Ausarbeitung

Informatik Seminar (Algorithmen) Sommersemester 2008

Thema: Dynamisches Programmieren 1
Algorithmen von Warshall und Floyd

Inhaltsverzeichnis

1 Dynamische Programmierung

Bei der Dynamischen Programmierung handelt es sich um ein optimierungs-Verfahren welches von Richard Bellman in den Fünfziegerjahren entwickelt wurde.

Das Verfahren der dynamischen Programmierung besteht darin, zuerst die optimalen Lösungen der kleinsten Teilprobleme direkt zu berechnen, und diese dann geeignet zu einer Lösung eines nächst größeren Teilproblems zusammenzusetzen, und so weiter. Einmal berechnete Teilergebnisse werden in einer Tabelle gespeichert. Bei nachfolgenden Berechnungen gleicher Teilprobleme wird auf diese Zwischenlösungen zurückgegriffen, anstatt sie jedes Mal neu zu berechnen.

Um also ein Optimierungsproblem mit Dynamischem Programmieren zu lösen, muss man es nicht wie beim Greedy- Verfahren von oben, sondern von unten versuchen zu lösen, in dem man sich die resultierenden Restprobleme ansieht.

1.1 Charakteristische Eigenschaften

Um herauszufinden ob ein Problem mit Dynamischem Programmieren gelöst werden kann, muss man es auf diese Eigenschaften prüfen.

1. **Die Optimale Teilstruktur Eigenschaft (OTE)**

   Die OTE ist dann erfüllt, wenn die optimale Lösung des Problems die optimalen Lösungen aller Teilprobleme enthält.
   Also, man löst erst die kleinen Teilprobleme um dann die größeren daraus zusammenzusetzen.

Wenn man die erste Eigenschaft nachgewiesen hat, kann das Problem prinzipiell mit Dynamischem Programmieren gelöst werden.
Es macht aber erst wirklich sinn, wenn die zweite Eigenschaft hinzukommt.

2. **Auftreten überlappender Teilprobleme**

   Wenn man bei der rekursiven Lösung des Problems mit der OTE feststellt, dass identische Teilprobleme mindestens zweimal gelöst werden müssen, dann besitzt es auch die zweie Eigenschaft.

Umso öfter diese Teilprobleme gelöst werden müssen umso sinnvoller ist die Anwendung der Dynamischen Programmierung, denn die Idee beim Dynamischen Programmieren ist ja grade diese Teilprobleme nur einmal zu lösen und das Ergebnis in einer Matrix oder Tabelle zu speichern.

Wenn es keinen Teilprobleme gibt die mehrmals gelöst werden müssen kann man gleich bei der Rekursiven Lösung bleiben, denn man gewinnt hier nichts durch die Dynamische Programmierung.

2 Berechnung der Transitiven Hülle in einem unbewerteten gerichteten Graphen.

Die transitive Hülle

Wir definieren auf einen beliebigen unbewerteten gerichteten Graphen eine Relation R auf A = {Knoten_1,Knoten_2…Knoten_n}.

Diese Relation R erweitern wir zu R* indem wir den Transitiven Abschluss bilden.

Unter Abschluss versteht das zufügen von Paaren bis die gewünschte Eigenschaft erreicht ist mit der Bedingung das R* hinterher minimal ist.

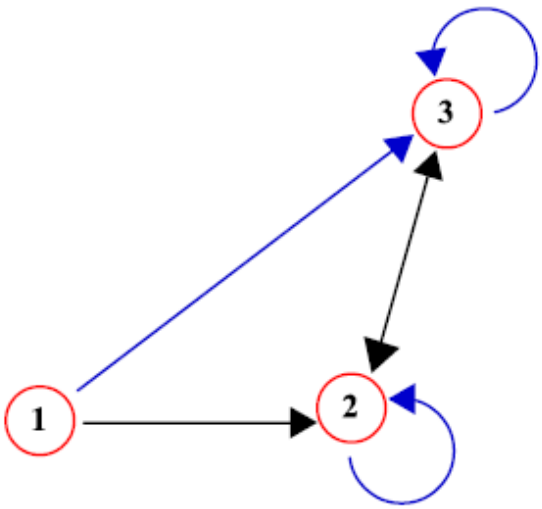
transitiv: wenn (x R y und y R z --> x R z) für alle x,y,z aus A.

Die Transitive Hülle ist der Transitive Abschluss dieser Relation!

Im Beispiel sieht es dann so aus.

