Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

JOHANNA RIDDER*

Wegeprobleme der Graphentheorie

* Herder-Gymnasium Berlin

ZIB-Report 08-26 (July 2008)

Johanna Ridder

Herder-Gymnasium

14.12.2007

Wegeprobleme der Graphentheorie

Facharbeit

Vorwort

Den kürzesten Weg in einem Graphen zu finden ist ein klassisches Problem der Graphentheorie. Über einen Vortrag zu diesem Thema beim Tag der Mathematik 2007 von R. Borndörfer kam ich in Kontakt mit dem Konrad-Zuse-Zentrum (ZIB), das sich u.a. mit Wegeoptimierung beschäftigt. Ein Forschungsschwerpunkt dort ist im Rahmen eines Projekts zur Chipverifikation das Zählen von Lösungen, das, wie wir sehen werden, eng mit dem Zählen von Wegen zusammenhängt.

Anhand von zwei Fragen aus der Graphentheorie soll diese Facharbeit unterschiedliche Lösungsmethoden untersuchen. Wie bestimmt man den kürzesten Weg zwischen zwei Knoten in einem Graphen und wie findet man alle möglichen Wege?

Nach einer Einführung in die Graphentheorie und einer Konkretisierung der Probleme wird zunächst für beide eine Lösung mit auf Graphen basierenden Algorithmen vorgestellt. Während der Algorithmus von Dijkstra sehr bekannt ist, habe ich für das Zählen von Wegen einen eigenen Algorithmus auf der Basis der Tiefensuche entwickelt.

Im zweiten Teil der Arbeit wird das Konzept der ganzzahligen Programmierung vorgestellt und die Lösungsmöglichkeiten für Wegeprobleme, die sich darüber ergeben.

Schließlich wurden die vorgestellten Algorithmen am Beispiel des S- und U-Bahnnetzes von Berlin implementiert und mit Programmen, die die gleichen Fragen über ganzzahlige Programmierung lösen, verglichen.

14.12.2007

Allgemeines

Facharbeit für das Abitur 2008

Titel	Wegeprobleme der Graphentheorie
Fach	Informatik
Referenzfach	Mathematik
Betreuender Lehrer	Herr Nuck
Externe Betreuer	Herr Borndörfer, Herr Heinz (ZIB)
Schule	Herder-Gymnasium

Inhalt

1 Graphentheorie

1.1 Zur Geschichte

Die Ursprünge der Graphentheorie finden sich in einem Problem, mit dem sich im Jahr 1736 der Mathematiker Leonhard Euler beschäftigte: Er fragte sich, ob es für die Bewohner der Stadt Königsberg möglich sei, einen Spaziergang zu machen, auf dem alle sieben Brücken der Stadt genau einmal überquert werden. Um zu zeigen, dass das bei der Verteilung der Brücken nicht möglich war, verallgemeinerte Euler das Problem und fand eine einfachere Darstellung: Die Brücken wurden Kanten und die durch sie verbundenen Landteile die Knoten eines Graphen.

Seit Euler hat sich die Graphentheorie weit entwickelt. Neben kleinen Problemen und Spielchen, in denen sie Anwendung findet, wie z. B. das „Ikosaeder-Spiel" von Hamilton, der Weg durch ein Labyrinth oder das Färben von Landkarten, ist sie zu einem eigenen Teilgebiet der Mathematik geworden und findet vielfach wichtige Anwendung in der Informatik. Am Computer und in Netzwerken lassen sich zahlreiche Probleme auf Graphen und z. B. die Suche nach einem bestimmten Weg in diesem Graphen zurückführen.

1.2 Definitionen

1.2.1 Ein ungerichteter Graph

Ein ungerichteter Graph G ist definiert durch

$G = (V, E), E \subseteq V \times V \setminus \left\{ v \in V \middle| (v, v) \right\}$[1]

V: Die Menge der Knoten (englisch: Vertices)[2]

E: Die Menge der Kanten (englisch: Edges)

Da die Kanten ungerichtet sind, gilt $\underset{v,w \in V}{\forall} (\mathrm{v}, \mathrm{w}) = (\mathrm{w}, \mathrm{v})$

[1] Bei dieser Definition werden Schlingen (Kanten mit (v,v) $v \in V$) ausgenommen, da sie beim Suchen von kürzesten, bzw. allen Wegen keine Bedeutung haben und einige Betrachtungen unnötig kompliziert machen.

Ich bezeichne mit

$n = |V|$ (Anzahl der Knoten, Ordnung des Graphen G)

$m = |E|$ (Anzahl der Kanten)

Kanten können gewichtet sein, d. h. es gibt eine Abbildung c: E → ℝ

c(e): Gewicht oder Länge der Kante $e \in E$

Bei manchen Algorithmen wird $\forall_{e \in E} c(e) \geq 0$ gefordert.

Für einen Knoten $v \in V$ wird weiterhin definiert:

$\delta(v) = \{e \in E | e = (v,w) \vee e = (w,v)\}_{w \in V}$ (mit v ‚inzidente' Kanten)

$\Gamma(v) = \left\{ w \in V \middle| \exists_{e \in E}\, e \in \delta(v) \wedge e \in \delta(w) \right\}$ (Nachbarn von v)

1.2.2 Ein Digraph

Gerichtete Graphen, die Digraphen genannt werden, haben keine ungerichteten Kanten, sondern gerichtete, die Bögen heißen.

$D = (V, A), A \subseteq V \times V \setminus \{v \in V | (v,v)\}$

A: Die Menge der Bögen (englisch: Arcs)

Für einen Knoten $v \in V$ wird definiert:

$\delta^+(v) = \{a \in A | a = (v,w)\}_{w \in V}$ (Ausgangsbögen von Knoten v)

$\delta^-(v) = \{a \in A | a = (w,v)\}_{w \in V}$ (Eingangsbögen von Knoten v)

Im Übrigen gelten die Definitionen vom ungerichteten Graphen.

Wenn nicht anderes geschrieben wird, wird in dieser Arbeit von Digraphen ausgegangen, da sich alle ungerichtete Graphen leicht in einen Digraphen umformen lassen, indem man alle Kanten durch zwei entgegengesetzt gerichtete Bögen ersetzt.

---

[2] In der Literatur findet man auch oft die Bezeichnung N (englisch: Nodes). Wörtlich übersetzt bedeutet 'node' Knoten und 'vertex' Ecke.

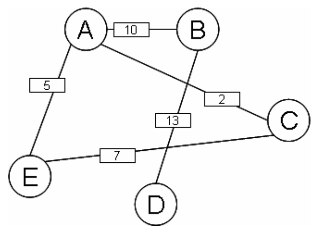
1.2.3 Beispiel

Das definierte abstrakte Konzept des Graphen wird grafisch dargestellt, indem Punkte oder Kreise die Knoten bilden und die Kanten Linien, die diese Punkte verbinden.

Ein ungerichteter Graph $G=(V,E)$ mit V={A,B,C,D,E},

E={(A,B),(A,C),(A,E),(B,D),(C,E)} und

$$c:\begin{cases}(A,B)\mapsto 10\\(A,C)\mapsto 2\\(A,E)\mapsto 5\\(B,D)\mapsto 13\\(C,E)\mapsto 7\end{cases}$$ wird also so dargestellt:

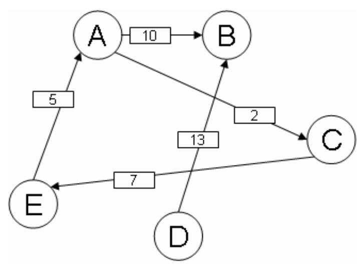


Ein ungerichteter Graph

Ein gerichteter Graph $D=(V,A)$ mit V={A,B,C,D,E},

A={(A,B),(A,C),(C,E),(D,B),(E,A)} und

$$c:\begin{cases}(A,B)\mapsto 10\\(A,C)\mapsto 2\\(C,E)\mapsto 7\\(D,B)\mapsto 13\\(E,A)\mapsto 5\end{cases}$$ so:



Ein gerichteter Graph

1.3 Wege, Kreise und Bäume

In einem Graphen ist ein Weg eine endliche Folge von Knoten und Bögen:

$v_0,a_1,v_1,a_2,v_2,a_3,v_3,\ldots,v_{n-1},a_n,v_n$ mit $n\geq 1$ und $a_i=(v_{i-1},v_i)$ für alle i.

