Einführung in die Informatik

Universität Stuttgart, Studienjahr 2005/06

Gliederung der Grundvorlesung

1. ~~Einführung in die Sprache Ada95~~
2. ~~Algorithmen und Sprachen~~
3. ~~Daten und ihre Strukturierung~~
4. ~~Begriffe der Programmierung~~
5. ~~Abstrakte Datentypen~~ Zurückgestellt
6. ~~Komplexität von Algorithmen und Programmen~~
7. ~~Semantik von Programmen~~
8. ~~Suchen~~
9. ~~Hashing~~
10. ~~Sortieren~~
11. Graphalgorithmen
12. Speicherverwaltung

Graphen

Viele zu verarbeitende Informationen besitzen eine Graph-Struktur: Sie bestehen aus Teilen (dies entspricht den Knoten), die untereinander in Beziehung stehen (dies entspricht den Kanten), wobei die Beziehungen gewichtet sind oder Kommunikation ermöglichen und die Informationen Attribute tragen oder wiederum aus Graphen bestehen (markierte Graphen, hierarchische Graphen).
Alle erforderlichen Definitionen über Graphen finden Sie in 3.7 und 3.8., sowie 8.2 und 8.8. Die Adjazenzdarstellung steht in 3.8.6; Graphdurchläufe (zunächst GD genannt, nun als Tiefen- und als Breitendurchlauf bezeichnet) finden Sie in 3.8.7 und 8.8.9. In 6.5.4 wird die transitive Hülle eines gerichteten Graphen in $O(n^3)$ Schritten berechnet.
Wir gehen davon aus, dass Sie die grundlegenden Begriffe bereits früher eingeübt haben und kennen.

11. Graphalgorithmen

Ziele des 11. Kapitels:

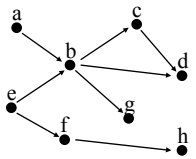
Gewisse Verfahren auf Graphen werden in den unterschiedlichsten Anwendungen gebraucht. Den Graph- und den Baumdurchlauf haben wir bereits kennen gelernt und ihn auf die Ermittlung der Zusammenhangskomponenten angewendet (DFS und BFS, inoder usw., Baumsortieren, siehe z.B. 3.7.7 und 8.8.9).

In diesem Kapitel lernen Sie weitere Verfahren, und zwar:
- ordne eine Halbordnung linear,
- finde den kürzesten Weg in einem kantenmarkierten Graphen,
- finde das minimale Gerüst in einem Graphen,
- kombiniere die Kanten eines Graphens optimal (= löse ein spezielles Zuordnungsproblem).

Sie sollen hierfür Lösungsalgorithmen beschreiben und deren Komplexität herleiten können. Zugleich sollen Sie die Korrektheit dieser Verfahren begründen können.

11.1 Topologisches Sortieren

Topologisches Sortieren ist das Einbetten einer Halbordnung in eine totale (= lineare) Ordnung. Die Halbordnung ist durch einen azyklischen Graphen und seine transitive Hülle gegeben.



Angabe zweier Funktionen

ord, vgl. Def. 8.8.2:

a	b	c	d	e	f	g	h
1	4	5	7	2	3	6	8

a	b	c	d	e	f	g	h
4	5	7	8	1	2	6	3

Es gibt noch weitere Anordnungen ord für dieses Beispiel.

Hilfssatz 11.1.1:

Es sei G ein DAG, dann gibt es mindestens einen Knoten mit Eingangsgrad 0 und mindestens einen Knoten mit Ausgangsgrad 0.

Diese Aussage ist "klar": Folge von irgendeinem Knoten den auslaufenden Kanten. Da man in einem azyklischen Graphen nicht wieder auf einen bereits besuchten Knoten stoßen darf, muss jede Kantenfolge nach spätestens n-1 Schritten in einem Knoten mit Ausgangsgrad 0 enden. Analog beim Rückwärts-Verfolgen von Kanten.

(DAG = gerichteter azyklischer Graph, siehe 3.8)

Satz 11.1.2: Ein gerichteter Graph besitzt genau dann eine topologische Sortierung, wenn er azyklisch ist.

