jeffrey Miller schneller sind als die anderen beiden.

Diese Ergebnisse müssen nun aber kritisch betrachtet werden. Hier wurde ein sehr kleiner Graph gewählt, was die Aussagekraft erheblich einschränkt. Es bleibt fraglich, wie sich diese Algorithmen bei dichteren und größeren -mehr Knoten und mehr Kanten- Graphen verhalten. Würde auch ein größerer Graph gewählt werden, wird die Vorberechnungszeit deutlich mehr ins Gewicht fallen. Diese Vorausberechnung wird aber nur dann benötigt, wenn eine neue Kante in den Graphen eingefügt werden soll.

Ein Nachteil gegenüber den anderen Methoden ist der, dass der Ansatz von Jeffrey Miller deutlich Speicherintensiver ist. Es müssen alle Wege zwischen allen Paaren von Knoten gespeichert werden. Bei einem deutlich größeren Graphen -ein größeres Straßennetz- ist mit einem deutlich höheren Hardware-Aufwand zu rechnen. Jeffrey Millers Algorithmen könnten also nur in einer Serverfarm eingesetzt werden.

11 Miller, Jeffrey. "Dynamically Computing Fastest Paths for Intelligent Transportation Systems." IEEE Intelligent Transportation Systems Magazine, Volume 1, Number 1, Spring 2009. S. 4

Der Ansatz von Demetrescu & Italiano

Zu den beiden Entwicklern dieses Ansatzes:

- Camil Demetrescu:

  Geboren 1971. Italienischer theoretischer Informatiker, lehrt an der Universität „La Sapienza“ in Rom.

- Guiseppe Italiano:

  Geboren 1961. Italienischer theoretischer Informatiker, lehrt auch an der Universität „La Sapienza“ in Rom. Italiano ist Demetrescus Doktorvater.

Die Methode von Demetrescu & Italiano unterscheidet sich grundlegend von Jeffrey Millers Ansatz. Nach Jeffrey Miller werden einfach alle Wege zwischen allen Paaren von Knoten gespeichert. Es wird also der gesamte Graph betrachtet.

In der Methode von Demetrescu & Italiano hingegen wird die Menge der zu betrachteten Kanten reduziert, die zur Berechnung der kürzesten Wege nötig sind.

Um die Arbeitsweise des Algorithmus von Demetrescu & Italiano beschreiben zu können, müssen zuvor noch einige Grundprinzipien dargestellt werden. Diese Prinzipien bilden die Grundlage des später beschriebenen Algorithmus.

Die Arbeitsweise des Algorithmus basiert auf der Idee eines „uniform paths“.[12]

Hierzu sind auf der nächsten Seite zwei Beispiele aufgeführt. Beide Beispiele zeigen jeweils einen „uniform path“, wobei nur bei einem Beispiel der „uniform path“ auch gleichzeitig der kürzeste Weg ist.

12 C. Demetrescu & G. F. Italiano, Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms, S.5 f

Definition „uniform path“:

Ein Weg π wird „uniform path“ genannt, wenn jeder Subweg von π ein kürzester Weg ist. Der Weg π selbst muss nicht zwangsläufig auch ein kürzester Weg sein.

Der „uniform path“ ist in den Beispieabbildungen grün markiert.

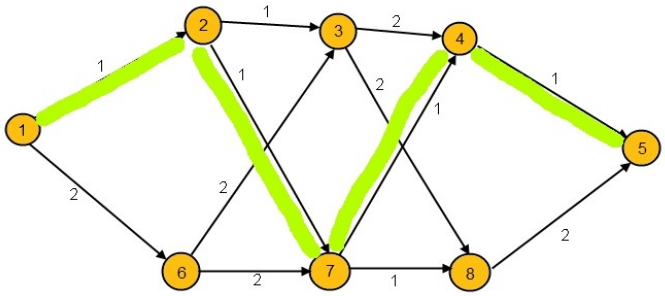

Abbildung 4: "uniform path" der gleichzeitig auch der kürzeste Weg ist

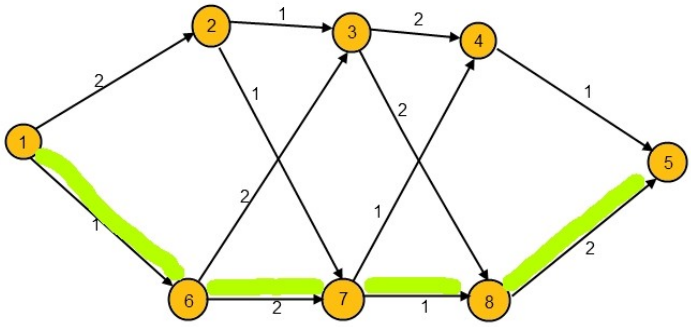

Abbildung 5: "uniform path" der nicht gleichzeitig der kürzeste Weg ist

Zwei weitere Definitionen werden noch benötig, bevor die Arbeitsweise des Algorithmus erläutert werden kann:

Definition „historical shortest Path“:

> Ein Weg wird „historical shortest Path“ genannt, wenn dieser Weg während der Updatesequenz ein kürzester Weg war.

Definition „potentially uniform path“:

> Ein Weg wird „potentially uniform path“ genannt, wenn jeder Subweg dieses Weges ein „historical shortest Path“ ist.

Die Idee, auf welche weise der Algorithmus arbeiten soll, wird folgendermaßen dargestellt:

Es soll dynamisch die Menge der „potentially uniform paths“ gepflegt werden. Kürzeste Wege und „uniform paths“ sind Spezialfälle von „potentially uniform paths“.

