Der Algorithmus von Dijkstra

Algorithmus

procedure Dijkstra(s) :
Q := V − { s};
for v in Q **do** d[v] := ∞**od**;
d[s] := 0;
while Q ≠ ∅**do**
  choose v in Q with minimal d[v];
  Q := Q − { v};
  forall u adjacent **to** v **do**
    d[u] := min { d[u], d[v] + length(v, u)}
  od
od

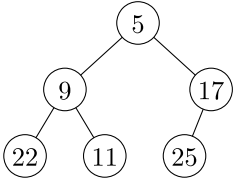
Wie implementieren wir Q?

Der Algorithmus von Dijkstra – Beispiel

Priority Queues (Prioritätswarteschlangen)

Operationen einer Prioritätswarteschlange Q:

1. Einfügen von x mit Gewicht w (insert)
2. Finden und Entfernen eines Elements mit minimalem Gewicht (extract-min)
3. Das Gewicht eines Elements x auf w verringern (decrease-weight)



Heap: alle Operationen in $O(\log n)$ Schritten
(n ist die aktuelle Anzahl von Elementen im Heap)

Algorithmus von Dijkstra – Laufzeit

Theorem

Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ Schritten.

Beweis.

Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt. Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. □

Ein Fibonacci-Heap benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

Java

```
public static⟨V extends Comparable⟨V⟩⟩ Map⟨V, Double⟩
dijkstra(Graph⟨V⟩ G, V s, Map⟨Edge⟨V⟩, Double⟩ length, Map⟨V, V⟩ pred) {
  Map⟨V, Double⟩ dist = new HashMap⟨V, Double⟩();
  PriorityQueue⟨V, Double⟩ queue = new SplayPriorityQueue⟨V, Double⟩();
  for(V u : G.allNodes()) dist.put(u, Double.MAX_VALUE);
  dist.put(s, 0.0); queue.insert(s, 0.0);
  while(!queue.isEmpty()) {
    V u = queue.extractMin();
    for(V v : G.neighbors(u)) {
      Double l = length.get(G.edge(u, v));
      if(l == null) continue;
      double d = dist.get(u) + l;
      if(d < dist.get(v)) {
        queue.decreaseKey(v, d); dist.put(v, d); if(pred ≠ null) pred.put(v, u); }
    }
  }
  return dist;
}
```

Spezialfall DAG

Können wir kürzeste Pfade in DAGs schneller finden?

Theorem

Kürzeste Pfade von einem Knoten s in einem DAG können in linearer Zeit gefunden werden.

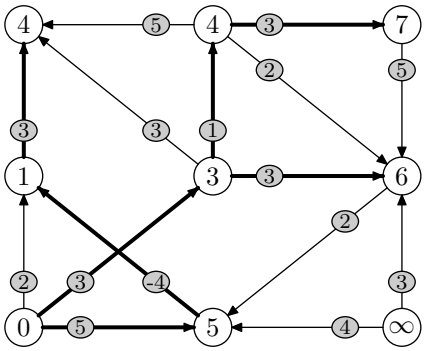
Beweis.

Relaxiere Knoten in topologischer Reihenfolge.

Laufzeit: $O(|V| + |E|)$.
Korrektheit: Jeder Knoten, der relaxiert, kennt zu diesem Zeitpunkt seinen echten Abstand. □

Frage: Sind negative Gewichte hier erlaubt?

Kürzeste Wege mit negativen Kantengewichten



Dijkstra funktioniert nicht mit negativen Kantengewichten.

Der Algorithmus von Bellman und Ford

Idee:

- Relaxiere alle Kanten
- Wiederhole dies, bis keine Änderung

Warum korrekt?

Induktion über die Länge eines kürzesten Pfads.
(Also genügen n Wiederholungen)

Was passiert bei Kreisen mit negativem Gewicht?

$\rightarrow$ Keine Terminierung.