"⇒": Ein Graph möge eine topologische Sortierung ord: $V \to \mathbb{N}$ mit $\forall u,v \in V$ mit $u \neq v$ gilt: $(u,v) \in E^* \Rightarrow ord(u) < ord(v)$ besitzen. Betrachte dann einen geschlossenen Weg in G: $(u_0, u_1, ..., u_{k-1}, u_0)$ mit $k \geq 0$. Wegen $(u_0, u_0) \in E^*$ muss dann $ord(u_0) < ord(u_0)$ gelten, Widerspruch.

"⇐": Setze i = 1. Ein gerichteter azyklischer Graph besitzt mindestens einen Knoten u mit dem Eingangsgrad 0. Setze ord(u) = i, erhöhe i um 1, entferne den Knoten u und die zu ihm inzidenten Kanten (es entsteht ein gerichteter azyklischer Graph G') und fahre rekursiv mit G' fort. So erhält jeder Knoten u aus G eine Nummer ord(u). Wenn nun $(u,v) \in E^*$ in G gilt, so kann der Eingangsgrad von v bei dieser Konstruktion erst dann 0 werden, wenn der Knoten u bereits entfernt ist. Dies heißt aber: ord(u) < ord(v).

11.1.3: Der Beweis liefert zugleich einen *Algorithmus zur Berechnung einer topologischen Sortierung*:

```
i := 0;
for j in 1..n loop
    wähle einen Knoten u mit Eingangsgrad 0;
    i:=i+1;  setze ord(u) := i;
    entferne u aus dem Graphen;
end loop;
```

Man beachte, dass sich der Eingangsgrad der Knoten bei jedem Entfernen eines Knotens um 1 verringern kann.

Da es mehrere Knoten mit dem Eingangsgrad 0 geben kann und da jede Auswahl eines solchen Knotens zu einer topolo-gischen Sortierung führt, sind topologische Sortierungen in der Regel nicht eindeutig.

Probleme bei diesem Algorithmus:
- Wie findet man am Anfang rasch einen Knoten mit dem Eingangsgrad 0? (siehe Programmstück unten)
- Wie aktualisiert man die Eingangsgrade beim Entfernen eine Knotens?

(Zur Adjazenzdarstellung von Graphen siehe 3.8.6.)

```
p := Anfang;              -- Die Eingangsgrade werden in zahl1 gespeichert
while p /= null loop p.zahl1 := 0; p := p.NKn; end loop;
p := Anfang;              -- Gehe alle Kanten durch und erhöhe den
while p /= null loop  edge := p.EIK;   -- Eingangsgrad des Zielknotens
   while  edge /= null  loop
      edge.EKn.zahl1 := edge.EKn.zahl1 + 1;
      edge := edge.NKa;
   end loop;
   p := p.NKn;
end loop;
```

Dieses Programmstück ermittelt alle Eingangsgrade in O(n+m) Schritten.

Um schnell einen Knoten mit Eingangsgrad 0 finden zu können, speichern wir alle diese Knoten in einer Liste (oder einem Feld) "*GradNull*":

```
p := Anfang; "Setze GradNull auf die leere Liste";
while p /= null loop
    if p.zahl1 = 0 then "füge p an GradNull an" end if;
    p := p.NKn;  end loop;
```

Nun müssen wir noch die Eingangsgrade aktualisieren, sobald ein Knoten u aus dem Graphen entfernt wird.

Hierzu erniedrigen wir die Eingangsgrade aller Knoten, die über die Kantenliste von u erreichbar sind, um 1. Falls hierbei ein Eingangsgrad auf 0 absinkt, wird der entsprechende Knoten in *GradNull* aufgenommen. (u sei der Verweis auf den als nächstes zu entfernenden Knoten mit Eingangsgrad 0.) Dies liefert:


```
edge := u.EIK;
while  edge /= null  loop
    edge.EKn.zahl1 := edge.EKn.zahl1 - 1;
    if edge.EKn.zahl1 = 0
        then "füge edge.EKn an GradNull an" end if;
    edge := edge.NKa;
end loop;
"entferne den Knoten u aus dem Graphen G"
```

Die restlichen Details überlassen wir den Leser(inne)n. (Könnte z.B. zahl1 irrtümlich kleiner als 0 werden, so dass die Abfrage "if edge.EKn.zahl1 = 0 ..." übergangen wird?) Zur Datenstruktur: Entweder arbeitet man auf einer Kopie von G und trägt zusätzlich zahl1 im Original ein oder man verwendet einen Booleschen Wert, um zu simulieren, dass man den jeweiligen Knoten gelöscht hat.