Um möglichst wenige Veränderungen in der Menge der „potentially uniform paths“ zu haben, wird die update Sequenz „on-the-fly“ so modifiziert, dass möglichst wenig „historical shortest paths“ entstehen. Durch diese Transformation wird die update Sequenz um $O(\log(n))$ vergrößert. Die Transformation benötigt zusätzlich $O(\log(n))$ Zeit.

Um eine update Sequenz in die Wegberechnung einfließen zu lassen, arbeitet der Algorithmus folgendermaßen:

- Im ersten Schritt werden alle gepflegten Pfade gelöscht, die die geupdatete Kante enthalten. Das bedeutet, jeder Pfad, der die betreffende Kante enthält, wird aus der Menge der „potentially uniform paths“ entfernt.
- Dann wird von allen Knoten eine dynamische Abwandlung des Dijkstra-Algorithmus gestartet.
- Bei jedem Berechnungsschritt wird ein kürzester Weg mit den geringsten Wegkosten aus einer Vorrangwarteschlage entnommen und mit einem „historical shortest path“ verbunden, um einen neuen „potentially uniform path“ zu formen.

Wenn der Dijkstra-Algorithmus mit einem Fibonacci-Heap[13]implementiert wird, ergibt sich für den gesamten Algorithmus folgende Komplexität:

$$O(n^2 \log^3(n))$$

Die zusätzlichen zwei Logarithmen ergeben sich aus den oben erwähnten Transformationen der update Sequenz und der daraus resultierenden Vergrößerung der Sequenz.

13 http://www-i1.informatik.rwth-aachen.de/Lehre/SS05/PSAuD/Handout_HurtzSchiffel.pdf

Schlussbemerkung

In dieser Arbeit sind drei verschiedene Methoden vorgestellt worden, die das Problem des Auffindens der kürzesten Wege zwischen allen Paaren von Knoten in einem Graphen lösen.

Abschließend sollen nun noch einmal die Ansätze nebeneinander gestellt und kritisch beleuchtet werden.

Der Floyd-Warshall Algorithmus hat den verheerenden Nachteil, dass er exponentiell lange braucht, um die kürzesten Wege zu berechnen. Weiter kann der Algorithmus nicht auf dynamische Veränderungen des Graphen reagieren, ohne einmal eine komplette Neuberechnung durchzuführen. Wie oben schon erwähnt, ist deswegen ein Einsatz bei praktischen Problemen nicht gegeben.

Miller, Demetrescu&Italiano sind auf diese Probleme eingegangen und haben die Nachteile anhand ihrer eigenen Ansätze verbessert.

Jeffrey Millers Methode besteht darin, alle Wege vorauszuberechnen und benötigt deswegen viel Rechenleistung und Speicherplatz. Demetrescus und Italianos Lösungsweg hingegen reduziert die zu betrachtenden Kanten im Graphen, was unter anderem dazu führt, dass dieser Algorithmus besonders bei dichten Graphen besonders schnell ist.[14] Weiterhin benötigt der Algorithmus nicht soviel Speicherplatz, weswegen eine geringerer Hardware-Aufwand zu erwarten ist.

Millers Anliegen bestand darin, dass er dynamische Verkehrsinformationssysteme entwickeln wollte, die schneller aktuelle Verkehrsinformationen in ihre Routenberechnungen integrieren könnten.

Demetrescu&Italiano verfolgten das Ziel, einen schnelleren Algorithmus für das erwähnte Grundproblem zu entwickeln und dieses anhand eines Programms zu zeigen. Eine echte Anwendung dieses Algorithmus mit aktuellen Verkehrsinformationen steht noch aus.

Auf dem Wege nach Lösungsstrategien dieses Themenbereichs scheinen die Ansätze Demetrescus&Italianos die aussichtsreicheren zu sein. Dies liegt besonders daran, dass Verkehrsnetze schlichtweg sehr große und dichte Graphen sind. Millers Ansatz könnte bei entsprechend großen Graphen an seine Grenzen stoßen.

Aus meiner Sicht werden allerdings Ameisensysteme auf dem Gebiet der dynamischen Verkehrsnavigation die erfolgreicheren Systeme sein. Dies liegt vorallem in der Natur dieser Systeme.

Da es leider für die beiden zuletzt erwähnten Methoden noch keine Beispielsysteme aus der Praxis vorhanden sind, mit denen echte Praxisvergleiche gezogen werden können, lässt sich an dieser Stelle keine abschließende Bewertung erbringen. Hier besteht noch ein besonderer Forschungsbedarf.

---

14 C. Demetrescu & G. F. Italiano, Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms, S.9 f

Quellen

- Turau, Volker: Algorithmische Graphentheorie, 2. Auflage, 2004
- *C. Demetrescu and G. F. ItalianoExperimental Analysis of Dynamic All Pairs Shortest Path AlgorithmsTo appear in ACM Transactions on Algorithms (TALG), 2005. Special Issue devoted to the best papers selected from the 15th Annual ACM-SIAM Symposium*
- *C. Demetrescu and G.F. ItalianoA New Approach to Dynamic All Pairs Shortest PathsJournal of the Association for Computing Machinery (JACM), 51(6), pp. 968-992, November 2004*
- Miller, Jeffrey. "Dynamically Computing Fastest Paths for Intelligent Transportation Systems." IEEE Intelligent Transportation Systems Magazine, Volume 1, Number 1, Spring 2009
- Miller, Jeffrey. "Algorithms and Data Structures for the Real-Time Processing of Traffic Data." *Ph.D. Dissertation*, University of Southern California, Los Angeles, California, USA, April 27, 2007

Für weitere Informationen:

- http://www.dis.uniroma1.it/~demetres/
- http://www.sigmacoding.com/jeff/publications.html