Kürzeste Wege bei nichtnegativen Gewichten

Wenn kein Kantengewicht negativ ist, gibt es eine Methode, die jede Kante nur einmal relaxiert.
Man lässt ein t von 0 nach ∞ wachsen und hält folgendes aufrecht, wobei $V_t = \{x \in V | d_s(x) \leq t\}$:

Invariante

1. Für alle $x \in V$ mit $d_s(x) \leq t$ gilt: $\delta[x] = d_s(x)$ und $\pi[x]$ ist Vorgänger auf kürzesten Pfad von s nach x
2. Für alle $x \in V$ mit $d_s(x) > t$ gilt: $\delta[x]$ ist Länge eine kürzesten Pfades von s nach x mit vorletzten Knoten u in V_t; und, falls so ein Pfad existiert, dann $\pi[x] = u$

Dijkstra Algorithmus

Der Algorithmus verwendet eine Prioritätenschlange für V mit $\delta[x]$ als Schlüssel von x.

Initialisierung: Für alle $x \in V$ setze $\delta[x] = \infty$, $\pi[x] = \text{NIL}$
Für Startknoten s setze $\delta[s] = 0$. Bilde (Min) Prioritätenschlange Q für V.

while Q not emptry **do**
  u = DELETEMIN(Q)
  for each $v \in \text{Out}(u)$ **do** // relaxiere Kante (u, v)
    if $\delta[v] > \delta[u] + w([u, v\rangle)$ **then**
      $\delta[v] = \delta[u] + w([u, v\rangle)$
      DECREASEKEY($Q, v, \delta[v]$)
      $\pi[v] = u$

Dijkstra Algorithmus

Initialisierung: Für alle $x \in V$ setze $\delta[x] = \infty$, $\pi[x] = \text{NIL}$
Für Startknoten s setze $\delta[s] = 0$.
Bilde (Min) Prioritätenschlange Q für V.

while Q not emptry **do**
  u = DELETEMIN(Q)
  for each $v \in \text{Out}(u)$ **do** // relaxiere Kante (u, v)
    if $\delta[v] > \delta[u] + w([u, v\rangle)$ **then**
      $\delta[v] = \delta[u] + w([u, v\rangle)$
      DECREASEKEY($Q, v, \delta[v]$)
      $\pi[v] = u$

Laufzeit

- Initialisierung: $O(n)$
- While Loop: n DELETEMINS
  m Relaxierungen
  höchstens m DECREASEKEYS

Dijkstra Algorithmus

while Q not emptry **do**
  u = DELETEMIN(Q)
  for each $v \in$ Out(u) **do** // relaxiere Kante (u, v)
    if $\delta[v] > \delta[u] + w([u, v\rangle)$ **then**
      $\delta[v] = \delta[u] + w([u, v\rangle)$
      DECREASEKEY($Q, v, \delta[v]$)
      $\pi[v] = u$

Laufzeit

- Initialisierung: $O(n)$
- While Loop: n DELETEMINS $O(n \log n)$
  m Relaxierungen $O(m)$
  höchstens m DECREASEKEYS $O(m)$

Mit Hollow-Heaps DELETEMIN $O(\log n)$ und DECREASEKEY $O(1)$

Gesamtlaufzeit $O(m + n \log n)$

Minimale Aufspannende Bäume - Minimum Spanning Trees (MST)

Definition

Sei $G = (V, E)$ ein ungerichteter Graph mit reellen Kantengewichten $w : E \to \mathbb{R}$.

Ein Teilgraph ohne Zyklen heißt *Wald*. Seine Zusammenhangskomponenten heißen *Bäume*.

Ein maximaler Teilgraph ohne Zyklen heißt *aufspannender Wald*, bzw. *aufspannender Baum*, falls er nur aus einer Komponente besteht.

Ein aufspannender Baum (bzw. Wald) mit minimaler Kantengewichtssumme heißt *minimaler aufspannernder Baum (MST)*, bzw. *minimaler aufspannender Wald*.