11.1.4: Komplexität dieses Algorithmus:

Zeitkomplexität des topologischen Sortierens: O(n+m).
Platzkomplexität O(n) (bei Verwendung eines Booleschen Wertes).

Begründung zur Zeitkomplexität:

Jeder Knoten erhält seinen Eingangsgrad: O(n+m).

Aufbau der Liste *GradNull*: O(n).

n mal: Einen Knoten mit Eingangsgrad 0 finden (= nimm stets den ersten Knoten in der Liste *GradNull*) und später entfernen, insgesamt: O(n).

m mal insgesamt: Eingangsgrade erniedrigen und Knoten mit Eingangsgrad 0 an *GradNull* anhängen: O(n+m).

Alle übrigen Operationen (i := i+1; setze ord(u) := i; usw.) erfordern insgesamt höchstens O(n) Schritte.

11.1.5: Wir betrachten folgenden Algorithmus TopSort, der in O(n+m) Schritten arbeitet und im Wesentlichen gleich der Tiefensuche ist (n = Zahl der Knoten, m = Zahl der Kanten):

```
procedure TS(u: NextKnoten) is                -- nummer ist global
begin  u.Besucht := true;
    for alle Nachfolgerknoten v von u loop
        if not v.Besucht then TS(v); end if; end loop;
    u.zahl2 := nummer;  nummer := nummer - 1;
end;
...
for alle Knoten p loop  p.Besucht := false;  end loop;
nummer := n;
for alle Knoten p loop
        if not p.Besucht then TS(p); end if;  end loop;
```

Hinweis: Diese zahl2-Werte heißen in der Literatur auch "finish"-Werte.

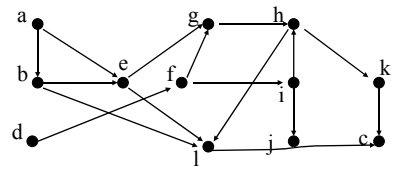
Dieser Algorithmus macht Folgendes:

Er beginnt in einem beliebigen Knoten und führt einen Tiefendurchlauf durch. Immer wenn er hierbei alle von einem Knoten ausgehenden Kanten abgearbeitet hat, ordnet er diesem Knoten die Zahl der Variablen "nummer" zu. nummer wird dann um 1 erniedrigt. Somit erhält jeder Knoten eine kleinere Nummer als alle seine Nachfolger. Da nummer mit n initialisiert wird, bekommt ein Knoten ohne Nachfolger den größten Wert.

Machen Sie sich klar, dass diese Nummern als ord-Werte verwendet werden können, da eine Ordnungsfunktion ord in einem DAG genau dann zu einer topologischen Sortierung gehört, wenn für jeden Knoten x gilt, dass ord(x) < ord(y) für alle Nachfolgerknoten y, d.h. für alle $(x,y) \in E$, ist.

Also berechnet der Algorithmus eine topologische Sortierung.

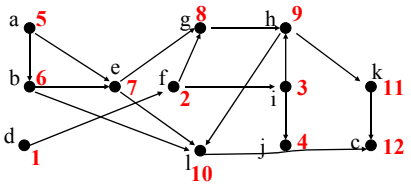
Beispiel: Azyklischer Graph mit n = 12 Knoten und m = 16 Kanten



Die Reihenfolge der Knoten in der Adjazenzliste sei k, e, h, a, f, j, i, d, c, b, g, l.

Dann liefert der obige Algorithmus TopSort folgende Zahlen für die Knotenkomponente *zahl2*, sofern man beim Nachfolger zuerst der Kante folgt, die bzgl. der Zahlen auf einer Uhr die kleinste ist.

