ADS: Algorithmen und Datenstrukturen 2

Teil 4

Prof. Peter F. Stadler & Sebastian Will

Bioinformatik/IZBI
Institut für Informatik
& Interdisziplinäres Zentrum für Bioinformatik
Universität Leipzig

30. April 2014

Kürzeste Wege

- kantenmarkierter (gewichteter) Graph $G = (V, E, g)$
  - Weg/Pfad P der Länge n: $(v_0, v_1), (v_1, v_2), , ..., (v_{n-1}, v_n)$
  - Gewicht (Länge) des Weges/Pfads $w(P) = \sum_{i=1}^{n} g((v_{i-1}, v_i))$
  - Distanz $d(u, v)$: Gewicht des kürzesten Pfades von u nach v
- Varianten
  - nicht-negative Gewichte vs. negative und positive Gewichte
  - Bestimmung der kürzesten Wege:
    a) zwischen allen Knotenpaaren,
    b) von einem Knoten u aus
    c) zwischen zwei Knoten u und v
- Bemerkungen
  - kürzeste Wege sind nicht immer eindeutig
  - kürzeste Wege müssen nicht existieren, z.B. wenn:
    - kein Weg existiert oder
    - ein Zyklus mit negativem Gewicht existiert

Zur Erinnerung: Warshall's Transitive Hülle

A = Adjazenzmatrix $G(V, E)$, $n = |V|$

```
Von i = 1 bis n
  A[i][i] = 1
Von j = 1 bis n
  Von i = 1 bis n
    Falls A[i][j] = 1 Dann
      Von k = 1 bis n
        Falls A[j][k] = 1 dann A[i][k] = 1
```


Warshall's Algorithmus für Distanzen

Warshall-Algorithmus lässt sich einfach modifizieren, um kürzeste Wege zwischen allen Knotenpaaren zu berechnen

```
Fuer alle Paare i,j tue A[i][j] = g((i,j))
Von j = 1 bis n
   Von i = 1 bis n
      Von k = 1 bis n
         A[i][k] = min (A[i][k], A[i][j] + A[j][k])
```

Annahme: kein Zyklus mit negativem Gewicht vorhanden
Komplexität: $O(n^3), n = |V|$

Dijkstra-Algorithmus I

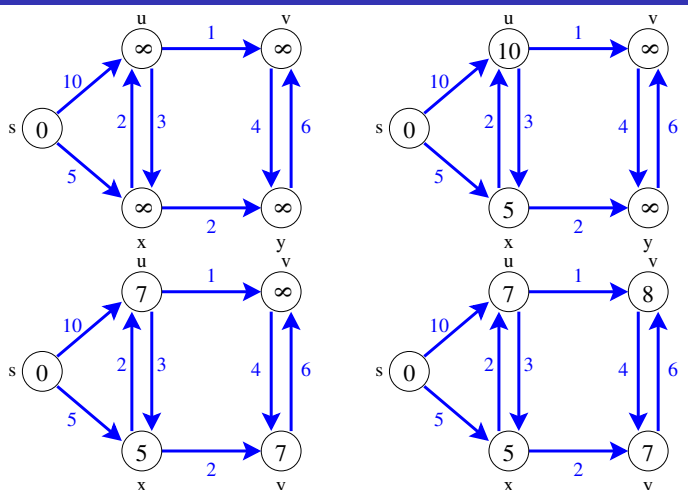
Bestimmung der von einem Knoten ausgehenden kürzesten Wege

- gegeben: kanten-bewerteter Graph $G = (V, E, g)$ mit $g : E \to \mathbb{R}_0^+$ (Kantengewichte)
- Startknoten s; zu jedem Knoten u wird die Distanz zu Startknoten s in $D[u]$ geführt
- Q sei Prioritäts-Warteschlange (sortierte Liste); Priorität = Distanzwert
- Funktion $succ(u)$ liefert die Menge der direkten Nachfolger von u
- Verallgemeinerung der Breitensuche (gewichtete Entfernung)
- funktioniert nur bei nicht-negativen Gewichten
- Optimierung gemäß Greedy-Prinzip

