Graphalgorithmen

Knotentraversierung

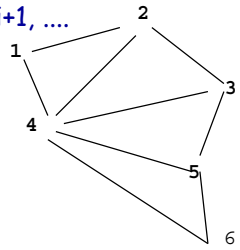
Breitensuche, (Breadth first search)

Erst alle Knoten im Abstand i von s, dann i+1, ...

bfs(Knoten s) {

Besuche jeden Nachfolger s' von s;
Für jeden Nachfolger s'
bfs (s');
}

Offenbar unzulänglich! Terminiert nicht.



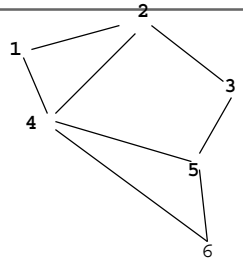
Status der Knoten merken:
- nicht bearbeitet ("weiß")
- besucht ohne die Nachfolger besucht zu haben (rekursiver Aufruf): grau
- besucht und alle Nachbarn besucht: schwarz

Fertig, wenn nur noch schwarze Knoten ?!

Graphalgorithmen: Breitensuche

- Annahme: ungerichteter zusammenhängender Baum

```
Queue q;
 void bfs (Node s);
s.colour=grey;
wListe.enqueue(s);
while (!wListe.empty()) {
   x = q.dequeue();
   for each nachfolger(x) {
    if (x.colour = white) {
          x.colour = gray;
       doSomethingWith(x);
      q.enqueue(x);
    }
    }
   u.colour = black;
   q.enqueue(u);
```



x	q	black
1	-	
1	2	-
1	2,4	-
2	4	1
2	4,3	1
4	3,5	1,2
4	3,5,6	1,2
3	5,6	1,2,4
5	6	1,2,4,3
6	- -	1,2,4,3,5
-	-	1,2,4,3,5,6

Breitensuche

- Ergänzung:

  für jeden Knoten x ist der Vorgänger (Nachbarknoten), y von dem x zum erstenmal (x.color = white) erreicht wurde, eindeutig.

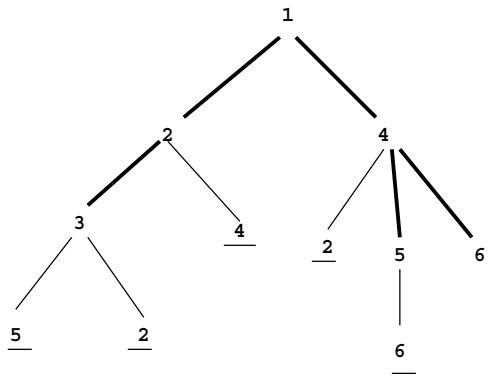
  .....

```
if (x.colour = white) {
        x.previous = u;
        x.colour = gray;
        doSomethingWith(x);
        q.enqueue(x);
    }
```

  Es gilt: Weg von s nach x ist (kanten-)minimal....

  ..... denn alle Knoten der Entfernung i vom Startknoten x werden vor denen der Entfernung i + 1 besucht.

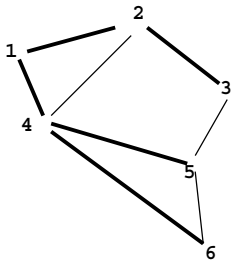
Baumdarstellung von Graphsuchproblemen



Alle Kanten markieren, die zu weißen (noch nicht besuchten) Knoten führen -> spannender Baum

Breitensuche und Spannender Baum

- Spannender Baum S(G)von G:
  Kanten so entfernen, dass
  b von a in S erreichbar gdw in G erreichbar und S hat minimale Kantenzahl



- Verfahren liefert spannenden Baum:
  - ungerichtet: klar, Wurzelpfad von Knoten bis zur Wurzel verfolgen und von der Wurzel bis zum Zielknoten.
  - gerichtet: nur wenn stark zusammenhängend.
- Frage: Minimal bei Kantengewicht w(k) != 1 für alle Kanten k? Sicher nicht!