Wobei a einen Bogen und v einen Knoten (vertex) bezeichnet.

Da Mehrfachbögen (Bögen mit gleichen Start- und Zielknoten) ausgeschlossen sind, genügt es aber auch, eine Folge von Knoten anzugeben:

$v_0,v_1, v_2, v_3,\ldots,v_{n-1},v_n$ mit $n\geq 1$ und $(v_i,v_{i+1})\in A$ für alle i.

Mit der Weglänge bezeichnet man die Summe der Kantengewichte aller durchlaufenen Kanten bzw. Bögen.

Ein Weg kann entweder geschlossen (der Anfangspunkt ist gleichzeitig Endpunkt) oder offen sein. Außerdem bezeichnet man ihn als kanteneinfach, wenn keine Kante und kein Bogen mehrmals durchlaufen wird und einfach, wenn alle Knoten paarweise verschieden sind (bzw. in einem geschlossenen Weg alle Knoten bis auf den Anfangs- und Endpunkt). Ein Weg von einem Startknoten s zu einem Zielknoten t nennt man auch s-t-Weg.

Als Kreis bezeichnet man einen geschlossenen, einfachen und kanteneinfachen Weg und entsprechend ist ein kreisfreier Graph ein Graph ohne eine solchen Kreis.

Ein Baum ist allgemein jeder kreisfreie, zusammenhängende Graph. Im gerichteten Baum nennt man einen Knoten, von dem aus alle anderen Knoten zu erreichen sind, Wurzelknoten.

2 Zwei graphentheoretische Wegeprobleme und Lösungsalgorithmen

2.1 Der kürzeste Weg

Das Kürzeste-Wege-Problem ist ein klassisches Problem der Graphentheorie.

Man unterscheidet zwischen dem Problem,

1. den kürzesten Weg von einem Startknoten zu einem Zielknoten oder zu allen anderen Knoten zu finden und

2. die kürzesten Wege zwischen allen Knotenpaaren des Graphen zu finden.

Im Übrigen kann man die Algorithmen, die zur Lösung von 1. benutzt werden, in „label-setting“- und „label-correcting“ –Algorithmen einteilen. „label-setting“-Algorithmen können nur bei Graphen mit nicht negativen Kantengewichten angewendet werden ($c \geq 0$). Zu jedem Zeitpunkt gibt es eine Menge von Knoten, zu denen bereits der kürzeste Weg gefunden wurde.

„label-correcting“-Algorithmen sind dagegen auch auf Graphen mit negativen Kanten anwendbar. Dieses Problem ist komplizierter, da es z. B. keine negativen Kreise geben darf. Bevor der Algorithmus ganz fertig ist, steht bei keinem Knoten der kürzeste Weg, der zu ihm führt, sicher fest.

2.2 Zählen von Wegen

Die Frage nach der Anzahl von Wegen zwischen zwei Punkten in einem Graphen macht nur bei einfachen oder kanteneinfachen Wegen Sinn, das heißt, kein Knoten bzw. keine Kante wird doppelt besucht.

Im Übrigen wollen wir nicht nur die Anzahl der Wege herausfinden, sondern diese Wege auch tatsächlich bestimmen und evtl. ausgeben können.

2.3 Der Dijkstra-Algorithmus

2.3.1 Einleitung

Der gebräuchlichste Algorithmus für das Kürzeste-Wege-Problem ist der *Algorithmus von Dijkstra.* Der Algorithmus findet die kürzesten Wege von einem Startknoten s zu allen anderen Knoten im Graphen, wenn er nicht abgebrochen wird, sobald ein bestimmter Zielknoten t erreicht wird. Er basiert darauf, dass jeder Abschitt des kürzesten Weges zu einem Punkt selbst auch ein kürzester Weg ist. Voraussetzung für den Dijkstra-Algorithmus sind nicht negative Kantengewichte.

2.3.2 Pseudocode

Algorithmus von Dijkstra zum Finden eines kürzesten Weg von Knoten s nach t

Input: D=(V,A)

$\forall_{v\in V} c(v) \geq 0$

$s,t \in V$

Datenstrukturen:

$d: V \to \mathbb{R}_+$ (Distanzlabel)

$p: V \to V$ (Vorgänger)

$V' \subseteq V$ (endgültig markierte Knoten)

Output:

//Initialisierung

$V' := \{s\}$

für $v \in V$:

wenn v Nachbar von s ist

$d(v) := c(u,v)$

$\mathrm{p(v)} := \mathrm{u}$

sonst $d(v) := \infty$

Wiederhole bis $t \in V$ //bzw. bis $\mathrm{V'= V}$

finde unter den Knoten, die nicht in V' sind, den Knoten w, sodass d(w) minimal ist

füge w zu V' hinzu

für jeden Nachbarn v von w, der nicht in V' ist:

/*Aktualisiere d(v) so, dass die neue Weglänge nach v entweder die alte Weglänge oder die bekannte kürzeste Weglänge nach w plus der Kantenlänge von w nach v ist.*/

wenn d(v)> d(w)+c(w,v):

d(v)= d(w)+c(w,v)

p(v)=w

End

2.3.3 Beschreibung

In der englischen Literatur wird der Dijkstra-Algorithmus zu den „label-setting“ Algorithmen gezählt, da jedem Knoten eine Distanzmarke d(v) (ein „label“) zugewiesen wird und es eine Teilmenge V’ von V gibt, in der d(v) den in dieser Suche endgültigen Wert erreicht hat und nicht mehr verändert wird.

d(v) gibt die Weglänge eines Weges vom Ursprungsknoten zum Knoten v an. Dieser Weg verläuft nur über Knoten, die sich in V’ befinden, sodass Knoten, die nicht Nachbarn eines Knotens in V’ sind, unerreicht sind, also $d(v) = \infty$ ist.

Anfangs gehört nur der Startpunkt s zu den Knoten V’ mit endgültig bestimmtem d, seinen Nachbarn lässt sich eine Distanz d zum Ursprungsknoten, nämlich das Gewicht der sie mit dem Startknoten verbindenden Kante, zuordnen.

Im nächsten Schritt wird der Knoten mit der geringsten Entfernung zum Startpunkt zu V’ hinzugefügt. Dadurch ändern sich die Distanzmarken der Nachbarn des hinzugefügten Knotens:

Wenn ein Weg, der über den zu V’ hinzugefügten Knoten führt, kürzer ist, als die bisherige Distanz, wird sie geändert. Dies kann der Fall sein, wenn der Knoten bisher nicht erreicht wurde und die Distanz also Unendlich betrug oder wenn ein anderer Weg, über den er vom Startpunkt aus erreicht werden konnte, länger als der neue Weg war.

Mit jedem Durchlauf der Schleife wird ein weiterer Knoten zu V’ hinzugefügt, bis zum Zielknoten oder bis zu allen Knoten der kürzeste Weg bestimmt wurde.

Um den Weg, auf dem ein Knoten für seine aktuelle Distanz erreicht wurde, zu speichern, genügt es, wenn jeder (erreichte) Knoten einen Verweis auf seinen Vorgänger behält. So baut der Dijkstra-Algorithmus über den Knoten aus V’ einen gerichteten Baum auf, dessen Wurzel der Startknoten s ist, von dem aus auf dem jeweils kürzesten Wegen alle anderen Knoten erreichbar sind.

2.3.4 Beweis der Korrektheit

Satz: Der Dijkstra-Algorithmus findet den kürzesten Weg von Knoten s nach t.

Vor dem Eintritt in die Schleife gilt offensichtlich:

(1) Für alle $v \in V'$ ist d(v) die Länge des kürzesten Weges zu v und p(v) gibt den Vorgänger von v auf diesem kürzesten Weg an.

(2) Für alle $v \in$ V\V' ist d(v) die minimale Länge eines Weges zu v, der nur über Knoten von V' führt. Existiert kein solcher Weg, ist $d(v) = \infty$.

Es soll gezeigt werden, dass diese beiden Eigenschaften in jedem Schritt der Schleife erhalten bleiben. Der Knoten $w \in$ V\V', für den d(w) minimal ist, wird zu V' hinzugefügt. (1) bleibt gültig, da w auf dem kürzest möglichen Weg erreicht wurde: Angenommen, es gäbe einen anderen kürzeren Weg nach w. Dieser Weg dürfte nicht nur über Knoten aus V' führen, denn nach (2) ist d(w) die Länge des kürzesten nur über Knoten aus V' führenden Weges. Es gäbe also einen ersten Knoten u auf dem Weg zu w mit $u \notin V'$. Da aber d(w) unter allen Knoten aus V\V' minimal ist, wäre $d(u) \geq d(w)$. Da keine negativen Kantengewichte zulässig sind, hat der Weg von u nach w eine Weglänge größer oder gleich null, sodass der Weg nach w über einen Knoten u nicht kürzer als der gefundene Weg nach w ist. (1) bleibt damit erfüllt.

