Graphen-Algorithmen

1 Grundlagen

1.1 Graphenrepräsentation

Es gibt hauptsächlich zwei Möglichkeiten, einen Graphen abzuspeichern:

1. **Adjazenmatrix**: Der Graph mit n Knoten wird in einer n x n-Matrix gespeichert, wobei der Eintrag $a_{i,j}$ genau dann eine 1 ist, wenn es vom Knoten *i* zum Knoten *j* eine Kante gibt.
2. **Adjazenlistenrepräsentation**: Hierbei gibt es eine Liste der Länge n mit einem Eintrag für jeden Knoten. An der Stelle *i* ist dann eine Liste mit allen Knoten *k*, für die es eine Kante von *i* zu *k* gibt, gespeichert.

Je nach Anwendung eignet sich die eine oder andere Repräsentation besser.

1.2 Traversierung

Zur Traversierung eines zusammenhängenden Graphen kann wie folgt vorgegangen werden: Es wird ein Knoten des Graphen ausgewählt, bei dem die Traversierung beginnt. Man verwaltet nun eine Menge von Knoten, die den *Rand* bilden. Dort sind alle Punkte gespeichert, welche eine Kante zu einem unbesuchten Knoten besitzen können. Zusätzlich muss man sich merken, welche Knoten bereits besucht wurden. Dann kann folgendermassen vorgegangen werden:

```
procedure Traversiere G=(V,E) ab Knoten b
B = {b}; R = {b};
while (R.isEmpty() == false) {
    wähle Knoten v aus R
    if (es gibt keine unbenutzen Kanten (v,v') in E) {
        lösche v aus R
    } else {
        if (v' unbesucht) {
            B = B + v';
            R = R + v';
        }
    }
}
```

Für *B* kann natürlich ein Bitarray eingesetzt werden, während man für den Rand *R* entweder ein Stack oder eine Queue wählen kann. Entscheidet man sich für einen Stack, so erhält man eine Tiefensuche, andernfalls eine Breitensuche.

1.3 Topologisches Sortieren

Gegeben ist ein azyklischer Graph mit n Knoten und m Kanten. Man soll nun jedem Knoten eine Nummer $nr(i)$ zuweisen, sodass folgende Bedingung gilt:

$$(i,j) \in E \rightarrow nr(i) < nr(j)$$

Dazu kann man folgende Überlegung machen: Wenn man stets einen Knoten mit in-degree 0 (also einen, ohne eingehende Kante) aus dem Graphen entfernt, und diesem die nächstgrössere Nummer gibt, erhält man schlussendlich eine topologische Sortierung.

Eine konkrete Implementierung könnte folgendermassen aussehen: Jeder Knoten erhält einen Zähler, welcher die Anzahl eingehende Kanten quantifiziert. Um diesen Zähler zu initialisieren, benötigt man $O(n+m)$ Zeit. Zusätzlich verwaltet man eine lineare Liste mit allen Knoten, deren Zähler 0 ist. Nun kann man solange diese Liste noch Elemente enthält, ein beliebiges entfernen, und diesem die nächstgrössere Nummer zuweisen. Zusätzlich folgt man allen Pfeilen dieses Knotens (was mit der Adjazenzlistenrepräsentation sehr einfach ist) und verkleinert die Zähler aller dieser Knoten. Wird ein Zähler 0, so wird dieser in die Liste aufgenommen.

Da jeder Knoten und jede Kante höchstens ein Mal bearbeitet wird, ergibt sich so eine totale Laufzeit von linearer Grösse im Input: $O(n+m)$.

2 Kürzeste Wege

2.1 Floyd-Warshall Algorithmus

Um in einem gewichteten, gerichteten Graph den kürzesten Pfad zwischen allen (!) Knotenpaaren zu finden, kann der Floyd-Warshall Algorithmus eingesetzt werden. Die Laufzeit des Algorithmus beträgt $O(n^3)$ und ist ein Beispiel für dynamische Programmierung.