Beispiel: Azyklischer Graph mit n = 12 Knoten und m = 16 Kanten



Knotenreihenfolge: k, e, h, a, f, j, i, d, c, b, g, l

Frage: Was liefert der Algorithmus für Werte für beliebige gerichtete Graphen?
Antwort: Man kann die Kanten dann in Kanten, die vom Algorithmus zu einem neuen Knoten durchlaufen werden (sog. Baum-Kanten) und andere Kanten unterteilen, wobei sich letztere wiederum in Rückwärts-, Vorwärts- und Quer-Kanten einteilen lassen. Siehe Algorithmik-Vorlesungen.

11.2 Kürzeste Wege

Gegeben sei ein gerichteter Graph G = (V, E, δ) mit $\delta: E \to \mathbb{R}^{\geq 0}$ (= Menge der nichtnegativen reellen Zahlen) und ein Knoten $u \in V$.
Gesucht werden alle kürzesten Entfernungen δ(u,v) für alle Knoten $v \in V$.
Wir wollen also SSSP lösen (8.8.5).

Hierfür verwendet man in der Regel den Dijkstra-Algorithmus, der sich wie eine Breitensuche (gewichtet bzgl. der Entfernungsfunktion δ) über den Graphen vorarbeitet. Zeitkomplexität bei Implementierung mit einem Heap: O((n+m)·log(n)).

Hinweis: Sucht man die kürzesten Abstände zwischen *allen* Knoten (APSP), so kann man den Dijkstra-Algorithmus für jeden Knoten einmal durchführen oder den Floyd-Algorithmus mit Zeitkomplexität $\Theta(n^3)$ verwenden. Dieser Algorithmus ähnelt stark dem Warshall-Algorithmus in 6.5.4 und ist in Lehrbüchern gut beschrieben.

(Edgar W. Dijkstra und R. W. Floyd: holländischer bzw. amerikanischer Wissenschaftler, "Pioniere" der Informatik. Ihre Algorithmen wurden 1959 und 1962 veröffentlicht. <Aussprache dieser Namen: "Deikstra" und "Fleut">)

11.2.1 Aufgabenstellung (SSSP)

SSSP = single source shortest paths
= alle kürzesten Wege, die von einem gegebenen Knoten zu jedem anderen Knoten führen.

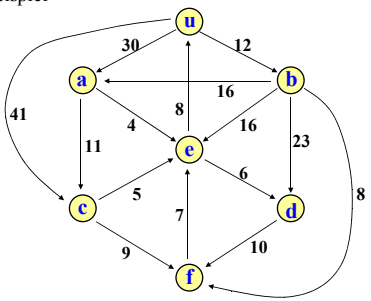
Gegeben ist ein gerichteter Kanten markierter Graph G = (V, E, δ) mit $\delta: E \to \mathbb{R}^{\geq 0}$ und ein "Startknoten" $u \in V$ (u ist "single source", also die einzige Quelle, von der ausgehend alle kürzesten Wege berechnet werden sollen; $\mathbb{R}^{\geq 0}$ ist die Menge der nichtnegativen reellen Zahlen).
Gesucht werden für jeden Knoten $v \in V$ ein kürzester Weg von u nach v einschließlich seiner Länge dist(u,v) = Summe aller δ((u,v)) für alle Kanten (u,v), aus denen dieser Weg besteht.

Zuerst zu klären: Wie sollen diese Wege dargestellt werden? Würde man alle Wege tatsächlich hinschreiben, so benötigt man $O(n^2)$ Speicherplatz, da jeder doppelpunktfreie Weg von u zu einem Knoten v bis zu n Knoten enthalten kann.

Vorschlag: Man speichert zu jedem Knoten v nur den Knoten, der auf einem kürzesten Weg von u nach v direkter Vorgänger von v ist. Dann kann man (rückwärts gehend) schrittweise einen kürzesten Weg von hinten nach vorne zu jedem Knoten aufbauen.

11.2.2 Beispiel

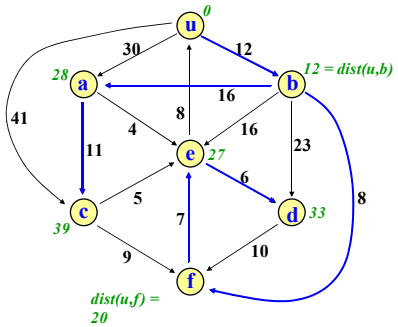
n=7, m= 15



Dieser Graph ist stark zusammenhängend. Startknoten sei u.
Wir konstruieren nun einen Teilgraphen mit kürzesten Wegen.