(2) bleibt nach dem Hinzufügen von w zu V' gültig, da alle Distanzmarken der Nachbarn n von v aktualisiert werden, sodass d(n) geändert wird, wenn über w ein kürzerer Weg zu n führt als das bisherige d(n).

Da (1) und (2) bis zum Ende erhalten bleiben, folgt, dass wenn Zielknoten t Teil von V' ist, der kürzeste Weg von s nach t gefunden wurde.□

2.3.5 Komplexitätsbetrachtung

Satz: Die Laufzeit des Dijkstra-Algorithmus ist $O(n^2)$.

Beweis:

Die beschriebene Implementierung kann in drei Teile gegliedert werden:

1.: Die Initialisierung: Da jedem Knoten ein d(v) zugewiesen wird, werden O(n) Operationen ausgeführt.

In der (n-1)-mal wiederholten Schleife:

2.: Das Finden des Knoten w mit minimalem d(w) innerhalb V\V': Beim ersten Durchlauf werden n-1, beim zweiten n-2, beim dritten n-3 usw. Operationen benötigt, was insgesamt $\frac{(n-1)\cdot n}{2}$ Operationen erfordert.

3.: Das Aktualisieren von d(v) für jeden Nachbarn von w: Dieser Schritt wird in allen Schleifendurchläufen insgesamt für jede Kante einmal ausgeführt, also m mal.

Insgesamt ergibt sich also eine Komplexität von $O(n)+O(n^2)+O(m)= O(n^2+n+m)=O(n^2+m)$.

Da $m \leq n\cdot(n-1)$ (Im vollständigen Graphen gibt es von jedem Knoten zu jedem anderen Knoten einen Bogen) ist $O(m) \leq O(n^2)$, sodass sich für die Laufzeit des Dijkstra-Algorithmus $O(n^2)$ ergibt. □

2.3.6 Veranschaulichung des Algorithmus

Man kann sich die Vorgehensweise des Dijkstra-Algorithmus auch auf folgende Weise anschaulich machen: Wenn man sich einen Graphen als ein Netz aus Knoten, die mit Bindfäden verbunden sind und deren Länge dem Kantengewicht entsprechen, vorstellt, dann findet man die kürzesten Wege so:

Liegt das Netz des Graphen auf einem Tisch, so nimmt man nun den Startpunkt und hebt ihn langsam hoch. Der nächste Knoten, der sich von der Tischplatte hebt, ist der erste Knoten, zu dem ein endgültiger kürzester Weg gefunden wurde, der nächste ist der zweite, usw. bis der Zielknoten bzw. alle Knoten in der Luft hängen. Die kürzesten Wege, auf denen sie zu erreichen sind, sind die gespannten Bindfäden, durch die sie mit dem Startknoten verbunden sind.

2.4 Ein Algorithmus zum Zählen von Wegen

2.4.1 Lösungsidee

Alle Wege von einem Startknoten s zu einem Zielknoten t können mit einem auf Tiefensuche basierenden Algorithmus gefunden werden.

Dabei wird einem Weg solange gefolgt, bis er in eine Sackgasse geführt hat oder das Ziel erreicht ist. Dann wird zum letzten Knoten zurückgekehrt und ein anderer Weg versucht. Über die schon bei Dijkstra verwendeten Vorgängermarken wird so ein immer wieder veränderter Baum aufgebaut, dessen Wurzel der Startknoten ist. Schon abgelaufene Knoten werden als besucht markiert.

2.4.2 Algorithmus: Suche Alle Wege von s nach t

Input: D=(V,A); $s,t \in V$

Datenstrukturen:

$m: V \rightarrow \{\text{true, false}\}$ (Besuchsmarkierung)

$p: V \rightarrow V$ (Vorgänger in der aktuellen Baumstruktur)

wegzahl: $\mathbb{R}_+$ (Anzahl der bisher gefundenen Wege)

Output:

sucheAlleWege (s, t: Knoten)

//Initialisierung

Für alle $v \in V$:

  m(v):=false

  wegzahl:=0

m(s):=true; //markiere den Startknoten

sucheAlleWege2(s,t)

end

sucheAlleWege2(s,t): Boolean;

Lokale Variablen:

  ergebnis: Boolean

  teilergebnis: Boolean

ergebnis:=false

Wenn s=t:

  Weg nach t ausgeben

  Inc(wegzahl)

  Ergebnis:=true

Sonst:

  Für alle unmarkierten Nachbarn n von s:

    m(n):=true //Markiere n

    p(n):=s;

    teilergebnis:=sucheAlleWege2(n,t);

    Wenn teilergebnis=true

      ergebnis:=true

  Wenn ergebnis=true

    demarkiere s und alle Nachfahren von s

return ergebnis

2.4.3 Funktionsbeschreibung

Der Algorithmus sucheAlleWege wird zunächst mit dem markierten Startknoten s und dem Zielknoten t aufgerufen. Rekursiv ruft dieser dann von einem Nachbarn von s, der als besucht markiert wird, wieder sucheAlleWege auf. Diese Rekursion kann auf zwei Arten beendet werden:

1.: Der neue Ausgangspunkt der Suche hat keine unmarkierten Nachbarn mehr, von denen aus sucheAlleWege neu aufgerufen werden könnte, der aktuell verfolgte Weg ist also eine Sackgasse.

2.: Der Zielknoten ist erreicht.

Im ersten Fall wird als Ergebnis der gerade aktuellen aufgerufenen Methode sucheAlleWege false zurückgegeben. Durch Backtracking wird zur zuletzt aufgerufenen Methode sucheAlleWege2 zurückgekehrt (bzw. zum letzten besuchten Knoten) und von einem anderen Nachbarn aus versucht, einen Weg zum Ziel zu finden. Der erfolglos besuchte Knoten bleibt (als schon besucht) markiert. Führen auch keine Wege über alle anderen Nachbarknoten zum Ziel, liefert auch dieser Aufruf von sucheAlleWege false zurück und es wird weiter zurückgegangen.

Im zweiten Fall wurde ein Weg zum Ziel gefunden. Dieser Weg kann gezählt und durch die Vorgängermarken jedes Knotens nachvollzogen und z. B. ausgegeben werden. Es ist nun von Interesse, ob es noch einen Weg zum Ziel mit einem gleichen Anfangsstück wie der gefundene Weg gibt, aber einem anderen Ende. Durch Backtracking geht der Algorithmus wieder zum zuletzt besuchten Knoten zurück, entfernt aber alle Markierungen, die am letzten Knoten und an von ihm ausgehenden Teilbäumen entstanden, da diese Knoten nicht mehr Bestandteil des aktuellen Baums sein sollen, sondern auf anderen Wegen wieder besucht werden können.

So liefert jeder Aufruf von sucheAlleWege entweder das Ergebnis false, was bedeutet, dass der vom Startknoten dieses Aufrufs ausgehende Teilbaum keinen Weg zum Ziel enthält, oder true, was bedeutet, dass der Startknoten auf mindestens einem Weg zum Ziel liegt und diese Wege alle gefunden wurden.

2.4.4 Beweis der Korrektheit

Satz: Der Algorithmus sucheAlleWege findet alle Wege von s nach t.

Beweis:

Angenommen es gäbe einen Weg von s nach t, den der Algorithmus nicht findet. Dann existiert ein maximales Anfangsstück $v_1,...,v_k$ ($k \geq 0$, wobei k=0 bedeutet, dass kein Anfangsstück gefunden wurde) von diesem Weg, das noch Bestandteil des Suchbaums zu einem Zeitpunkt des Algorithmusablaufs gewesen ist, v_{k+1}, von dem ein Weg zu t führt, fehlte jedoch.

Da bei dem Aufruf von sucheAlleWege, bei dem v_k Startknoten ist, alle unmarkierten Nachbarn besucht werden, muss v_{k+1} markiert worden sein, um nicht erreicht zu werden. Dafür muss v_{k+1} aber von der Suche erreicht worden sein.