Der Algorithmus geht induktiv vor, und berechnet in jedem Schritt den kürzesten Pfad zwischen allen Knotenpaaren unter Verwendung der ersten k Knoten als Zwischenstationen. Es gibt eine Funktion $shortPath(i, j, k)$, welche den kürzesten Pfad vom Knoten *i* zum Knoten *j* zurückgibt, wenn nur die ersten *k* Knoten als Zwischenstationen verwendet werden dürfen. *k* wird dann schrittweise erhöht, wobei bei jedem Schritt folgende Überlegung gilt: Es gibt genau zwei Kandidaten, für den kürzesten Pfad von i nach j über k Zwischenstationen, entweder war der Pfad bereits zuvor optimal, oder der kürzeste Pfad führt von i über k, und von dort zu j (wobei nur die ersten $k-1$ Knoten benutze werden). Das ergibt direkt folgende rekursive Formel:

$$shortPath(i, j, k) = min\{\, shortPath(i, j, k-i), shortPath(i, k, k-1) + shortPath(k, j, k-1) \,\}$$

$$shortPath(i, j, 0) = edgeCost(i, j)$$

Ein solches Problem lässt sich trivial mit dynamischer Programmierung lösen:

```
/* A 2-dimensional matrix. At each step in the algorithm,
   path[i][j] is the shortest path from i to j using
   intermediate vertices (1..k-1). */
int path[][];

for i = 1 to n
    for j = 1 to n
        path[i][j] = edgeCost(i,j); // or +inf if no edge exists
    endfor
endfor

for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            path[i][j] = min(path[i][j], path[i][k] + path[k][j]);
        endfor
    endfor
endfor
```

2.1.1 Transitive (und reflexive) Hülle

Mit einem identischen Ansatz kann man auch die transitive Hülle eines Graphen berechnen: Dazu wird in der Initialisierung überall eine 1 gesetzt, wo es einen Kante gibt, und 0 sonst. Soll die die Relation zusätzlich auch noch reflexiv sein, so werden auf der Diagonale lauter 1 gesetzt.

Dann wird wiederum per Induktion vorgegangen: Für die ersten *k* Knoten ist die transitive Hülle bereits gefunden, und um die Relation um den Knoten k+1 zu erweitern, muss man für alle die Kante (i,j) genau dann hinzufügen, wenn entweder (i,j) bereits Teil des Graphen war, oder wenn es sowohl (i,k) und (k,j) bereits gibt. Anders gesagt ersetzt man die Addition durch ein logisches AND, und die Minimumsoperation wird durch ein OR ersetzt.

2.2 Dijkstra-Algorithmus

Der Algorithmus von Dijkstra dient der Berechnung aller kürzesten Pfade von einem einzelnen Startknoten aus in einem kantengewichteten Graphen. Die funktioniert aber nur bei positiven Kantengewichten.

Es wird dabei folgendermassen vorgegangen: Man verwaltet einen „Rand“ und die Menge der bereits besuchten Knoten. Im Rand befinden sich Knoten, die noch nicht besucht sind, aber von einem besuchten Knoten aus erreicht werden können. Solange der Rand noch nicht leer ist, wird folgendermassen vorgegangen:

- Entferne das Minimum aus dem Rand, denn zu diesem Knoten kommen wir sicher nicht mehr günstiger. Der Knoten wird als besucht markiert.
- Dann werden alle Nachbarn des inspiziert. Ist der Nachbar bereits im Rand und der Wert des jetzigen Knotens plus das Kantengewicht der Verbindung zusammen kleiner als der Wert des Nachbars, wird der Wert verkleinert. Ist der Wert bereits kleiner oder gleich gross, muss nichts gemacht werden. Ist der Nachbar hingegen noch nicht im Rand, so wird dieser Knoten neu eingefügt, und erhält den Wert des jetzigen Knotens + das Kantengewicht.

Zu Beginn wird der Rand mit den Nachbarn des Startknotens initialisiert. Alternative kann anstelle eines Randes auch die Menge aller unbesuchten Knoten verwaltet werden. Zu Beginn werden dann einfach alle Werte auf unendlich gesetzt.

Die Distanz und ob ein Knoten besucht ist oder nicht kann direkt bei den Knoten gespeichert werden, oder man verwendet ein zusätzliches Array dafür. Um den Rand zu repräsentieren eignet sich am besten ein Fibbonacci-Heap, was in einer schlussendlichen Performance von $O\big(m + nlog(n)\big)$ resultiert.