Der Algorithmus von Bellman und Ford

Algorithmus

```
function Bellman − Ford(s) boolean :
for v in V do d[v] := ∞od;
d[s] := 0;
for i = 1 to|V| − 1 do
  forall(v, u) in E do
    d[u] := min { d[u], d[v] + length(v, u)}
  od
od;
forall(v, u) in E do
  if d[u] > d[v] + length(v, u) then return false fi
od;
return true
```

Der Algorithmus von Bellman und Ford

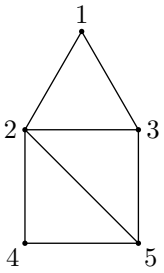
Theorem

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit Kantengewichten $E \to \mathbf{R}$ und ein Knoten $s \in V$.
Wir können in $O(|V| \cdot |E|)$ feststellen, ob ein Kreis mit negativem Gewicht existiert, und falls nicht, die kürzesten Wege von s zu allen Knoten berechnen.

Beweis.

Wir haben die Korrektheit bereits nachgewiesen.
Zur Laufzeit: Jeder Knoten wird $|V|$ mal relaxiert, also wird auch jede Kante $|V|$ mal relaxiert (in konstanter Zeit). □

Rekapitulation: Darstellung von Graphen



Adjazenzliste

1	2, 3
2	1, 3, 4, 5
3	1, 2, 5
4	2, 5
5	2, 3, 4

Alle bisherigen Algorithmen:

Adjazenzliste gute Darstellung

Kritische Operation: Alle ausgehenden Kanten besuchen.

All Pairs Shortest Paths

Eingabe: Gerichteter Graph mit Kantengewichten

Ausgabe: Abstände und kürzeste Wege zwischen allen Knotenpaaren

Laufzeit $O((|V|^2 + |V| \cdot |E|) \log |V|)$ mit Dijkstra.

$\rightarrow$ Wende Dijkstra auf jeden Knoten an.

Algorithmus von Floyd

Algorithmus

procedure Floyd1 :
for i $= 1, ..., n$ **do**
  for j $= 1, ..., n$ **do** $d[i, j, 0] := \text{length}[i, j]$ **od**
od;
for k $= 1, ..., n$ **do**
  for i $= 1, ..., n$ **do**
    for j $= 1, ..., n$ **do**
      $d[i, j, k] := \min \{ d[i, j, k-1], d[i, k, k-1] + d[k, j, k-1] \}$
    od
  od
od

Der Abstand von i nach j ist in $d[i, j, n]$ zu finden.

Theorem

Die kürzesten Wege zwischen allen Knotenpaaren eines gerichteten Graphen $G = (V, E)$ können in $O(|V|^3)$ Schritten gefunden werden.

Beweis.

Per Induktion: $d[i, j, k]$ enthält die Länge des kürzesten Pfades von i nach j, wenn nur $1, \ldots, k$ als Zwischenstationen erlaubt sind.
Dann enthält $d[i, j, n]$ den wirklichen Abstand. □

Algorithmus von Floyd

Einfachere Version:

Algorithmus

```
procedure Floyd :
for i = 1, ..., n do
  for j = 1, ..., n do d[i, j] := length[i, j] od
od;
for k = 1, ..., n do
  for i = 1, ..., n do
    for j = 1, ..., n do
      d[i, j] := min { d[i, j], d[i, k] + d[k, j]}
    od
  od
od
```

Spezialfall: Transitive Hülle – Algorithmus von Warshall

Frage: Zwischen welchen Knotenpaaren gibt es einen Weg?

Algorithmus

```
procedure Warshall :
for i = 1, ..., n do
  for j = 1, ..., n do D[i,j] := A[i,j] od
od;
for k = 1, ..., n do
  for i = 1, ..., n do
    for j = 1, ..., n do
      D[i,j] := D[i,j] || D[i,k] && D[k,j]
    od
  od
od
```

Transitive Hülle

Theorem

Wir können die transitive Hülle eines gerichteten Graphen $G = (V, E)$, der k starke Zusammenhangskomponenten hat, in $O(|V| + |E'| + k^3)$ Schritten berechnen, wobei E' die Kanten der transitiven Hülle sind.

Beweis.

1. Berechne die starken Zusammenhangskomponenten
2. Schrumpfe jede SCC X zu einem Knoten
3. Berechne die transitive Hülle H dieses Graphen
4. Für jede Kante (X, Y) in H gib alle Kanten (x, y) mit $x \in X, y \in Y$ aus.

□