Somit kann es keinen s-t-Weg geben, den der Algorithmus nicht findet.□

3 Wegeprobleme als ganzzahliges Programm

In den vorigen Kapiteln wurden Grundlagen der Graphentheorie und zwei Wegeprobleme vorgestellt. Sowohl zum Finden des kürzesten Weges als auch zum Zählen aller Wege wurde ein Algorithmus gefunden, der das Problem löst. Beide Algorithmen basieren auf dem Konzept des Graphen und trennen sich nicht von dieser Vorstellung des Problems, um es zu lösen.

Man kann graphentheoretische Probleme aber auch mit Methoden anderer Bereiche lösen: Wie man den kürzesten Weg und alle möglichen Wege zwischen zwei Knoten mit ganzzahliger Programmierung bestimmen kann, soll in diesem Kapitel verdeutlicht werden.

Dazu wird zunächst erklärt, was ein ganzzahliges Programm ist und dann vorgestellt, wie man die Wegeprobleme aus Kapitel 2 auf diese Weise formulieren und lösen kann.

3.1 Einführung

Die Probleme, mit denen sich die ganzzahlige Programmierung (englisch: Integer Programming, kurz: IP) beschäftigt, sind verwandt mit denen der linearen Optimierung (oder linearen Programmierung). Es geht darum, eine Funktion, die von mehreren Variablen abhängt, zu minimieren oder zu maximieren. Dabei haben die Variablen einem System von Bedingungen in Form von Gleichungen und Ungleichungen zu genügen. Alle Belegungen der Variablen, die die Bedingungen erfüllen, werden mögliche Lösungen des IPs genannt. Von ihnen ist die optimale gesucht. Im Gegensatz zur linearen Optimierung dürfen die Variablen bei einem ganzzahligen Programm nur ganzzahlige Werte annehmen. In dieser Arbeit sollen sogar nur solche IPs betrachtet werden, deren Variablen auf die Werte 0 und 1 beschränkt sind. Diese Form des IPs wird binäres Programm oder 0/1 IP genannt und reicht zur Modellierung der Wegeprobleme aus.

3.2 Definition eines binären Programms (0/1 IP)

Ein binäres Programm besteht aus zwei Teilen: Einer Funktion, die minimiert oder maximiert werden soll und einem System von Ungleichungen, das die Variablen des 0/1 IPs erfüllen müssen.

Ein binäres Programm, bei dem etwas minimiert werden soll, hat die Form

$\min\{cx : Ax \leq b, x \in B^n\}$.

Dabei stellt der *n-dimensionale Vektor x* die Variablen dar. Er ist aus der *Menge B^n von n-dimensionalen binären Vektoren*, d. h. die Komponenten von x, bzw. die Variablen, haben jeweils den Wert 0 oder 1.

A ist eine m×n-Matrix und *b ein m-dimensionaler Vektor*, sodass Ax ≤ b ein System von m Ungleichungen darstellt, dem die n Variablen, also der Vektor x, genügen soll. Auch Gleichungen können in der Form der Ungleichung beschrieben werden, indem eine Gleichung durch zwei Ungleichungen ersetzt wird, z.B.
$3x = 2$ durch $3x \leq 2 \wedge 3x \geq 2 \Leftrightarrow 3x \leq 2 \wedge -3x \leq -2$.

c ist ein n-dimensionaler Vektor, sodass f(x)=cx die Funktion darstellt, die optimiert werden soll, was in diesem Fall bedeutet, dass die Besetzung für x gesucht ist, bei der cx den minimalen Wert annimmt.

Alle x, die das Ungleichungssystem Ax ≤ b erfüllen, werden mögliche Lösungen des 0/1 IPs genannt. Die optimale Lösung ist die Lösung, für die f(x) minimal ist.

3.3 Modellierung von Wegeproblemen mit ganzzahliger Programmierung

3.3.1 Ein Weg als Lösung eines 0/1 IPs

Im Folgenden wird von einem Digraphen ohne Mehrfachkanten ausgegangen, die Formulierungen lassen sich aber auch auf ungerichtete Graphen übertragen.

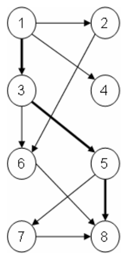
Die Menge A der Bögen ist Teilmenge von V×V, sodass (u,v) den Bogen von Knoten u nach v bezeichnet. Wie in 1.2.1 definiert, gibt es außerdem eine Gewichtsfunktion c: A → $\mathbb{R}$, die auch als Länge des Bogens interpretiert werden kann.

Neu eingeführt wird die Funktion x: A → {0,1}, die jedem Bogen eine binäre Variable zuordnet. Der Wert x(a) kennzeichnet, ob ein Weg durch den Bogen a führt (x(a)=1) oder nicht (x(a)=0).

Ein Vektor x, dessen Dimension der Anzahl der Bögen des Graphen entspricht, stellt dar, welchem Bogen die Funktion x den Wert 0 und welchem sie den Wert 1 zuordnet, bzw. durch welche Bögen ein Weg führt.

Damit ist es möglich, nur durch den Vektor x einen Weg in einem Graphen zu beschreiben.

Im Folgenden wird der Einfachheit halber das Gewicht des Bogens (z. B. interpretiert als Länge) mit c_{uv} statt mit c((u,v)) bezeichnet und statt x((u,v)) x_{uv} geschrieben. [3]

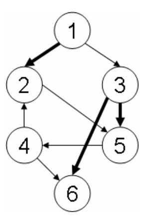


In der Abbildung entspricht der dick markierte Weg bei $x = (x_{12}, x_{13}, x_{14}, x_{26}, x_{35}, x_{36} x_{57}, x_{58}, x_{68}, x_{78})$ dem Vektor $x = (0,1,0,0,1,0,0,1,0,0)$.

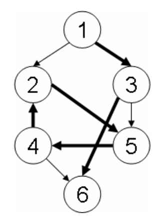
Weg in einem gerichteten Graphen

3.3.2 Die Bedingungen für einen Weg als Gleichungssystem

Jeder Weg in einem Graphen lässt sich durch einen Vektor x wie oben gezeigt darstellen, aber nicht jeder m-dimensionaler Vektor x stellt tatsächlich einen Weg dar:



Mit $x = (x_{12}, x_{13}, x_{25}, x_{35}, x_{36}, x_{42}, x_{46}, x_{54})$ entsprechen die links dick markierten Bögen x=(1,0,0,1,1,0,0,0) und die rechts markierten Bögen x=(0,1,1,0,1,1,0,1). Beides sind keine Wege.



Unzusammenhängende Wegstücke

Kreislösung eines IPs

Um ein Weg von Knoten s zu Knoten t zu sein, muss x (bzw. müssen die Komponenten von x) bestimmte Bedingungen erfüllen:

Folgende Bezeichnungen werden dabei verwendet: (vergleiche Kapitel 1.2.2)

$\delta^{+}(v) \subseteq A$: Menge der ausgehenden Bögen von Knoten v,

$\delta^{-}(v) \subseteq A$: Menge der eingehenden Bögen in Knoten v,

für $B \subseteq A$: $x(B) := \sum_{(u,v)\in B} x_{uv}$,

[3] x_{uv} bzw. c_{uv} bedeuten jedoch nicht, dass es für jede Kombination $i, j \in V$ ein c_{uv} und ein x_{uv} gibt, sondern nur für alle $(i, j) \in A$, also nur für alle tatsächlich existierenden Bögen.

s: Startknoten des Weges

t: Zielknoten des Weges

Damit x ein Weg von s nach t ist, muss gelten:

(1) $x(\delta^{+}(s)) = 1,\ x(\delta^{-}(s)) = 0$

(2) $x(\delta^{-}(t)) = 1, x(\delta^{+}(t)) = 0$

(3) $\underset{\substack{v \in V \\ v \neq s,t}}{\forall} x(\delta^{-}(v)) = x(\delta^{+}(v))$

(4) $\underset{v \in V}{\forall} x(\delta^{-}(v)) \leq 1$
$\underset{v \in V}{\forall} x(\delta^{+}(v)) \leq 1$

(5) $\underset{\substack{W \subseteq V \\ W \neq \{\} \\ s,t \notin W}}{\forall} x(A(W)) \leq |W| - 1$

$x_{uv} \in \{0,1\}$

Von s soll genau ein Weg ausgehen, sodass genau einer der Ausgangsbögen von s den Wert 1 annehmen soll, aber keiner der in s eingehenden Bögen (1). Entsprechend soll am Ziel t genau ein Weg ankommen, sodass genau einer der in t ankommenden Bögen den Wert 1 haben soll, aber keiner der Ausgangsbögen (2).