Laufzeit Breitensuche

Laufzeit: Adjazenzlisten-Repräsenation

```
while (!wListe.empty()) {
      x = q.dequeue();
      for each nachfolger(x) {.....
```

Jeder Knoten wird nur einmal auf Warteliste gesetzt : O(E)
... und nur einmal entfernt. Dabei Traversierung aller Nachfolger => Adjazenzliste wird nur einmal traversiert => insgesamt O(K) Operationen auf Kanten

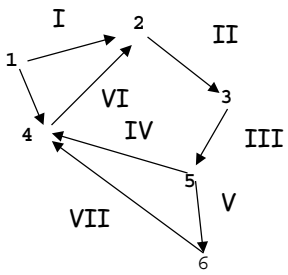
Laufzeit: O(K + N)

mit K, N sind hier jeweils die Anzahlen |K|, |N| gemeint

Graphalgorithmen: Tiefensuche

- Prinzip: verlängere aktuellen Pfad bis keine Verlängerung mehr möglich. Backtrack zum letzten Knoten, der Alternative (ausgehende Kante, die noch nicht durchlaufen wurde) hat.

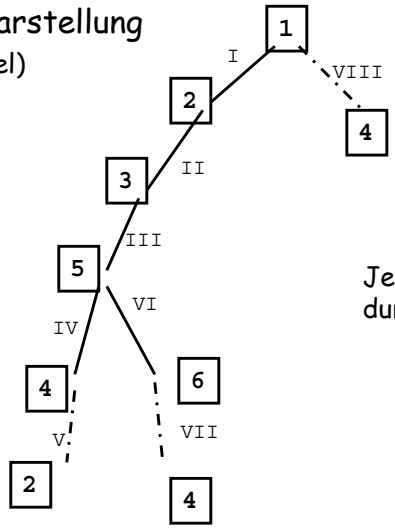
gray	black
1	-
1,2	-
1,2,3	-
1,2,3,5	-
1,2,3,5,4	-
1,2,3,5	4
1,2,3,5,6	4
1,2,3,5	4,6
1,2	4,6,5,3
1	4,6,5,3,2
1	4,6,5,3,2
-	4,6,5,3,2,1



Backtrack-Implementierung am einfachsten implizit mit Rekursionsstapel (stack)

Graphalgorithmen: Tiefensuche

- Baumdarstellung (Beispiel)

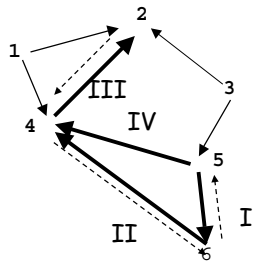


Jede Kante einmal durchlaufen

Tiefensuche ("depth first search")

```
void dfsVisitNode(node s) {
  s.colour = grey;
  for all  x=nachfolger(s) {
     if(x.colour = white) {
        x.previous = s;
        x.colour = gray;
        x.doSomething;
        dfsVisitNode(s);
        }
    x.colour = black;
 }
}
```

Modifiziertes Beispiel:



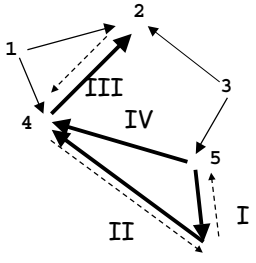
Falsch: Nur korrekt, wenn Graph stark zusammenhängend!
s=5 : Nur 5,6,4,2 werden traversiert.
Allgemein entsteht Wald von Bäumen.

Tiefensuche

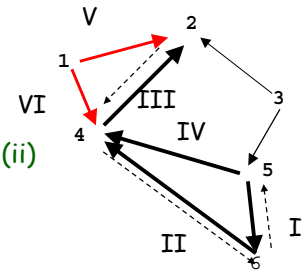
```
void dfs( ) {
// requires: all nodes white
 for all nodes s
   if (s.color = white)
    dfsVisitNode(s);
}
```

Beispiel: Reihenfolge s= 5,1,2,3,4,6

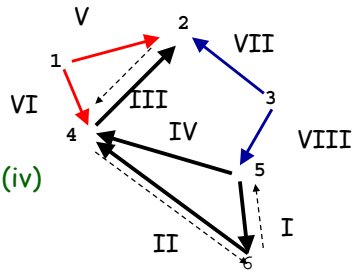
(i)



(ii)



(iv)



Laufzeit Tiefensuche

- Laufzeit Adjazenzlisten-Repräsentation

```
for all nodes s
   if (s.color = white)
      dfsVisitNode(s)
```

O(E) Aufrufe

```
for all  x=nachfolger(s) {
     if(x.colour = white) {
        x.previous = s;
          x.colour = gray;
        x.doSomething;
          dfsVisitNode(s);
        }
```

O(|Nachfolger(s)|

Insgesamt: $O(E) + \sum_{s \in E} |\text{Nachfolger}(s)| = O(E+K)$

Kürzeste Wege (Shortest path)

- Aufgabe
  Finde in gerichtetem, kantengewichtetem Graphen
  G=(E,K) , w: K -> Real, Weg x = x0,x2,...,xn=y
  von x nach y mit minimaler Summe der Kantenkosten*:
  d(x,y) := $\sum$ w((xi,xi+1)) minimal unter allen Wegen p von x nach y

- Anwendungen:
  - Routenplaner
  - Verdrahtung von Platinen
  - Optimale Weg aus Labyrinth
  - n x n – Puzzle .....