Dijkstra-Algorithmus II

```
Fuer alle Knoten v tue D[v] = unendlich
D[s] = 0
PriorityQueue Q = V
Solange Q nicht leer dann
   v = naechster Knoten aus Q mit kleinstem Abstand D
   Entferne v aus Q
   Fuer jeden Nachbarn u in succ(v) & Q
      Falls D[v] + g((v,u)) < D[u] dann
         D[u] = D[v] + g((v,u))
```


Dijkstra: Beispiel



Dijkstra-Algorithmus: Korrektheit

Korrektheitsbeweis

- nach i Schleifendurchgängen sind die Längen von i Knoten, die am nächsten an s liegen, korrekt berechnet und diese Knoten sind aus Q entfernt.
- Induktionsanfang: s wird gewählt, $D(s) = 0$
- Induktionsschritt: Nimm an, v wird aus Q genommen. Der kürzeste Pfad zu v gehe über direkten Vorgänger v' von v. Da v' näher an s liegt, ist v' nach Induktionsvoraussetzung mit richtiger Länge $D(v')$ bereits entfernt. Da der kürzeste Weg zu v die Länge $D(v') + g((v', v))$ hat und dieser Wert bei Entfernen von v' bereits v zugewiesen wurde, wird v mit der richtigen Länge entfernt.
- erfordert nicht-negative Kantengewichte (wachsende Weglänge durch hinzugenommene Kanten)

Dijkstra-Algorithmus: Eigenschaften

Komplexität $O(n^2), n = |V|$

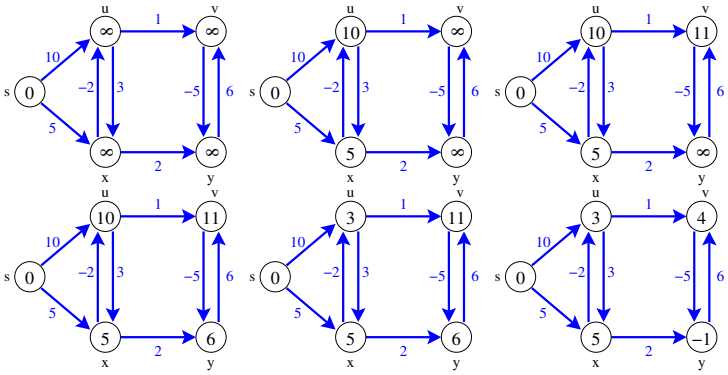
- n-maliges Durchlaufen der äußeren Schleife liefert Faktor $O(n)$
- innere Schleife: Auffinden des Minimums begrenzt durch $O(n)$, ebenso das Aufsuchen der Nachbarn von v
- Pfade bilden aufspannenden Baum (der die Wegstrecken von s aus gesehen minimiert)
- Bestimmung des kürzesten Weges zwischen u und v: Spezialfall für Dijkstra-Algorithmus mit Start-Knoten u (Beendigung sobald v aus Q entfernt wird)

Kürzeste Wege mit negativen Kantengewichten (ohne negative Zyklen)

Bellmann-Ford-Algorithmus BF(G,s)

```
Fuer alle Knoten v tue D[v] = unendlich
D[s] = 0
Von i = 1 bis n-1
   Fuer alle Paare (u,v)
      Falls D[u] + g((u,v)) < D[v] dann
         D[v] = D[u] + g((u,v))
```


Bellmann-Ford: Beispiel

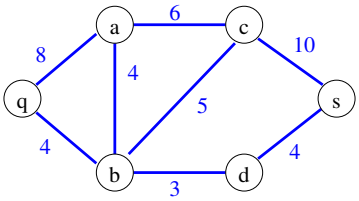


(Kanten werden hier in lexikographischer Ordnung durchlaufen), also $(s,u)(s,x)(u,v)(u,x)(v,y)$