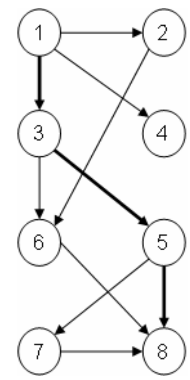
Für alle anderen Knoten muss gelten, dass, falls der Weg durch sie führt, es sowohl einen Eingangs- als auch einen Ausgangsbogen mit dem Wert 1 gibt und wenn der Knoten nicht auf dem Weg liegt, dass sowohl die Eingangs- als auch die Ausgangsbögen alle den Wert 0 zugeordnet bekommen (3). Für die Wegeprobleme sollen die durch x beschriebenen Wege durch keinen Knoten doppelt führen, deshalb darf nie mehr als ein Eingangs- und ein Ausgangsbogen an einem Knoten auf dem Weg liegen (4).

Zuletzt muss ausgeschlossen werden, dass es neben dem Weg von s nach t noch weitere, unabhängige Kreise gibt, wie z.B. bei dem rechten Beispiel am Anfang des Kapitels. (5) besagt deshalb, dass in keiner Teilmenge von Knoten die Anzahl der Bögen, durch die ein Weg führt, gleich der Anzahl der Knoten der Teilmenge ist. Im Fall von Kreisen ist dies für die Menge der auf dem Kreis liegenden Knoten jedoch der Fall, daher werden die Kreislösungen durch (5) ungültig. Das Ausschließen der Kreise nennt man „Subtour Elimination“.

3.3.3 Beispiel

Der Graph in der Abbildung ist durch Folgendes beschrieben:

$$V = \{1,2,3,4,5,6,7,8\}$$
$$A = \{(1,2),(1,3),(1,4),(2,6),(3,5),(3,6),(5,7),(5,8),(6,8),(7,8)\}$$ [4]
$$\underset{(u,v)\in A}{\forall} c_{uv} = 1$$



Weg in einem gerichteten Graphen

Der fett markierte Weg von 1 bis 8 wird durch den Vektor

x= $(x_{12}, x_{13}, x_{14}, x_{26}, x_{35}, x_{36} x_{57}, x_{58}, x_{68}, x_{78}) = (0,1,0,0,1,0,0,1,0,0)$

dargestellt.

Jeder Fluss von 1 nach 8 muss nun diese Bedingungen erfüllen:

(1) $x_{12} + x_{13} + x_{14} = 1$

(2) $x_{58} + x_{68} + x_{78} = 1$

(3) und (4) $x_{12} = x_{26} \leq 1$

$$x_{13} = x_{35} + x_{36} \leq 1$$
$$x_{14} = 0 \leq 1$$
$$x_{26} + x_{36} = x_{68} \leq 1$$
$$x_{35} = x_{57} + x_{58} \leq 1$$
$$x_{36} + x_{26} = x_{68} \leq 1$$
$$x_{57} = x_{78} \leq 1$$

(5) wird hier aus Platzgründen nicht aufgeführt.

3.3.4 Das Kürzeste-Wege-Problem als IP

Nach dem oben dargestellten Prinzip können die Bedingungen eines ganzzahligen Programms erhalten werden.

Die zu optimierende Funktion des IPs hängt direkt von der Problemstellung ab. Bei der Suche nach dem kürzesten Weg soll $\sum_{(u,v)\in A} x_{uv} c_{uv}$, was der Weglänge entspricht, minimal sein.

---

[4] c_{uv} bezeichnet die Kantengewichte, bzw. Längen, die in diesem Fall alle 1 betragen.

Das IP lautet damit:

$$\min \sum_{(u,v)\in A} x_{uv} c_{uv}$$

(1) $x(\delta^{+}(s)) = 1$, $x(\delta^{-}(s)) = 0$

(2) $x(\delta^{-}(t)) = 1$, $x(\delta^{+}(t)) = 0$

(3) $\underset{\substack{v\in V\\ v\neq s,t}}{\forall}\ x(\delta^{-}(v)) = x(\delta^{+}(v))$

(4) $\underset{v\in V}{\forall}\ x(\delta^{-}(v)) \le 1$
$\underset{v\in V}{\forall}\ x(\delta^{+}(v)) \le 1$

(5) $\underset{\substack{W\subseteq V\\ W\neq\{\}\\ s,t\notin W}}{\forall}\ x(A(W)) \le |W| - 1$

$x_{uv} \in \{0,1\}$

Dabei ist die Bedingung (5) unpraktisch, da für jede Teilmenge von V eine Ungleichung aufgestellt wird, die Anzahl der möglichen Teilmengen aber mit der Anzahl der Knoten exponentiell wächst. Lässt man (5) weg, kommen zu den möglichen Lösungen solche mit zusätzlichen Kreisen hinzu.

Wenn man den kürzesten Weg in einem Graphen mit positiven Gewichten (c>0) sucht, kann man diese Lösungen aber zulassen, da die Summe der Kantenlängen des Kreises größer als null sein wird und daher keine Kreislösung optimaler ist, als die zugehörige Lösung, die den gleichen Weg, aber ohne zusätzlichen Kreis darstellt.

3.3.5 Wege zählen mit ganzzahliger Programmierung

Wenn man alle Wege zwischen Startknoten s und Endknoten t sucht, dann müssen alle möglichen Lösungen des IPs gefunden werden. Im Gegensatz zum Optimieren beim Problem des kürzesten Weges mit positiven Kantengewichten, ist das Weglassen von Bedingung (5) bei diesem Problem nicht möglich.

Da mit Bedingung (5) $2^{|V|-2}-1$ Teilmengen von V untersucht werden müssen, wird das IP schnell sehr groß. Daher sind die Anwendungsmöglichkeiten von IPs zum Zählen von Wegen deutlich mehr beschränkt als die Möglichkeiten, IPs zur Wegoptimierung zu nutzen.

3.3.6 Geometrische Vorstellung

Um ein IP zu lösen, verwendet man ein einfacheres Problem, nämlich ein lineares Programm (LP), was bedeutet, dass man statt $x_{uv} \in \{0,1\}$ „nur“ $0 \leq x_{uv} \leq 1$ fordert. Dies nennt man „LP-Relaxierung“.

Die Lösungsmenge eines LPs lässt sich als Polyeder[5] in einem Raum mit so vielen Dimensionen, wie das LP Variablen hat, auffassen. Die Bedingung $0 \leq x_{uv} \leq 1$ für alle x_{uv} beschränkt die Lösungsmenge durch einen Hyperwürfel[5], sodass es sich bei der Lösungsmenge des linearen Programms um ein Polytop[5] handelt.

Kann man eine lineare Zielfunktion über einem Polytop minimieren oder maximieren, ist immer eine der Ecklösungen optimal. Im Fall des Kürzesten-Wege-Problems mit c>0 (positiven Kantengewichten) sind die Ecken des Polytops der LP-Relaxierung genau die ganzzahligen Lösungen des Ausgangs-IP. Das heißt, man kann die IP-Formulierung des Kürzesten-Wege-Problems lösen, indem man die zugehörige LP-Relaxierung löst.

3.3.7 IPs lösen

Man führt IPs zum Lösen auf lineare Optimierungsprobleme zurück, für die es diverse Lösungsmethoden gibt. Das Problem des kürzesten Weges bei positiven Kantengewichten ist als lineares Optimierungsproblem gut lösbar.

Es gibt auch Programme, die die Ecken eines durch ein IP beschriebenen Polytops zählen. Sie wären gut geeignet, die Lösungen eines IPs ohne „Subtour Elimination“ (Elimination der vom Weg getrennten Kreise) zu finden und zu zählen. Mit „Subtour Elimination“ sind sie aber nur für kleine Graphen anwendbar.

[5] Ein Polyeder ist eine Schnittmenge von Halbräumen, einen beschränkten Polyeder nennt man Polytop. Mit Hyperwürfel wird ein Würfel im mehrdimensionalen Raum bezeichnet.