* Kosten = Gewicht

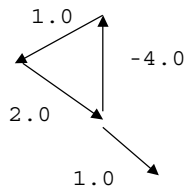
Kürzeste Wege: Klassifikation

- Kürzester Weg s-> y (Single pair shortest path)
- Alle kürzesten Wege von s zu beliebigem Knoten x (single source shortest path)
- Alle kürzesten Wege von beliebigem Startknoten zu *einem* Zielknoten t (single destination shortest path)
  -> auf zweites Problem zurückführen: alle Richtungen umkehren!
- Alle Paare von kürzesten Wegen (all pairs shortest path)

"Kürzeste Wege" und "Alle kürzesten Wege" sind gleich aufwendig!

Kürzeste Wege

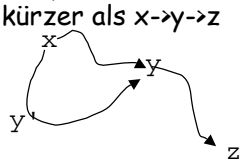
- Kantengewichtung ...

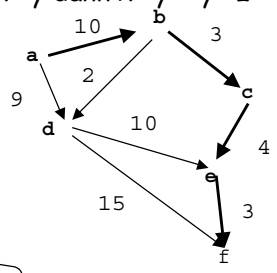


- ... sollte nicht negativ sein?
- Kürzeste-Wege-Problem nicht wohldefiniert, wenn es einen Zyklus mit negativer Kantensumme gibt, der von s erreichbar ist
- Annahme hier: positive Kantengewichte

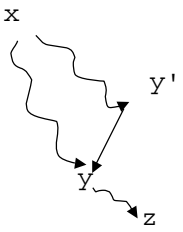
Kürzeste Pfade

- Eigenschaften
  - x-> y-> z kürzester Pfad, dann x -> y kürzester Pfad klar, sonst : wenn x->y' kürzer als x->y dann x->y'->y->z kürzer als x->y->z



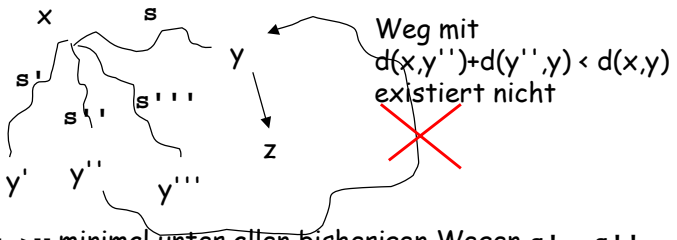


  - d(x,y) > d(x,y') +w((y',y)) dann ist s -> y kein Teilpfad des kürzesten Weges von x nach z



Kürzeste Pfade

- .... Eigenschaften



`s = x->y` minimal unter allen bisherigen Wegen `s', s'',....,` dann ist s Weg minimaler Länge von x nach y.

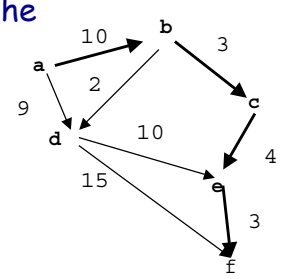
Also: wenn man unter den bisherigen Wegen den minimalen x->y fortsetzt (nach z), hat man den minimalen Weg s->y gefunden.

Achtung: gilt nicht, wenn negative Kantengewichte zugelassen. Warum?

Algorithmus von Dijkstra*

Idee: Kostengesteuerte Breitensuche

Menge P aller kürzesten Wege x -> e schrittweise aufbauen, bereits ermittelte kürzeste Pfade merken:



Wenn q = x->e minimale
Gesamtkosten d(x,e) aller q' aus P hat:

P = P ∪
{(s->x-z; d(x,e)+w((e,z))) | z Nachfolger von e}\{(x->e)}

SP= SP ∪ {(x->e, d(x,e))