Flüsse in Netzwerken I

Anwendungsprobleme:

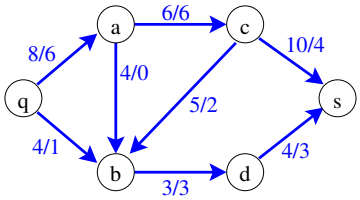
- Wieviel Autos können durch ein Straßennetz fahren?
- Wieviel Abwasser fasst ein Kanalnetz?
- Wieviel Strom kann durch ein Leitungsnetz fließen?
- und Optimierung



Kantenwert: Kapazität $c(e)$

Flüsse in Netzwerken II

- Ein (Fluss-) Netzwerk ist ein gerichteter Graph $G = (V, E, c)$ mit ausgezeichneten Knoten q (Quelle) und s (Senke), sowie einer Kapazitätsfunktion $c :\rightarrow \mathbb{R}^+$.
- Ein Fluss für das Netzwerk ist eine Funktion $f : E \rightarrow \mathbb{R}_0^+$, so dass gilt:
  - Kapazitätsbeschränkung: $\forall e \in E : f(e) \leq c(e)$.
  - Flusserhaltung: $\forall v \in V \setminus \{q, s\} : \sum_{(v',v)\in E} f((v', v)) = \sum_{(v,v')\in E} f((v, v'))$
  - Der Wert von f, $w(f)$, ist die Summe der Flusswerte der die Quelle q verlassenden Kanten: $\sum_{(q,v)\in E} f((q, v))$



Flüsse in Netzwerken III

Gesucht: Fluss mit maximalem Wert

- begrenzt durch Summe der aus q wegführenden bzw. in s eingehenden Kapazitäten
- jeder weitere "Schnitt" durch den Graphen, der q und s trennt, begrenzt max. Fluss

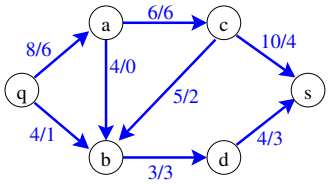
Schnitt (A, B) eines Fluss-Netzwerks ist eine Zerlegung von V in disjunkte Teilmengen A und B, so dass $q \in A$ und $s \in B$.

- Die Kapazität des Schnitts ist $c(A, B) = \sum_{u\in A, v\in B} c((u, v))$
- minimaler Schnitt (minimal cut): Schnitt mit kleinster Kapazität

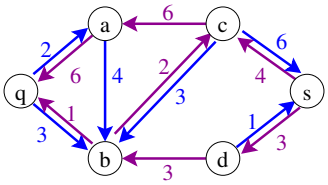
Restgraph

Sei f ein zulässiger Fluss für $G = (V, E, c)$. Sei $E' = \{(v, w)|(v, w) \in E \vee (w, v) \in E\}$

- Wir definieren die Restkapazität einer Kante $e = (v, w)$ wie folgt:
  - $rest(e) = c(e) - f(e)$ falls $e \in E$
  - $f((w, v))$ falls $(w, v) \in E$
- Der Restgraph von f (bzgl. G) besteht aus den Kanten $e \in E'$, für die $rest(e) > 0$



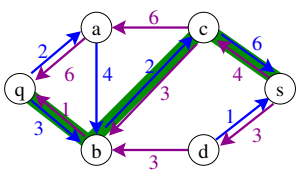
$c(e)/f(e)$



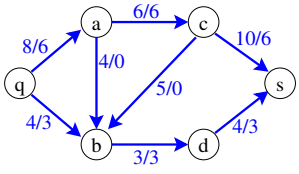
$rest(e)$

Zunehmender Weg

- Jeder gerichtete Pfad von q nach s im Restgraphen heißt zunehmender Weg.