4 Statt Lösungen von Gleichungssystemen Wege zählen

4.1 Einleitung

Nachdem nun beschrieben wurde, wie mit den Methoden der ganzzahligen Programmierung kürzeste Wege und alle Wege in Graphen gefunden werden können, soll nun zum Abschluss des theoretischen Teils noch ein anderer Aspekt des Zusammenhangs zwischen ganzzahliger Programmierung und graphentheoretischen Wegeproblemen erläutert werden.

Während bisher immer von dem Wegeproblem im Graphen ausgegangen wurde und dafür unterschiedliche Lösungsmethoden gesucht wurden, wird im diesem Kapitel der umgekehrte Weg thematisiert: Ein allgemeines Problem, nämlich die Lösungen beliebiger 0/1 IPs zu finden und zu zählen, kann man auf das Problem zurückführen, in einem Graphen Wege zu zählen.

Prinzipiell wird dabei so vorgegangen, dass aus dem Ungleichungssystem des 0/1 IPs ein spezieller Graph konstruiert wird, in dem alle Lösungen als Wege dargestellt werden. Kann man diese Wege finden und zählen, hat man die Lösungen des 0/1 IPs erhalten.

4.2 Binäre Entscheidungsdiagramme (BDDs)

Ein binäres Entscheidungsdiagramm, kurz BDD (englisch: Binary Decision Diagram) ist ein gerichteter Graph, der Belegungen von Binärvariablen und damit die Lösungen eines binären ganzzahligen Programms darstellt.

Der Graph hat zwei besondere Knoten, von denen einer mit „1“ bezeichnet wird und der andere mit „0“. Die Knoten des BDDs sind, wenn es sortiert ist, wovon ausgegangen wird[6], auf Ebenen angeordnet, sodass ein Knoten in der obersten Ebene liegt und die beiden Knoten 1 und 0 in der untersten. Bis auf die unterste Ebene steht jede Ebene für eine Variable des Ungleichungssystems, für das die möglichen Lösungen gesucht sind. Alle Knoten des Graphen sind auf diese Ebenen verteilt, ihnen lässt sich also jeweils eine Variable zuordnen. Von jedem Knoten gehen genau zwei Bögen aus, von denen der eine

[6] Sortierte binäre Entscheidungsdiagramme werden in der Literatur auch mit OBDD für „Ordered Binary Decision Diagram“ abgekürzt.

für die Besetzung „0“ der dem Knoten zugeordneten Variable steht, der andere für die Besetzung „1“ und die in eine tiefere Ebene führen.

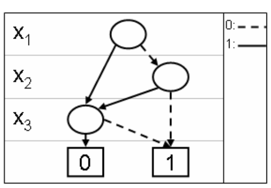
Jeder Weg von der obersten Ebene zur untersten stellt eine oder mehrere Belegungen aller Variablen des Ungleichungssystems dar. Je nachdem, entlang welchem Ausgangsbogen eines Knoten der Weg führt, ist die dem Knoten zugeordnete Variable mit eins oder null belegt.

Alle Wege von ganz oben nach ganz unten stellen somit alle Belegungen für die Variablen dar, wobei es allerdings möglich ist, das mehrere Belegungen durch den gleichen Weg dargestellt werden, da auch Ebenen des Baums von dem Weg „übersprungen“ werden können. (Siehe Beispiel unten)

Von all diesen Wegen stellen alle, die zum Knoten „1“ führen, Belegungen von Variablen dar, die das Ungleichungssystem erfüllen und alle zum Knoten „0“ sind entsprechend ungültige Einsetzungen.

4.2.1 Beispiel

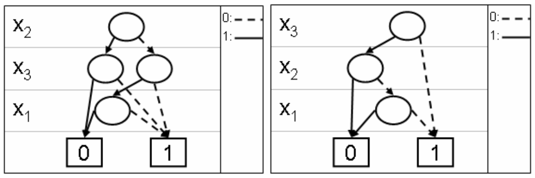
Ein Beispiel eines kleinen BDDs, das die Lösungen der Ungleichung $x_1 + 2x_2 + 3x_3 \leq 3$ veranschaulicht, ist in der Abbildung dargestellt. Es wird hier kein expliziter Algorithmus zum Finden solcher BDDs angegeben, dazu sei auf Behle und Eisenbrand [10] verwiesen. Hier soll lediglich das Konzept des BDDs veranschaulicht und auf sich ergebende Probleme aufmerksam gemacht werden.



Beispiel für ein BDD

Dieses BDD stellt die Lösungen für $x_1 + 2x_2 + 3x_3 \leq 3$ dar. Die einzigen gültigen Einsetzungen für (x_1,x_2,x_3) sind demnach (0,0,0), (0,0,1), (0,1,0), (1,0,0) und (1,1,0).

Doch ist das oben abgebildete BDD nicht das einzig mögliche BDD, denn auch diese beiden (unter anderen) stellen die Lösungen der Ungleichung richtig dar:



Verschiedene BDDs für die gleiche Ungleichung

Man sieht, dass die Wahl der Variablenreihenfolge einen Effekt auf die Struktur des BDDs hat und teilweise auch unterschiedlich komplizierte BDDs möglich sind.

4.2.2 Lösen eines 0/1 IPs mit BDDs

$Ax \leq b$ mit $A \in \Re^{n \times m}$, $x \in \Re^{n}$ und $b \in \Re^{m}$, wobei n die Anzahl der Variablen und m die Anzahl der Bedingungen ist, seien die Bedingungen eines binären Programms (0/1 IPs).

Für jede Ungleichung, also für jede Zeile der Matrix, lässt sich ein BDD formulieren, in dem die Wege zum Knoten 1 die gültigen Einsetzungen darstellen.

Indem diese m BDDs nun über einen Algorithmus (siehe Behle und Eisenbrand [10]) in mehreren Schritten jeweils paarweise zu einem großen BDD „verschmolzen" werden, erhält man einen Graphen, der alle Lösungen des ursprünglichen Ungleichungssystems als Wege darstellt.

Mit dem in 2.4 geschilderten Algorithmus kann man diese Lösungen einfach zählen und so mit Tiefensuche im Graphen die Lösungen eines 0/1 IPs bestimmen.

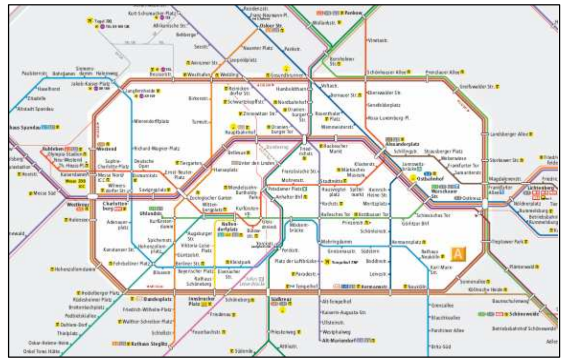
Ein Programm, das diese Methode verwendet, ist zum Beispiel Azove (Another Zero One Vertex Enumeration tool).

Ein Problem dieses Ansatzes ist, dass die BDDs zu groß werden können. Damit hat man bei zu vielen Variablen ein Speicherproblem. Falls man aber genügend Speicher hat, ist das Verfahren, um in BDDs Lösungen zu zählen, sehr effizient. (Siehe Behle und Eisenbrand [10] und Behle [9])

5 Implementierung: Wege im S- und U-Bahnnetz von Berlin

5.1 Der Graph

Das S- und U-Bahnnetz, das implementiert wurde, ist aus dem Jahr 1997 und beinhaltet 307 Bahnhöfe, die die Knoten des Graphen bilden und 445 Streckenteile, die jeweils zwei Bahnhöfe verbinden und als zwei entgegengesetzte Bögen im Graphen dargestellt werden.



Das S- und U-Bahnnetz von Berlin aus [13]

Alle Kanten haben das Gewicht 1.Wenn zwischen zwei Bahnhöfen mehrere Linien fahren, wurde der Streckenteil dennoch nur als ein Kante aufgefasst, um die möglichen verschiedenen Wege beim Zählen einzuschränken.

Der Graph des S- und U-Bahnnetzes wurde ansonsten nicht vereinfacht.[7]

5.2 Routenplaner

Der hier implementierte Dijkstra-Algorithmus sucht den kürzesten Weg zwischen zwei Bahnhöfen. Ein Programm der BVG, das Routen von einem Start- zu einem Zielbahnhof ausrechnet, findet man im Internet unter http://www.fahrinfo-berlin.de. Die Umsetzung dort ist noch etwas komplizierter, da auch die Fahrpläne, Umsteige- und Fahrzeiten beachtet werden. Dadurch bekommt das Problem noch eine zeitliche Komponente, grundsätzlich basieren solche Routenplaner aber auch auf Algorithmen, wie dem implementierten Dijkstra-Algorithmus.