	1	2	3
1	F	W	W
2	F	W	W
3	F	W	W

Adjazenzmatrix



R = {(1,2),(2,3),(3,2)}
R* = {(1,2),(2,3),(3,2),(1,3),(2,2),(3,3)}

Die Blauen Pfeile werden für den Transitiven Abschluss hinzugefügt.

Man hat jetzt eine neue Adjazenzmatrix in der die indirekten Verbindungen direkt geworden sind.

2.1 Die Optimale Teilstruktur Eigenschaft bei der Berechnung der Transitiven Hülle.

Wir gehen davon aus, dass es nur zwei Möglichkeiten gibt einen Knoten zu erreichen.

1. Direkt
   (Y → X)
2. Indirekt
   (Y → N → X)
   N steht hier für beliebig viele Knoten.

Wenn wir uns jetzt die indirekten Wege ansehen stellen wir fest, dass diese ja aus den Direkten zusammengesetzt sind.

Also, wenn es einen Weg gibt von (Y → N) und einen von (N → X), dann existiert auch zwangsläufig ein Weg von (Y → X).

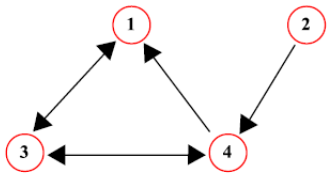
Damit haben wir unsere Optimale Teilstruktur Eigenschaft gefunden, wir sehen die größeren Probleme werden aus den kleineren zusammengesetzt!

2.2 Auftreten überlappender Teilprobleme bei der Berechnung der Transitiven Hülle.

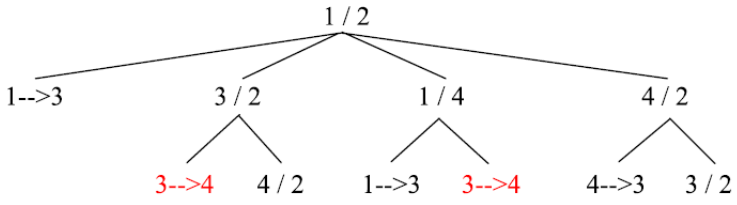
Wenn wir uns jetzt auf der Basis der eben gefundenen "Optimalen Teilstruktur Eigenschaft" eine rekursive Lösung überlegen, könne wir die überlappenden Teilprobleme feststellen.

Beispiel:

	1	2	3	4
1	F	F	W	F
2	F	F	F	W
3	W	F	F	W
4	W	F	W	F



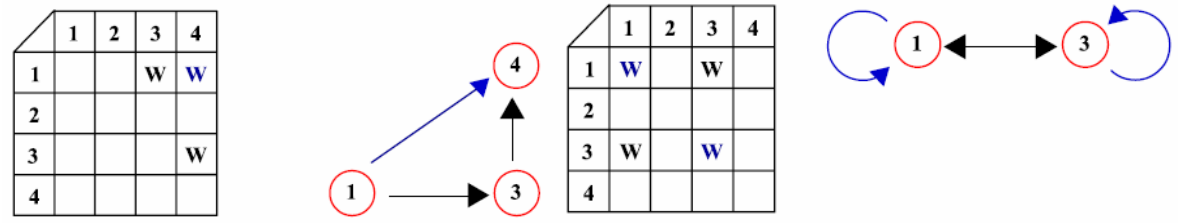
Gibt es einen weg von 1 zu 2??



Wir sehen, dass schon bei relativ wenig Knoten 4 Wege doppelt berechnet wurden. Das Problem besitzt also auch die Zweite Eigenschaft die überlappenden Teilprobleme und damit Ist eine Lösung mit Dynamischem Programmieren hier auch sinnvoll.

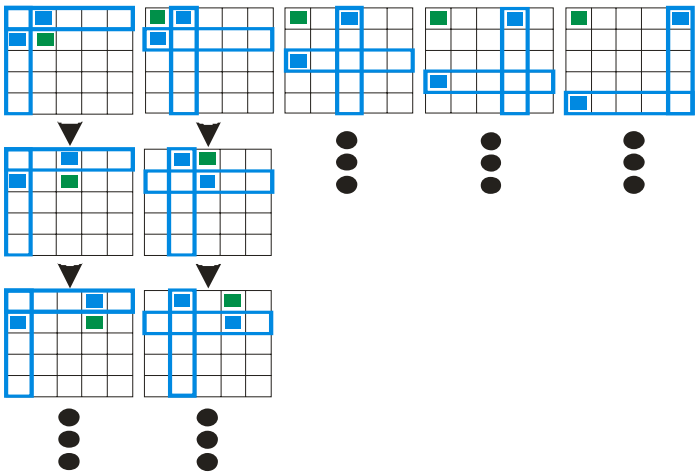
2.3 Der Algorithmus von Warshall

Kommen wir zu Umsetzung, hier wird ausgenutzt das die Transitivität in der Adjazenzmatrix immer rechtwinklig zu finden ist.




```
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            M[i][j] = M[i][j] oder (M[i][k] und M[k][j]);
```