Beispiel:

n=7, m= 15

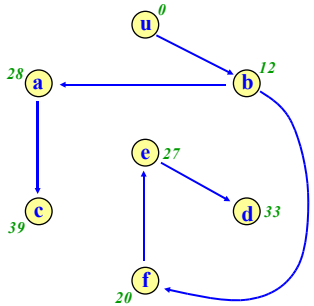


Der Teilgraph mit den kürzesten Wegen bildet einen Baum!

Beispiel:

n=7, m= 15

Dies ist ein "Kürzeste-Wege-Baum" für den Knoten u

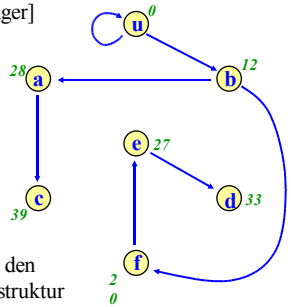


Also stellen wir kürzeste Wege bzgl. u wie folgt dar:

Schema für die Darstellung kürzester Wege:
(Knoten, Vorgängerknoten, Distanz von u)
[für u nehmen wir u als Vorgänger]

Hieraus kann man kürzeste Wege rückwärts rekonstruieren.

(u, u, 0),
(a, b, 28),
(b, u, 12),
(c, a, 39),
(d, e, 33),
(e, f, 27),
(f, b, 20).



Diese Werte schreibt man oft in den Graphen hinein, d.h., die Datenstruktur "Knoten" muss dann um entsprechende Komponenten erweitert werden.

11.2.3 Hinweis: Im Allgemeinen gibt es mehrere kürzeste Wege zwischen Knoten.

Wir interessieren uns hier immer nur für genau *einen* kürzesten Weg, der beliebig ausgewählt werden darf. Unter dieser Annahme bilden die kürzesten Wege einen Baum.
Grund: Wenn in diesem Teilgraphen zwei verschiedene Wege von einem Knoten x zu einem Knoten y führen würden, dann lägen mindestens zwei kürzeste Wege vor, von denen wir einen (durch Entfernen mindestens einer Kante) streichen. Entfernt man auf diese Weise alle doppelten Wege, so entsteht schließlich der genannte "Kürzeste-Wege-Baum" von u zu allen erreichbaren Knoten. (Dieser ist nicht eindeutig, aber für jeden Startknoten u sind selbstverständlich die Entfernungen zu allen Knoten in jedem zu u gehörenden Kürzeste-Wege-Baum gleich.)

An obigem Beispiel sieht man die Vorgehensweise, um einen Kürzeste-Wege-Baum zu einem Knoten u zu konstruieren:

Annahme, man hat bereits zu einer Menge von Knoten B je einen kürzesten Weg mit Distanz $d(v) = \text{dist}(u,v)$ für jedes $v \in B$ berechnet, dann wähle man als nächstes einen Knoten x ($x \notin B$), der die geringste Distanz von u besitzt, für den also gilt:
$d(v) + \delta((v,x)) \le d(v) + \delta((v,y))$ für alle Knoten $v \in B$, $y \notin B$.
Setze $d(x)$ = "dieses Minimum $d(v) + \delta((v,x))$", füge x zu B hinzu und fahre rekursiv fort, bis alle Knoten betrachtet wurden.
Initialisierung: $B = \{u\}$ mit $d(u) = 0$.

Dieses Vorgehen wird als Dijkstra-Algorithmus bezeichnet.
Problem hierbei: Finde einen solchen Knoten x möglichst schnell. Man speichert hierzu die als nächstes zu betrachtenden Knoten in einer "Prioritätswarteschlange". Dann gibt es drei Grundoperationen, die wir nun vorstellen.