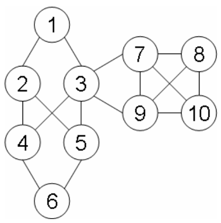
5.3 Aufgaben dieses Programms

Auf dem Graphen des S- und U-Bahnnetzes wurden zwei Methoden implementiert:

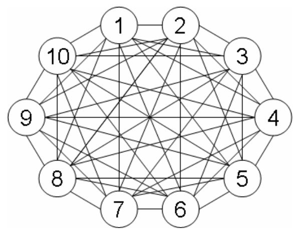
[7] Für manche Anwendungen ist es sinnvoll, z. B. alle Bahnhöfe, an denen nicht umgestiegen werden kann, zu entfernen. Ein solches Vereinfachen des Graphen, bevor der eigentliche Algorithmus angewendet wird, nennt man "Preprocessing" und kann die Zeit, die der Algorithmus zum Lösen eines Problems braucht, erheblich verringern.

Der Dijkstra-Algorithmus, der zwischen zwei Bahnhöfen den kürzesten Weg bestimmen soll und ein Algorithmus, der auf dem Prinzip der Tiefensuche aufbauend alle Wege zwischen zwei Bahnhöfen finden und zählen soll.

Da die Anzahl der Wege in einem so großen Graphen wie dem S- und U-Bahnnetz nicht zu überprüfen ist, soll dieser Algorithmus noch an zwei kleineren Graphen zum Testen ausprobiert werden:



Kleiner Testgraph



Vollständiger Graph

Der Graph in der linken Abbildung ist ein recht einfacher Graph, an dem die Zahl der Wege von zum Beispiel Knoten 1 zu Knoten 6 leicht abgezählt werden kann. (Es gibt acht Wege.)

Der zweite Graph ist ein vollständiger Graph mit zehn Knoten, d.h. jeder Knoten ist mit jedem anderem durch eine Kante verbunden. Durch diese Struktur ist trotz der großen Anzahl von Wegen diese leicht zu berechnen:

[Anzahl der Wege]=[Anzahl der Wege mit 10 Knoten]+[Anzahl der Wege über 9 Knoten]+...+ [Anzahl der Wege mit 2 Knoten]

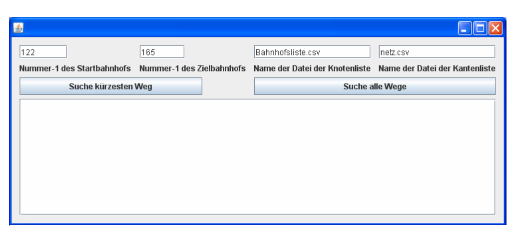
$$=\binom{8}{8}+\binom{7}{8}+\binom{6}{8}+\binom{5}{8}+\binom{4}{8}+\binom{3}{8}+\binom{2}{8}+\binom{1}{8}+\binom{0}{8}$$

$$=\frac{8!}{0!}+\frac{8!}{1!}+\frac{8!}{2!}+\frac{8!}{3!}+\frac{8!}{4!}+\frac{8!}{5!}+\frac{8!}{6!}+\frac{8!}{7!}+\frac{8!}{8!}$$

=40320+40320+20160+6720+1680+336+56+8+1

=109601

Die Bedienung des Programms erfolgt über eine einfache grafische Oberfläche, wo auch der kürzeste Weg ausgegeben wird. Beim Suchen aller Wege wird nur die Anzahl der Wege ausgegeben und die ersten hundert gefundenen Wege in eine Textdatei geschrieben.



Screenshot des implementierten Programms

5.4 Darstellung des Graphen

Graphen lassen sich als Matrizen oder Listen darstellen und die Kanten explizit als Kanten (bzw. Bögen) oder implizit als Nachbarschaft (Adjazenz) zweier Knoten speichern.

In dieser Implementierung enthält der Graph eine Knotenliste und es gibt zusätzlich zur Klasse Graph eine Klasse Knoten und eine Klasse Kante. In der Knotenliste sind die Knoten gespeichert und jeder Knoten enthält eine Kantenliste. Eine Kante (korrekter: Bogen) enthält neben Informationen über Gewicht und Name nur einen Verweis auf den Zielknoten, da sie im Startknoten in der Kantenliste gespeichert ist.

Eine solche Datenstruktur zum Speichern des Graphen nennt man Inzidenzliste.

5.5 Rechnung

Die in Kapitel 2 beschriebenen Algorithmen wurden in Java (Version 1.6.0) implementiert und auf einem Windows XP (Version 2002) mit einer CPU AMD Sempron(tm) 2200+ mit 1,51 GHz und 1,43 GB RAM getestet.

Einige exemplarische Ergebnisse des Zählens von Wegen und des Dijkstra-Algorithmus sind in den Tabellen aufgeführt:

Startbahnhof	Zielbahnhof	Anzahl Wege	Rechenzeit
Heerstr.	Dahlem-Dorf	106.805.501	3721,4s
Zoologischer Garten	Nollendorfplatz	37.449.495	450,3s
Nollendorfplatz	Zoologischer Garten	37.449.495	616,7s
Heerstr.	Nollendorfplatz	129.684.481	1761,6s
Heerstr.	Eichkamp	44.328.964	1814,9s
Heerstr.	Ostkreuz	111.869.616	2481,0s
Dahlem-Dorf	Ostkreuz	94.130.989	2271,7s

Startbahnhof	Zielbahnhof	Kürzester Weg	Rechenzeit
Heerstr.	Dahlem-Dorf	Heerstr. ; Eichkamp ; Westkreuz ; Halensee ; Hohenzollerndamm ; Heidelberger Platz ; R"udesheimer Platz ; Breitenbachplatz ; Podbielskiallee ; Dahlem-Dorf ;	0,0s
Zoologischer Garten	Nollendorfplatz	Zoologischer Garten ; Wittenbergplatz ; Nollendorfplatz ;	0,0s
Heerstr.	Nollendorfplatz	Heerstr. ; Eichkamp ; Westkreuz ; Charlottenburg ; Savignyplatz ; Zoologischer Garten ; Wittenbergplatz ; Nollendorfplatz ;	0,0s
Heerstr.	Eichkamp	Heerstr. ; Eichkamp ;	0,0s

Ergebnis im kleinen Graphen:

Startknoten	Zielknoten	Anzahl Wege	Rechenzeit
1	6	8	0,0s

Ergebnis im vollständigen Graphen:

Startknoten	Zielknoten	Anzahl Wege	Rechenzeit
1	10	109601	0,2s

5.6 Ergebnis

Die Experimente ergaben, dass der Dijkstra-Algorithmus einwandfrei und schnell funktioniert. Auch das Wegezählen funktioniert, dauert aber wegen der großen Anzahl von möglichen Wegen in einem großen Graphen deutlich länger.

Es überrascht vielleicht, dass es so viele Wege zwischen zwei Bahnhöfen im S- und U-Bahnnetz gibt. Es gibt eine Formel, mit der man für zufällig generierte Graphen mit einer gewissen „Kantenwahrscheinlichkeit“, die die Menge der Kanten des Graphen bestimmt, die Anzahl der Wege zwischen zwei Knoten schätzen kann. Mit ihr erhält man für einen Graphen mit 307 Knoten und 445 Kanten eine Anzahl von 229.144.026 Wegen zwischen zwei Knoten, was immerhin in der gleichen Größenordung wie die tatsächlichen Werte liegt. (Siehe Borndörfer, Grötschel und Löbel [8], S. 10)

Außerdem fällt auf, dass die Rechenzeit teilweise selbst bei der gleichen Anzahl gefundener Wege sehr unterschiedlich ist. Dies ist dadurch zu erklären, dass die Tiefensuche, wenn sie in viele Sackgassen läuft oder mehrere Sackgassen mehrmals besucht, deutlich länger brauchen kann.

6 Abschließender Vergleich

6.1 Lösung der Wegeprobleme als IP-Programm

Wie in Kapitel 3 besprochen, kann man das Kürzeste-Wege-Problem und auch das Zählen aller Wege mit Methoden der ganzzahligen Programmierung lösen.