Mit den drei Vorschleifen wird die Adjazenzmatrix so durchlaufen das immer rechtwinklig geprüft wird.



Man guckt also immer, wenn es keinen direkten Weg gibt, ob es einen Indirekten Weg gibt über einen anderen Knoten.
Wenn ein indirekter Weg gefunden wurde wird er als direkter Weg gespeichert.
Im weiteren Verlauf wird dieser Weg wider zum vergleich genutzt und so können dann auch Wege über mehrere Knoten gefunden werden.

Also, wir lösen erst die kleinen Probleme um dann die großen daraus zusammenzusetzen!

2.4 Laufzeitverhalten

Zeitkomplexität:	O(n^3)	wegen den 3 Vorschleifen
Speicherkomplexität:	O(n^2)	wegen der Matrix

3 All-Pairs Shortest Path

Wir wollen in einem positiv bewerteten gerichteten Graphen für alle Paare von Knoten die kürzesten Wege berechnen.

Also, wir suchen in einem positiv bewerteten gerichteten Graphen die kürzesten Verbindungen und tragen sie in der Adjazenzmatrix als direkte Verbindungen ein. Damit erhalten wir eine Matrix in der wir jeweils die günstigste Verbindung direkt ablesen können.

3.1 Die Optimale Teilstruktur Eigenschaft beim All-Pairs Shortest Path Problem

Wir gehen wieder davon aus, dass es nur zwei Möglichkeiten gibt einen Knoten zu erreichen.

1. Direkt
   (Y → X)
2. Indirekt
   (Y → N → X)
   N steht hier für beliebig viele Knoten.

Wenn man einen kürzesten indirekten Weg gefunden hat, kann man sich leicht überlegen, dass auch die Teilstrecken kürzeste Wege sein müssen.

(Y → N → X) kürzester Weg

(Y → N) (N → X) also auch kürzeste Wege

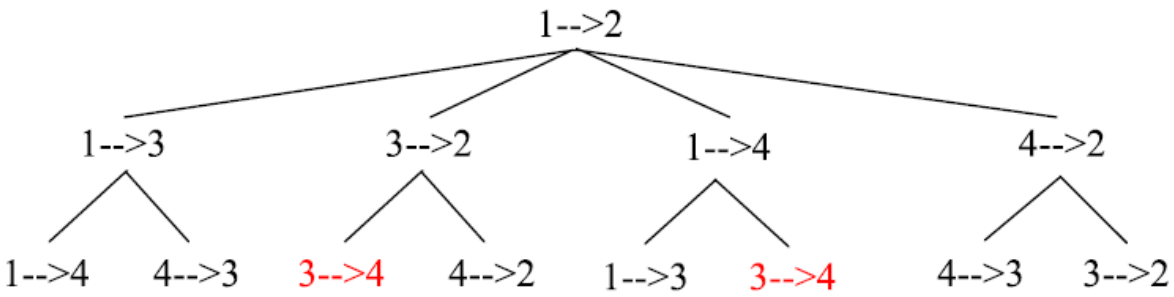
Der Umkehrschluss währe, dass es für eine Teilstrecke einen kürzeren Weg geben würde, was dann zur folge hätte, dass der gesamte Weg nicht mehr der kürzeste Weg währe.

Damit haben wir unsere Optimale Teilstruktur Eigenschaft gefunden, wir sehen die größeren Probleme werden wieder aus den kleineren zusammengesetzt!

3.2 Auftreten überlappender Teilprobleme beim All-Pairs Shortest Path Problem

Wenn wir uns jetzt wieder auf der Basis der eben gefundenen “Optimalen Teilstruktur Eigenschaft“ eine rekursive Lösung überlegen, könne wir nach den überlappenden Teilprobleme suchen.