11.2.4 Struktur des Dijkstra-Algorithmus

Grundlegende Mengen und Funktionen (blau gefärbt):

Gerichteter Graph $G = (V, E, \delta)$ mit $\delta: E \to \mathbb{R}^{\ge 0}$ und Knoten u.
B = Menge der bereits abschließend "bearbeiteten" Knoten.
R = Menge der "Randknoten", die von B aus erreichbar sind.
U = Menge der übrigen, bisher noch "unsichtbaren" Knoten.
Es gilt stets: $V = B \cup R \cup U$ (und B, R, U sind paarweise disjunkt).
Initialisierung: $B = \{u\}$, $R = \{v \mid (u,v) \in E\}$, $U = V - R - \{u\}$.
$D: V \to \mathbb{R}^{\ge 0}$ mit $D(v) =$ *vorläufiger Wert* für die kürzeste Entfernung von u zum Knoten v (stets ist $D(v) \ge \text{dist}(u,v)$).
Aber: Für die Knoten v in B wird gelten: $D(v) = \text{dist}(u,v)$.
Man stellt D als array (V) of real dar mit der Initialisierung D(v) = maximal darstellbare reelle Zahl, außer D(u) = 0.0.
Vorg: array (V) of V(mit null) gibt zu jedem Knoten den Vorgänger auf einem kürzesten Weg an. Initialisierung mit null.

In jedem Schritt wird ein Knoten x aus R mit seinem D-Wert in die Menge B aufgenommen, die Menge R wird um alle Knoten v aus U, die von x aus erreichbar sind, erweitert und die von x aus erreichbaren Knoten y in R werden darauf überprüft, ob über x ein kürzerer Weg von u nach y existiert.

Hierfür verwenden wir folgende drei Grundoperationen, wobei der D-Wert als "Schlüssel" (key) verwendet wird:

DELETEMIN = Finde und liefere x mit minimalem D-Wert, entferne x aus R, füge x zu B hinzu.

INSERT (y) = Berechne für y den D-Wert und füge y in R ein.

DECREASEKEY (y) = Wenn ein kürzerer Weg von u über den Knoten x nach $y \in R$ existiert, so berechne das kleinere D(y) und positioniere y neu in R.

Dies liefert dann folgendes Verfahren (man sieht rasch, dass man die Menge U nicht explizit braucht):

```
B := {u};  R := {v | (u,v) ∈ E};
while R ≠ ∅ do
    X := DELETEMIN;
    for all  Y ∈ S(X) and Y ∉ B  do
        if  Y ∉ R  then  INSERT(Y)
        else  DECREASEKEY(Y)  fi
    od
od;
```

Hierbei ist $S(x) = \{y \mid (x,y) \in E\}$ die Menge der Nachfolgerknoten ("successors") eines Knotens x im gegebenen Graphen.

Um dies entsprechend der Bedeutungen der Grundoperationen in ein Ada-Programm umzusetzen, muss die Datenstruktur für R festgelegt werden. Alles Übrige ergibt sich hieraus.

11.2.5 Prioritätswarteschlange

Eine "abstrakte" Datenstruktur, in der diverse Elemente auf ihre Abarbeitung warten (wobei sich deren Priorität nach einem Schlüssel aus einer geordneten Menge richtet), in die jederzeit Elemente eingefügt werden dürfen und aus der jederzeit das Element mit dem kleinsten Schlüssel entfernt werden kann, bezeichnet man als Prioritätswarteschlange.

Eine Prioritätswarteschlange besitzt die drei Grundoperationen DELETEMIN, INSERT und DECREASEKEY.

Anregung: Formulieren Sie die Prioritätswarteschlange als abstrakten Datentyp (sofern Kapitel 5 bekannt ist).

Prioritätswarteschlangen realisiert man oft als Heap (10.4.3).

Eine Menge von n Schlüsseln werde als Heap gespeichert.

Dann realisiert man die Operation X := DELETEMIN, indem man das oberste Element des Heaps der Variablen X zuweist, das letzte Element des Heaps an die oberste Stelle setzt und es dann absinken lässt. Zeitkomplexität: $\le 2 \cdot \log(n)$ Vergleiche.

Die Operation INSERT(Y) wird realisiert, indem der Wert von Y als letztes Element an den Heap angehängt wird und dieses Element dann im Heap aufsteigt, d.h., man vergleicht diesen neuen Schlüssel D(Y) mit seinem Elternknoten; falls der Elternknoten ein größeres Element enthält, werden die Inhalte vertauscht und man fährt genauso an dem neuen Knoten fort. Zeitkomplexität: $\le \log(n)+1$ Vergleiche.