Daher sollen nun zum Abschluss die selbst implementierten, auf dem Graphenkonzept aufbauenden Algorithmen mit Programmen zum Lösen von IPs verglichen werden. Die gleichen Aufgaben wie in Kapitel 5 sollen auf den zum IP umgeformten Graphen getestet werden.

6.1.1 Umwandlung des Graphen in ein 0/1 IP

Zum Erstellen eines 0/1 IPs ist die Kantenliste des Graphen sowie die Angabe eines Start- und Zielknotens erforderlich. Automatisch kann dann ein Gleichungssystem erstellt werden, das für jeden Knoten die Gleichung [Anzahl der auf dem Weg liegenden Ausgangsbögen]=[Anzahl der auf dem Weg liegenden Eingangsbögen] aufstellt und für den Startknoten [Anzahl der auf dem Weg liegenden Ausgangsbögen]=1 sowie für den Zielknoten [Anzahl der auf dem Weg liegenden Eingangsbögen]=1.

Ein Programm, das ein mathematisches Model in ein IP bzw. in das zugehörige lineare Programm umwandelt, ist z.B. das vom Konrad-Zuse-Zentrum (ZIB) entwickelte Programm Zimpl. (Siehe [11])

6.1.2 Der kürzeste Weg

Um das erzeugte lineare Programm zu lösen, lässt sich zum Beispiel das ebenfalls vom ZIB entwickelte Programm SCIP (Solving Constraint Integer Programs) benutzen. (Siehe [12])

Es liest das lineare Programm ein und gibt die optimale Variablenbelegung aus.

Dies kann dann für den Weg von Heerstraße nach Dahlem-Dorf so aussehen:

```
SCIP> display solution

objective value:                                    9
x$Westkreuz$Halensee                                1   (obj:1)
x$Halensee$Hohenzollerndamm                         1   (obj:1)
x$Hohenzollerndamm$HeidelbergerPlatz                    1         (obj:1)
x$HeidelbergerPlatz$RudesheimerPlatz                    1         (obj:1)
x$Heerstr$Eichkamp                                  1   (obj:1)
x$Eichkamp$Westkreuz                                1   (obj:1)
x$Podbielskiallee$Dahlem_Dorf@181                    1  (obj:1)
x$Breitenbachplatz$Podbielskiallee                    1           (obj:1)
x$RudesheimerPlatz$Breitenbachplatz                    1          (obj:1)
```

Ausgabe von SCIP für den kürzesten Weg von Heerstr. nach Dahlem-Dorf

Sortiert man die Kanten in die richtige Reihenfolge, erhält man den gleichen Weg, den der Dijkstra-Algorithmus fand.

6.1.3 Zählen von Wegen

Komplizierter wird es bei dem Zählen von allen Wegen, denn es müssen alle Lösungen ausgeschlossen werden, bei denen neben dem eigentlichen Weg noch Kreise zwischen anderen Knoten existieren. (Vergleiche Kapitel 3.3.5) Diese „Subtour Elimination" macht das Lösen des IPs sehr aufwendig. Es gibt Programme, die alle möglichen Lösungen eines 0/1 IPs bestimmen und zählen können, z.B. Zerone oder Azove (siehe [14] und [15]). Bei einem großen Graphen wie dem S- und U-Bahnnetz sind die Lösungen ohne Kreise wegen der vielen zusätzlichen Bedingungen jedoch quasi unlösbar. Nur in kleineren Graphen, wie den in Kapitel 5 verwendeten Testgraphen, lassen sich über IPs tatsächlich Wege zählen.

Eine Testrechnung mit CPlex (siehe [15]), einem kommerziellen IP-Löser, der auch Lösungen zählen kann, brauchte im vollständigen Graphen mit 10 Knoten 1234,2 s um die 109601 Wege zu finden. Das IP, in dem CPlex dafür Lösungen gezählt hat, hatte 90 Variablen und 1031 Nebenbedingungen.

6.2 Vergleich

Die Versuche zeigen, dass der Graphenansatz die Struktur des Problems deutlich besser ausnutzt als der IP-Ansatz. Während der kürzeste Weg auf beide Arten gut bestimmbar ist, ist das Zählen von Wegen als Zählen von kreisfreien Lösungen eines IPs nur deutlich langsamer und bei großen Graphen gar nicht mehr möglich.

Dieses Ergebnis ist dadurch zu erklären, dass Programme, die IPs lösen, für allgemeinere IPs konzipiert sind, von denen Wegeprobleme in IP-Form ein Spezialfall sind. Der Algorithmus von Dijkstra und die Tiefensuche sind dagegen genau für das jeweilige Wegeproblem entwickelt und nutzen dadurch die Graphstruktur beim Finden der Wege besser aus.

6.3 Schlusswort

Wie kann man graphentheoretische Wegeprobleme lösen?

Am Beispiel des Kürzeste-Wege-Problems und des Zählens von Wegen wurde diese Frage aus unterschiedlichen Aspekten heraus beantwortet: Im ersten Teil wurden Grundlagen der Graphentheorie erläutert. Mit dem Dijkstra-Algorithmus wurde ein bekannter Algorithmus vorgestellt und außerdem ein eigener Algorithmus zum Zählen von Wegen entwickelt.

Durch die Zusammenarbeit mit dem Konrad-Zuse-Zentrum kam ich dazu, dass man Wegeprobleme auch als ganzzahliges Programm betrachten kann, was eng mit der Thematik aktueller Forschungsprojekte dort zusammenhängt. Diese Herangehensweise an Wegeprobleme wird in Büchern zur Graphentheorie nicht erwähnt, stellt aber eine interessante Verbindung der Graphentheorie zu einem anderen Gebiet der Mathematik dar.

An dieser Stelle möchte ich mich daher noch einmal herzlich bei R. Borndörfer und S. Heinz vom ZIB dafür danken, dass sie sich die Zeit genommen haben, mir einen Einblick in ihre Arbeit zu ermöglichen und viele Fragen zu beantworten.

Ich hoffe, dass der Leser in dieser Facharbeit etwas über die Hintergründe von Routenplanern und Ähnlichem erfahren konnte und wer hätte schließlich gedacht, dass es mehrere zehn Millionen verschiedene Wege gibt, die man in Berlin mit der S- und U-Bahn nehmen kann, um ans Ziel zu kommen?

7 Literatur- und Quellenverzeichnis

1 J. F. Kurose, K. W. Ross: *Computer Networking*. Pearson Education, 2005

2 P. Läuchli: *Algorithmische Graphentheorie*. Birkhauser Verlag Basel, 1991

3 G. L. Nemhauser, A. H. G. Rinnooy Kann, M. J. Todd: *Optimization*. Elsevier Science Publishers B.V., 1989

4 N. L. Biggs, E. K. Lloyd, R. J. Wilson: *Graph Theory 1736-1936*. Oxford University Press, 1976, 1986

5 R. K. Ahuja, T. L. Magnanti, J. B. Orlin: *Network Flows*. Prentince-Hall, 1993

6 G. Stiege: *Graphen und Graphalgorithmen*. Shaker Verlag, 2006

7 M. Grötschel: *Graphen- und Netzwerkalgorithmen*, Skriptum zur Vorlesung, 2003

8 R. Borndörfer, M. Grötschel, A. Löbel: *Der Schnellste Weg zum Ziel*, ZIB-Report 99-32, 2000

9 M. Behle: *On Threshold BDDs and the Optimal Variable Ordering Problem,* COCOA 07 LNCS 4616 127-135 2007

10 M. Behle, F. Eisenbrand: *0/1 vertex and facet enumeration with BDDs,* ALENEX 07

11 http://zimpl.zib.de (14.12.2007)

12 http://scip.zib.de (14.12.2007)

13 http://www.bvg.de (14.12.2007)

14 http://www.math.tu-bs.de/mo/research/zerone.html (14.12.2007)

15 http://www.mpi-inf.mpg.de/~behle/azove.html (14.12.2007)

16 http://www.ilog.com/products/optimization (14.12.2007)

8 Anhang

8.1 CD

Lauffähiges Javaprogramm mit Javadoc-Dokumentation (im Ordner „dist“)

Das S- und U-Bahnnetz als Knoten- und Kantenliste in csv-Format

Knotenliste für 10 Knoten mit zwei Kantenlisten (ein kleiner Testgraph und der vollständige Graph)

Das S- und U-Bahnnetz als lineares Programm für Wege von Heerstr. nach Dahlem-Dorf

8.2 Quelltext des Programms