*Edgar W. Dijkstra, berühmter holländischer Informatiker, *1930

Dijkstra-Algorithmus

Bestimmung der gewichteten Weglänge x -> y für alle Knoten y

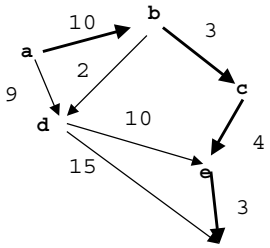
SP'={ (y, d, minimal)| yKnoten von G(E,K), y.d = ∞, minimal = false};
`x.d` : Gewicht des bisher gewichtsminimalen Weges x -> y,
`boolean minimal`: true wenn kürzester Weg x->y gefunden

```
void shortestPath(node s) {
  heap h = new...;  // z.B.Min-heap
  x.d=0;
  for all node y {h.enqueue(y);]
  while (!h.empty) {
    y = h.dequeueMin;  // minimal path up to now
    y.minimal = true;
      for each z=y.successor {
       if(y.d + w((y,z)) < z.d) z.d=y.d + w((y,z));
                                // shorter path found
      } ... // Minimale Kosten d(x,y) für y gefunden
} // Minimale Kosten d(x,e) für Knoten e gefunden
```

Frage: wie repräsentiert man den Weg??

Dijkstra-Algorithmus: Beispiel

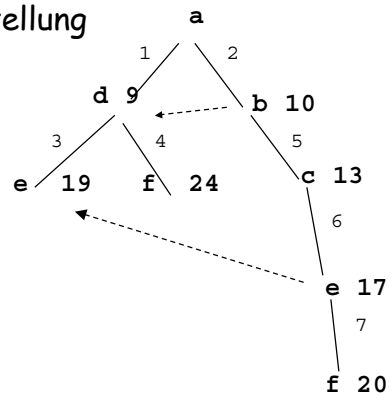
SP: hier Menge der von a ausgehenden *minimalen* Wege (`minimal=true`), andere Weg(`minimal=false`) in P



1. P = {(ab;10), **(ad;9)**}
   SP = {(a->d;9)}
2. P = {**(ab;10),** (ad,e;19), (adf, 24)}
   SP = {(ad;9), (ab;10)}
3. P = {(abd;12),(abc;13), (ade;19), (adf; 24)}
   SP = {(ad;9), (ab;10), (abc;12)}
4. P = { **(abc;13)**, (ade;19), (adf; 24)}
   SP = {(ad;9), (ab;10), (abc;13)}
5. P = {**(abce;17),** (abde;22), (adf; 24)}
   SP = {(ad;9), (ab;10), (abc;13), (abce;17)}
6. P = {**(abcef;17),** (adf; 24)}
   SP = {(ad;9), (ab;10), (abc;13), (abce;17), (abcef;20),}

Kürzeste Wege (Dijkstra)

- Baumdarstellung



Kürzeste Wege (Dijkstra): Laufzeit

- Jeder Knoten genau einmal in Prioritätsschlange: O(E)
- Je einmal pro Knoten Nachfolger bestimmen: O(K) für Adjazenzliste

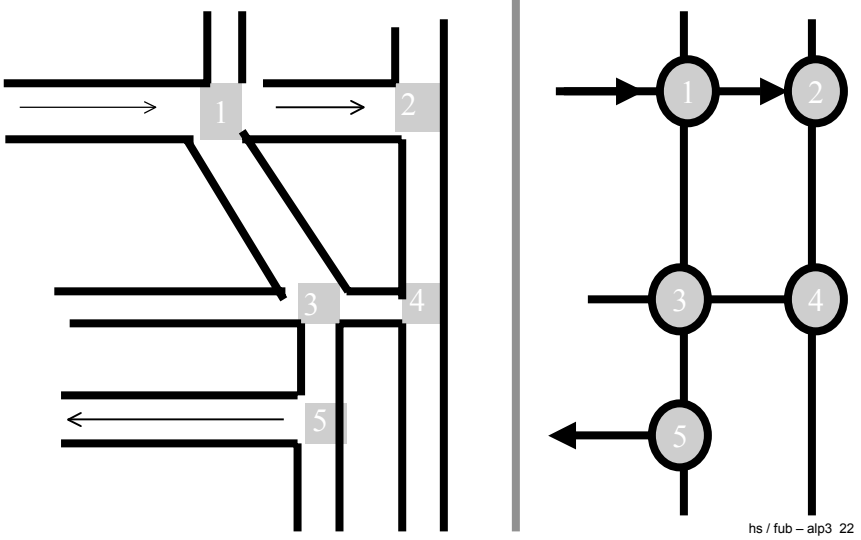
=> O(E + K) + O(Verwaltung des heap)
oder einer anderen Datenstruktur
Schlechtester Fall: jede Verlängerung eines Weges um Kante (x,z) liefert neue Weglänge s->z: O(K) Veränderungen
Minimum bestimmen: O(log E)

obere Schranke im schlechtesten Fall
O(E+K) + O(K)*O(logE)

Abhängig von Implementierung der Verwaltungsdatenstrukturen!