2.3 Ford-Bellmann

Mit einer Laufzeit von $O(m \cdot n) = O(n^3)$ ist der Algorithmus von Ford-Bellmann zwar nicht der Schnellste, insbesondere langsamer als Dijkstra's Lösung, dafür aber können Kantengewichte auch negativ sein. Gibt es negative Zyklen, so ist der Begriff „kürzester Pfad“ zwar nicht sinnvoll zu definieren, mit dem Algorithmus von Ford-Bellmann lassen sich solche negativen Zyklen jedoch erkennen. Wie bei Dijkstra werden auch hier alle kürzesten Pfade von einem Startknoten aus gefunden.

Das Verfahren ist sehr simple: Es gibt ein Array d, indem die kürzeste Distanz zum jeweiligen Knoten gespeichert wird. Nun wird einfach $n-1$ mal über alle Kanten iteriert, wobei mit zuerst nur Pfade der Länge 1, dann 2 usw. betrachtet werden. Da es höchstens $n-1$ Kanten in einem Pfad haben kann, liefert der Algorithmus stets das korrekte Ergebnis. Will man zusätzlich wissen, ob es negative Zyklen gibt, iteriert man bis n. Verändert sich in diesem letzen Schritt noch etwas, gibt es mindestens einen negativen Zyklus.

```
// init
for i = 0 to n
    if (i is source) d[i] = 0
    else             d[i] = infinity
endfor

// algorithm
for i = 1 to n-1
    forall (u,v) in E
        if (d[u]+value(u,v) < d[v])
            d[v] = d[u]+value(u,v)
            p[v] = u // optional
        endif
    endfor
endfor
```

Es ist einfach, auch gleich noch den kürzesten Pfad zu berechnen, anstelle nur die Länge dieses Pfades. Dazu speichert man einfach in einem zweiten Array p den Vorgänger jedes Knotens.

2.4 A*-Suchalgorithmus

Will man in einem Graphen den kürzesten Pfad von einem gegebenen Startknoten zu einem Zielknoten finden, so sucht man möglicherweise grosse Teile des Graphs ab, welche sehr weit vom Zielknoten entfernt sind, und damit gar nicht in Frage kommen. Insbesondere in Zustandsgraphen, die riesig sein können, macht es Sinn, die Suche einzuschränken. Dazu erkundet man den Graph, und zu jedem Zeitpunkt hat man eine gewisse Menge an Knoten bereits gesehen, und kennt die Distanz zu diesen, während andere unentdeckt sind. Für gewisse Knoten kennt man auch erste eine obere Schranke (bei Dijkstra sind diese Knoten im Rand) für die Kosten, $h(x)$. Anstelle nun wie bei Dijkstra, immer beim Knoten zu expandieren, der am nächsten beim Startpunkt liegt, wird nun eine heuristische Funktion $g(x)$ benutzt, die eine **obere** Schranke für die Distanz von x zum Zielknoten darstellt. Dabei sollte die Funktion möglichst genau sein, darf aber unter keinen Umständen den korrekten Wert überschreiten.

Nun kann man jeweils beim Knoten mit dem kleinsten Wert für $h(x) + g(x)$ expandieren, und so die Suche zielgerichteter durchführen. Dieser Algorithmus ist optimal, d.h. er findet immer eine optimale Lösung, falls eine solche existiert.

3 Minimale Spannbäume (MST)

3.1 Verfahren

Ein Verfahren, um minimale Spannbäume in einem gewichteten Graphen zu finden, funktioniert greedy:

Jede Kante hat zu jedem Zeitpunkt einen Status. Entweder ist sie gewählt, verworfen oder noch unentschieden, wobei zu Beginn alle Kanten unentschieden sind. Weiter soll während des ganzen Verfahrens folgende Invariante gelten:

$$\exists \text{ MST } T \text{ von } G\colon T \text{ enthält alle gewählten, aber keine verworfenen Kanten}$$

Ein Schnitt in einem Graphen $G = (V, E)$ sei nun eine Aufteilung von V in S und $\bar{S} = V \setminus S$. Eine Kante $e \in E$ kreuzt diesen Schnitt nun, wenn der eine Endpunkt in S und der andere in $\bar{S}$ liegt. Das Verfahren befolgt nun ständig eine der folgenden Regeln, bis der MST gefunden wurde:

1. Betrachte einen Schnitt, den keine schon gewählte Kante kreuzt. **Wähle** eine kürzeste Kante unter allen unentschiedenen Kante, welche diesen Schnitt kreuzen.
2. Wähle einen einfachen Zyklus (einen Zyklus also, der sich selbst nicht „kreuzt“), der keine verworfene Kante enthält. **Verwerfe** unter den unentschiedenen Kanten die längste.