Grundbegriffe

- Schnitt: Eine Partition von V in $S \dot{\cup} V \setminus S$ mit $1 \leq |S| < n$.
- Kreuzen: Liegt für $\{u, v\} \in E$ ein Endpunkt in S und der andere in $V \setminus S$, so kreuzt $\{u, v\}$ den Schnitt.
- Respektieren: Eine Kante, die einen Schnitt nicht kreuzt, respektiert ihn.
- Leichte Kante: Eine Kante, die den Schnitt kreuzt und minimales Gewicht hat.

Eigenschaften des MST

Eigenschaft des Schnitts:

Sei S eine Menge von Knoten von G mit $|S| > 1$ und $|S| < n$. Jeder minimale aufspannende Baum von G enthält eine leichte Kante $\{u, v\}$ mit u in S und v in $G \setminus S$.

Keine Zyklen:

Sei c ein einfacher Zyklus in G, und sei $\{u, v\}$ die Kante in c, die das höchste Gewicht hat. Dann ist $\{u, v\}$ nicht Teil eines minimalen aufspannenden Baums.

Grundidee der folgenden Algorithmen:

Wiederholtes Einfügen von leichten Kanten

Kruskal Algorithmus

Algorithm 1: Kruskal Algorithmus zum Finden eines MST:

```
1  A = ∅
2  for alle Knoten v von G do
3  |   Speichere Knoten v als separaten Baum
4  end
5  Sortiere die Kanten von G in aufsteigender Reihenfolge (Gewicht)
6  for alle Kanten {u, v} von G in aufsteigender Reihenfolge do
7  |   if u und v sind in in zwei verschiedenen Bäumen then
8  |   |   A = A ∪ {u, v}
9  |   |   Verbinde den Baum, der u enthält mit dem Baum, der v
   |   |    enthält, mit Kante {u, v}
10 |   end
11 end
12 Ergebnis: A
```

Kruskal Algorithmus

Der Algorithmus verwaltet einen Wald. Anfangs ist jeder Knoten ein Baum. Es werden immer wieder zwei Bäume vereinigt. Der Algorithmus muss feststellen können, ob zwei Knoten im gleichen Baum des derzeitigen Waldes liegen.

Darstellung der Bäume:

- Jeder Baum hat einen eindeutigen id zwischen 1 und n. Jeder Knoten v speichert den id $B[v]$ seines Baumes.
- Für jeden Baum können die Knoten in linearer Zeit aufgezählt werden (z.B. mit verketteter Liste) und jeder Baum speicher seine Größe.
- Test, ob u, v im gleichen Baum: $B[u] = B[v]$? ($O(1)$ Zeit)
- Zwei Bäume zusammenfügen: Setze die $B[]$-Werte im kleineren Baum auf den $B[]$ Wert des anderen. (insgesamt höchstens Zeit $O(n \log n)$, weil jeder Knoten höchstens $\log_2 n$ mal einen neuen $B[]$-Wert bekommen, da sich jedesmal die Baumgröße mindestens verdoppelt.

Damit Laufzeit des Kruskal Algorithmus $O(n \log n + m \log m)$.

Prim Algorithmus

- Beginnend mit Startknoten s lässt man einen Baum wachsen, indem man immer wieder eine leichte Kante findet, die aus diesem Baum herauskreuzt.
- Min-Priority Queue Q enthält Knoten v, die noch nicht im wachsenden Baum liegen, mit Priorität $k[v]$, nämlich der kürzesten Kante, die von v zum wachsenden Baum kreuzt.
- Algorithmus und Laufzeitanalyse ähnlich zu Dijkstra Algorithmus.
- Mit Verwendung von Hollow Heaps erzielt man eine Laufzeit von $O(m + n \log n)$.

Prim Algorithmus

Algorithm 2: Prim Algorithmus zum Finden eines MST:

```
1  for alle Knoten u von G do
2  |   k[u] = ∞
3  |   π[u] = NULL
4  end
5  k[s] = 0
6  Füge alle Knoten von G in Q ein
7  while Q ≠ ∅ do
8  |   u = EXTRACT-MIN(Q)
9  |   for jeden Nachbarn v von u in G do
10 |   |   if v ∈ Q und w({u, v}) < k[v] then
11 |   |   |   DECREASE-KEY(Q, v, w({u, v}))
12 |   |   |   π[v] = u
13 |   |   end
14 |   end
15 end
16 Ergebnis: Kanten { {v, π[v]} | v ∈ V \ {s} }
```