Bei DECREASEKEY(Y) lässt man das Element Y, das ja schon im Heap steht, aufsteigen, sofern sich D(Y) durch einen Weg über den Knoten X verringert hat. Zeitkomplexität: $\le \log(n)+1$ Vergleiche.

Mit dieser Heap-Realisierung lässt sich nun die Komplexität des Dijkstra-Algorithmus abschätzen. Weil in jedem äußeren Schleifendurchlauf genau ein Element aus V-{u} bearbeitet wird, gibt es maximal n-1 Durchgänge der while-Schleife.

```
B := {u};  R := {v | (u,v) ∈ E};        -- n Schritte
while R ≠ ∅ do                          -- maximal n-1 Schleifen
    X := DELETEMIN;                     -- ≤2·log(n) Vergleiche
    for all  Y ∈ S(X) and Y ∉ B  do     -- maximal n-1 Schleifen
        if  Y ∉ R  then  INSERT(Y)      -- ≤log(n) Vergleiche
        else  DECREASEKEY(Y)  fi        -- ≤log(n) Vergleiche
    od
od;
```

Es gibt also höchstens

$(n-1) \cdot (2 \cdot \log(n) + (n-1) \cdot \log(n)) = O(n^2 \cdot \log(n))$ Vergleiche.

Eine genauere Betrachtung ergibt eine günstigere Schranke:

Die Abfrage "Y ∈ S(X) and Y ∉ B" wird für jede Kante genau ein Mal durchgeführt, insgesamt also m Mal (m = |E|). Das Einfügen und das Aufsteigen eines Elements erfordert jedes Mal höchstes log(n) Vertauschungen. Somit erfordert die innere Schleife *insgesamt* höchstens $O(m \cdot \log(n))$ Schritte.

Hinzu kommen dann nur noch die maximal $O((n-1) \cdot 2 \cdot \log(n))$ Operationen, die durch DELETEMIN verursacht werden.

Satz 11.2.6: Der gesamte Zeitaufwand beträgt daher

$$O(\underbrace{(n-1) \cdot 2 \cdot \log(n)}_{\text{verursacht durch DELETEMIN}} + \underbrace{m \cdot \log(n)}_{\text{verursacht durch INSERT und DECREASEKEY}}) = \mathbf{O((n+m) \cdot \log(n))}.$$

Laufzeit des Dijkstra-Algorithmus, wenn man ihn mit Heaps implementiert

Aufgabe: Man könnte nun folgendermaßen argumentieren:

Die Abfrage "Y ∈ S(X) and Y ∉ B" wird für jede Kante genau ein Mal durchgeführt, insgesamt also m Mal (m = |E|). Das Einfügen und das Aufsteigen eines Elements erfordern insgesamt höchstes $n \cdot \log(n)$ Vertauschungen (spätestens dann ist das jeweilige Element an der Spitze des Heaps angekommen); alle n-1 Elemente zusammen benötigen daher maximal $(n-1) \cdot \log(n)$ Vertauschungen. Somit erfordert die innere Schleife insgesamt höchstens $O(m + n \cdot \log(n))$ Schritte.
Hinzu kommen dann nur noch die maximal $O((n-1) \cdot 2 \cdot \log(n))$ Operationen, die durch DELETEMIN verursacht werden.
Somit würden wir insgesamt folgenden Zeitaufwand erhalten:
$O((n-1) \cdot 2 \cdot \log(n) + m + n \cdot \log(n)) = O(m + n \cdot \log(n))$.

Was ist an dieser Argumentation falsch??

11.2.7 Realisierung des Dijkstra-Algorithmus in Ada

Wir geben eine "strukturierte Lösung" für die Realisierung an, also eine Lösung, die den Algorithmus aus 11.2.4 beibehält und die Prioritätswarteschlange als Modul (package) formuliert. Wer sich hier einige Stunden lang durchbeißt (und eventuelle Fehler aufdeckt), hat den Algorithmus und die Implementierung wirklich verstanden.