3.2 Implementierungen

3.2.1 Kruskal

Es werden alle Kanten in aufsteigend sortierter Reihenfolge betrachtet. Verbindet die gerade betrachtete Kante zwei unabhängige Teilbäume, so nehmen wir die Kante. Andernfalls, also wenn beide Endpunkte im selben Teilbaum liegen, verwerfen wir die Kante. Hinweis: Zu Beginn gibt es n Teilbäume der Grösse 1, bestehend aus nur einem Knoten.

```
MST = {}
sortiere E aufsteigend
für alle v aus V: makeSet(v)
für alle (v,w) aus E, aufsteigend
    if (find(v) != find(w))
        MST = MST + {(v,w)}
        union(find(v),find(w))
    endif
endfür
```

Benötigt wird also eine einfach Union-Find-Datenstruktur, um den Algorithmus von Kruskal zu implementieren.

Laufzeit: $O(m \log(n))$

3.2.2 Prim/Dijkstra

Eine weitere Möglichkeit, dieses Verfahren zu implementieren benutzt nur die Regel 1. Es werden also nirgends Kanten explizit ausgeschlossen: Man verwaltet eine Menge von erledigten Knoten, wobei es nur gewählte Kanten zwischen erledigten Knoten gibt. Damit gibt es immer einen Schnitt zwischen den erledigten und unerledigten Knoten, wobei keine gewählte Kante diesen Schnitt überquert. Folglich kann also die kürzeste Kante, welche diesen Schnitt kreuzt, gewählt werden.

```
V' = {v} // Menge der erledigten Knoten
MST = {}

while |V'| < n
    wähle kürzeste Kante (u,v) mit u in
     V', v aber nicht.

    MST = MST + {(u,v)}
    V' = V' + {v}
endwhile
```

Konkret wird nicht die Menge der erledigten Knoten verwaltet, sondern man hält alle unerledigten Knoten, die aus V' erreichbar sind, in einem Fibbonacci Heap. Dann kann man jeweils das Minimum aus dem Heap entfernen und diese Kante zum MST hinzufügen. Zudem muss man nun alle Nachbarn b des gerade betrachteten Knotens a ansehen, wobei es zwei Fälle gibt:

1. Der Knoten b ist bereits im Heap, aber der Wert von b im Heap ist grösser, als das Gewicht der Kante (a, b). Dann muss der Wert von b im Heap verringert werden.
2. Der Knoten b ist nicht im Heap, und ist nun (neu) erreichbar aus V'. b wird also eingefügt, mit den Wert der Kante (a, b).

Pseudocode:

```
MST = {}  // Der minimale Spannbaum
H = {}    // Der Heap, welcher alle Knoten enthält, die aus dem MST erreichbar sind
parent[]  // In parent[a] ist für einen nichterlediten Knoten, welcher einen
          // Anschluss an den MST hat, gespeichert, mit welchem Knoten des MST er
          // am günstigsten an den MST angeschlossen werden kann.

/* Hinweise:
   -  value(u,v) gibt den Wert der Kante (u,v) zurück
   -  H.insert(v, val) fügt den Knoten v mit Wert val in den Heap ein
   -  H[z].value greift auf z im Heap zu, und liefert den Wert von z zurück.
      Damit dies möglich ist, benötigt man ein Array mit Pointern in den Heap
      -> nasty hack :)
*/

// init
for all neighbors u of v
    H.insert(u, value(u,v))
    parent[u] = v
endfor

// algorithm
while not H.isEmpty()
    z = H.deleteMin()

    MST = MST + {(parent(z),z)}

    for all neighbors u of z
        if (H.contains(z)) then
            if (value(z,u) < H[z].value) then
                H.decrease_key(z,value(z,u))
            endif
        else
            H.insert(u,value(z,u))
        endif
    endfor
endwhile
```