Beispiel: Wir Versuchen in einem beliebigen Graphen mit 4 knoten den günstigsten Weg von Knoten 1 zu Knoten 2 zu finden.



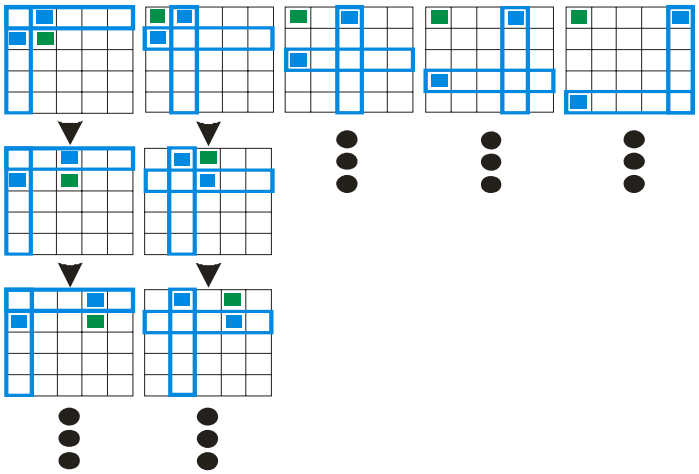
Erst gucken wir nach der direkten Verbindung,
dann nach den Verbindungen über einen anderen Knoten und schließlich ob wir von dem jeweilig anderen Knoten über einen weiteren Knoten zum Ziel kommen.

Wir sehen wieder, dass viele Wege mehrfach berechnet wurden und damit haben wir die überlappenden Teilprobleme gefunden.

Eine Lösung mit Dynamischem Programmieren ist also wieder sinnvoll!

3.3 Der Algorithmus von Floyd

```
for (int k = 1; k < n; k++)
        for (int i  = 1; i  < n; i++)
                for (int j  = 1; j < n; j++)
                        if (m[i][k] + m[k][j])  <  m[i][j]
                                m[i][j] = (m[i][k] + m[k][j]);
```



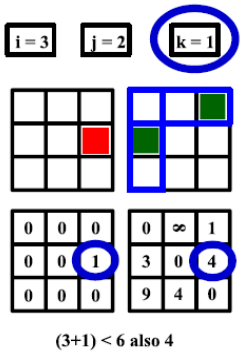
Zur besseren Verdeutlichung wurden die unnötigen Vergleiche übersprungen.

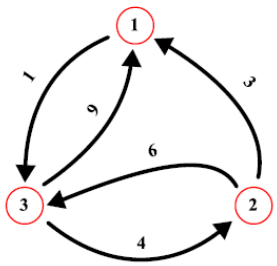
Die Adjazenzmatrix wird in derselben Art und Weise durchlaufen wie beim Warshall Algorithmus, nur das hier nicht auf Existenz eines Weges geprüft wird, sondern ob der jeweilige Weg über den anderen Knoten kürzer ist.
Wenn ein kürzerer Weg gefunden wurde, wird dieser im weiteren Verlauf wieder zum vergleichen genutzt und so können dann auch wieder kürzeste Wege über mehrere Knoten erkannt werden.

Also, wir lösen erst wieder die kleinen Probleme um dann die großen daraus zusammenzusetzen!

3.3.1 Die Pfadmatrix

Es gibt jetzt noch die Möglichkeit die Pfade zu den kürzesten Wegen mit zu speichern, indem man eine weitere Matrix einsetzt.

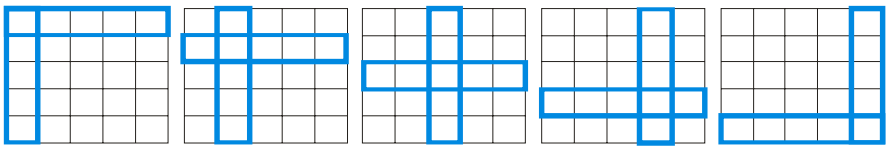




Die Werte an den Beiden Grünmarkiertenpositionen werden addiert und mit dem Wert an der Rotmarkiertenposition verglichen.
In diesem Fall sind 3+1 kleiner als 6.

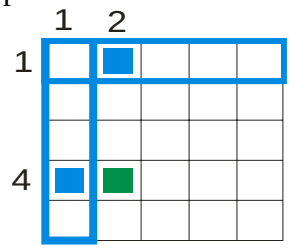
In dieser Matrix wird immer, wenn ein kürzerer Weg gefunden wurde, der k Index an derselben Stelle gespeichert.