Routenplaner

Graph



Breadth-first Search (cont'd)
(Branching Factor = 3)

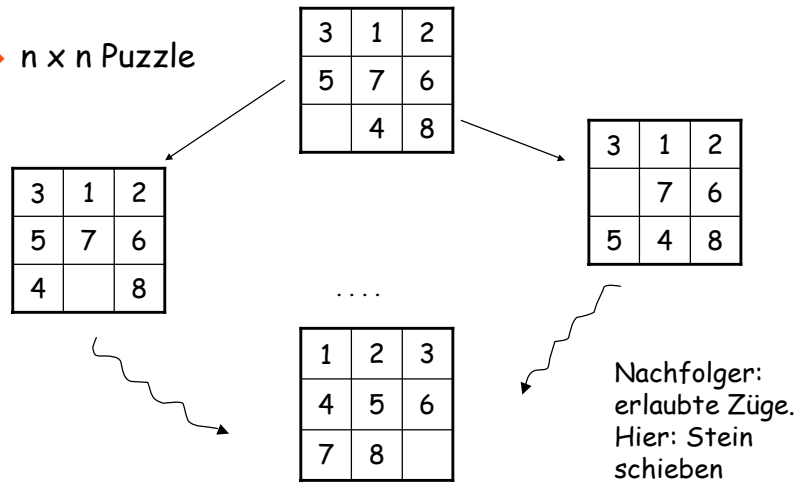
Platzbedarf kritischer als Laufzeit!
US-Straßenkarte: Tiefe ungefähr 3000)

Depth	Nodes	Time	Memory
0	1	1 ms	100 B
2	13	13 ms	1 kB
4	121	.1 sec	12 kB
6	10^3	1 sec	100 kB
8	10^4	10 sec	1 MB
10	10^5	100 sec	10 MB
50	10^{24}	10^{13}	10^{20} TB

Aus Artificial Intelligence: A Modern Approach,
Stuart Russell and Peter Norvig,
Upper Saddle River, NJ: Prentice Hall, c. 1995

Graphrepräsentation von Spielen

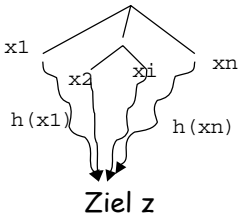
- n x n Puzzle



Viele Knoten, implizit: alle erlaubten Zustände.

Kürzeste Wege: Heuristiken

- Sehr große Knotenzahlen, evtl. implizit
  - Dijkstra – Algorithmus nicht anwendbar
  - Einschränkung auf "single pair shortest path"
  - Bringt das was?? Gleiche asymptotische Laufzeit !!
  - Ja: heuristische Schätzung der Entfernung vom Zielknoten einführen



Bisheriger Weg
Länge w(xi)

Geschätzte Entfernung
vom Ziel: h(xi,z)

Heuristische Wegsuche

- A*-Algorithmus
  - Ziel: finde gewichts-(kosten-)minimalen Weg von Startknoten s zum Zielknoten z
  - Voraussetzung: für jeden Knoten x sei eine optimistische Schätzfunktion h(x,z) für den Weg x -> z gegeben.
    Seien d(x,t) die (unbekannten) tatsächlichen Kosten,
    h heißt optimistisch, wenn h(x,t) < = d(x,t)
  - Es existiert immer eine optimistische Schätzfunktion:
    Für alle x: h(x,t) = 0
  - Ist x ein Knoten, zu dem ein Weg s -> x gefunden wurde, dann seien die (nicht notwendig minimalen) Kosten d(s,x)

Heuristische Suche

- A*-Algorithmus: analog Dijkstra mit Gewichtsfunktion
  f(s,t) = d(s,x) + h(x,t)
  Kosten = tatsächliche des zurückgelegten Weges pus Schätzung
  für den Restweg.

- Änderung Dijkstra-Algorithmus:
  wenn f(s,t) = d(s,x) + h(x,t) minimal, muss d(s,x) NICHT minimal sein. Folge: auch zu Knoten, die schon expandiert wurden, kann evtl. günstigerer Weg gefunden werden.
  Technisch: kein Knoten darf von queue entfernt werden.
- Gefundener Weg minimal, wenn h optimistisch