- Optimierung des Flusses für G:
  - entlang des zunehmenden Weges q
  - um $\min(rest(e))$ mit e liegt auf q



Min-Cut-Max-Flow-Theorem

Theorem (Min-Cut-Max-Flow-Theorem): Sei f ein zulässiger Fluss für G. Dann sind folgende Aussagen äquivalent:

- f ist maximaler Fluss in G.
- Der Restgraph von f enthält keinen zunehmenden Weg.
- $w(f) = c(A, B)$ für einen Schnitt (A, B) von G.

Ford-Fulkerson-Algorithmus

Ford-Fulkerson-Algorithmus:

- füge solange zunehmende Wege zum Gesamtfluss hinzu wie möglich
- Kapazität erhöht sich jeweils um Minimum der verfügbaren Restkapazität der einzelnen Kanten des zunehmenden Weges

```
Solange es einen zunehmenden Weg p in Restgraph G gibt
   r = min(rest(e)| e liegt auf p)
   Fuer alle e = (v,w) auf p tue
      Falls e in E dann
         f(e) = f(e) + r
      Sonst f((w,v)) = f((w,v)) - r
```


Maximales Bipartites Matching

Beispiel:

- Eine Gruppe von Erwachsenen und eine Gruppe von Kindern besuchen Disneyland.
- Auf der Achterbahn darf ein Kind jeweils nur in Begleitung eines Erwachsenen fahren.
- Nur Erwachsene/Kinder, die sich kennen, sollen zusammen fahren. Wieviele Kinder können maximal eine Fahrt mitmachen?

Matching

Matching (Zuordnung) M für ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge der Kanten, so dass jeder Knoten in V in höchstens einer Kante vorkommt

- $|M|$ = Größe der Zuordnung
- Perfektes Matching: kein Knoten bleibt "allein" (unmatched), d.h. jeder Knoten ist in einer Kante von M vertreten

Matching M ist maximal, wenn es kein Matching M' gibt mit $|M| < |M'|$

Ein bipartiter Graph ist ein Graph, dessen Knotenmenge V in zwei disjunkte Teilmengen V_1 und V_2 aufgeteilt ist, und dessen Kanten jeweils einen Knoten aus V_1 mit einem aus V_2 verbinden. (also keine Kanten innerhalb von V_1 oder V_2)

Bipartites Matching und maximaler Fluss

Maximales Matching kann auf maximalen Fluss zurückgeführt werden:

- Quelle und Senke hinzufügen.
- Kanten von V_1 nach V_2 richten.
- Jeder Knoten in V_1 erhält eingehende Kante von der Quelle.
- Jeder Knoten in V_2 erhält ausgehende Kante zur Senke.
- Alle Kanten erhalten Kapazität $c(e) = 1$

→ Anwendung des Ford-Fulkerson-Algorithmus

Zusammenfassung I

- Viele wichtige Informatikprobleme lassen sich mit gerichteten bzw. ungerichteten Graphen behandeln.
- wesentliche Implementierungsalternativen: Adjazenzmatrix und Adjazenzlisten
- Algorithmen mit linearem Aufwand:
  - Traversierung von Graphen: Breitensuche vs. Tiefensuche
  - Topologisches Sortieren
  - Test auf Azyklität (Kreisfreiheit)

Zusammenfassung II

- Weitere wichtige Algorithmen:
  - Bestimmung der transitiven Hülle: Warshall-Algorithmus
  - Kürzeste Wege: Dijkstra, Bellmann-Ford und Warshall
  - Minimale Spannbäume: Kruskal-Algorithmus
  - maximale Flüsse bzw. maximales Matching: Ford-Fulkerson-Algorithmus
- viele NP-vollständige Optimierungsprobleme
  - Traveling-Salesman-Problem, Cliquenproblem, Färbungsproblem ...
  - Bestimmung eines planaren Graphen (2D-Graph-Darstellung ohne überschneidende Kanten)