Der k Index ist nichts anderes als der Knoten über den der kürzere Weg gefunden wurde.

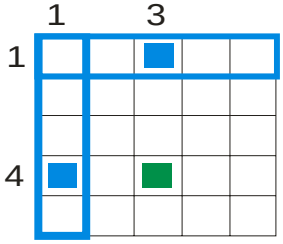


In den blauen Bereichen finden jeweils für ein k die vergleiche statt.

Beispiel:

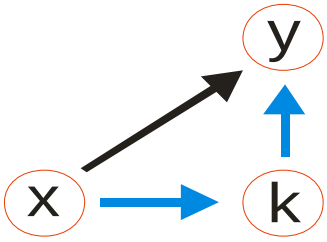


Die Abfrage ist hier:
Ist der Weg von 4 zu 2 kürzer als der Weg von 4 zu 1 plus den Weg von 1 zu 2.



Die Abfrage ist hier:
Ist der Weg von 4 zu 3 kürzer als der Weg von 4 zu 1 plus den Weg von 1 zu 3.

und so weiter…



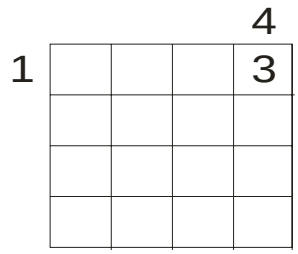
Wir gucken also immer ob der Direkte Weg länger ist als
der Indirekte Weg über k.

Ablauf:

Die Pfadmatrix wird mit Nullen initialisiert und nach und nach mit den kürzeren indirekten Wegen gefüllt.
Eine Null steht hier dafür, dass die direkte Verbindung die kürzeste ist, weil ja kein kürzerer indirekter Weg gefunden wurde.
Beim Auswerten ist hier zu beachten, dass auch dann eine Null vorhanden ist wenn überhaupt kein Weg existiert.
Man muss also erst in der anderen Matrix nachschauen ob ein Weg existiert.

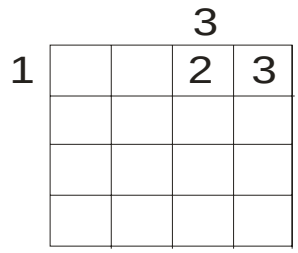
3.3.2 Auswerten der Pfadmatrix

Wir gucken in der wagerechten vom Knoten x und in der senkrechten zum Knoten y (wie in der Adjazenzmatrix).
Denn wir haben die indirekten kürzeren Wege ja an denselben Positionen gespeichert, wo wir in der anderen Matrix auch die neuen Kosten eingetragen haben.



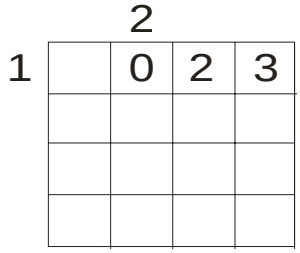
Von der 1 zur 4 müssen wir über die 3.
Jetzt wissen wir, dass der indirekte kürzeste Weg zur 4 über die 3 geht.

Es könnte aber jetzt noch sein das der direkte Weg zur 3 nicht der kürzeste ist.



Von der 1 zur 3 müssen wir über die 2.
Jetzt wissen wir, dass der indirekte kürzeste Weg zur 3 über die 2 geht.

Wir sind aber erst fertig wenn wir eine Null gefunden haben, denn die Null bedeutet hier, dass die direkte Verbindung die Kürzeste ist.



Von der 1 zur 2 ist der direkte Weg der kürzeste.
Jetzt wissen wir, dass wir fertig sind.

1→2→3→4

jetzt haben wir den kürzesten Weg von hinten nach vorne rekonstruiert.
Der gefundene Knoten in der Pfadmatrix ist also immer, der letzte Konten des Weges bei dem man grade nachschaut.

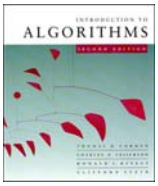
3.4 Laufzeitverhalten

Zeitkomplexität: O(n^3) wegen den 3 Vorschleifen
Speicherkomplexität: O(2n^2) wegen den 2 Matrizen

4 Literatur

Introduction to the Design and Analysis of Algorithms
Anany V. Levitin

Algorithmen und Datenstrukturen
Bernd Owsnicki-Klewe



Introduction to Algorithms
Thomas H. Cormen

Diskrete Mathematik für Informatiker
Rod Haggarty