Eine Prioritätswarteschlange wird über einer Schlüsselmenge definiert, die eine Ordnung besitzt. In Ada bietet sich hierfür ein generisches Paket an:
Die Typen für Knoten und Schlüssel sind generisch, zusammen mit den Operationen "Addition" und "Kleiner-Relation" auf dem Schlüsseltyp.
Das Paket muss einen privaten Typ für die Prioritätswarteschlange (pqk), die drei Grundoperationen und einige Hilfs-Funktionen (Initialisieren, Test auf Leerheit) besitzen. Die Spezifikation kann man daher unmittelbar hinschreiben:


```
generic   type Node is private; type Key is private;
          Z: constant Natural;
          with function "+"(I, J: Key) return Key;
          with function "<"(I, J: Key) return Boolean;
package Prio_Queue is
  type Item is record
     Kn: Node; Vor: Node; Sch: Key;
  end record;
  type pqk is private;
  procedure Empty;
  function Isempty return Boolean;
  function DeleteMin return Item;
  procedure Insert (K: Item);
  procedure DecreaseKey (K: Item);
  private
     type MaxHeap is 0..Z;
     type pqk is array (MaxHeap) of Item;    -- pqk ≡ priority queue w.r.t. key
end package;
```

Aktueller Knoten *Kn*, Vorgängerknoten *Vor* auf einem kürzesten Weg, *Sch* = Distanz zum Startknoten (zum Speichern des D-Werts) (*Sch* negativ => Fehlerfall)

Die Implementierung erfolgt in Ada als package body.

Die vorläufigen Werte für die kürzeste Entfernung werden in der Item-Komponente *Sch* für jeden Knoten gespeichert. Die Programmierung muss dafür sorgen, dass dieser Wert beim Löschen in R und Aufnehmen in B erhalten bleibt. Dies geschieht außerhalb des Pakets Prio_Queue. Um die kürzesten Wege später rekonstruieren zu können, benötigen wir die Komponente *Vor*.
Wir stellen nun die zwölfseitige Implementierung des Pakets vor. Das Feld KK vom Typ pqk bildet den Heap der Items. In der Reihenfolge des Entfernens von Knoten aus KK wird das Feld D der Ergebnisse aufgebaut. Um auf die Knoten im Heap direkt zugreifen zu können, führen wir das Feld HeapIndex ein mit
HeapIndex(K) = "der Index i mit KK(i).Kn = K".

```
package body Prio_Queue is
type MaxHeap is 0..Z;
type pqk is array (MaxHeap) of Item;
KK: pqk;                          -- KK wird hier als Heap aufgefasst
Anzahl: MaxHeap;
HeapIndex: array (Node) of MaxHeap;
procedure Empty is
  begin Anzahl := 0; end;
function Isempty return Boolean is
  begin return (Anzahl=0); end;

-- Für das Folgende beachte man:
-- Es besteht eine Beziehung zwischen einem Knoten und seinem Index im
-- Heap. Hierfür verwenden wir HeapIndex: array (Node) of MaxHeap.
-- Im Feld HeapIndex wird für jeden Knoten K notiert, ob er sich im Heap
-- befindet; genau dann gilt: (HeapIndex(K) > 0), und in diesem Falle ist
-- HeapIndex(K) sein Index im Heap KK.
-- Delta: array (Node,Node) of Key ist die Kantenmarkierung δ im Graphen,
-- die eigentlich als Komponente W in den Kanten gespeichert ist.
```



```
-- Wir übernehmen die Prozedur sink für den Heap KK aus 10.4.4,
-- müssen allerdings die Verknüpfung zwischen einem Knoten und seinem
-- Index im Heap stets aktualisieren, d.h., HeapIndex muss beim
-- Aufsteigen oder Absteigen eines Knotens im Heap entsprechend
-- umgesetzt werden.

procedure sink (links, rechts: MaxHeap) is
i, j: Natural;  weiter: Boolean:=true;  v: Item;
begin  v := KK(links);  i := links;  j := i+i;
  while  (j <= rechts) and weiter  loop
    if  j = rechts  then
        if v.Sch < KK(j).Sch then
            KK(i):=KK(j);
            HeapIndex(KK(j).Kn) := i;
            i:=j;
        end if;
        weiter:=false;
```



```
    elsif KK(j).Sch < KK(j+1).Sch then
        if v.Sch < KK(j+1).Sch then
            KK(i) := KK(j+1);
            HeapIndex(KK(j+1).Kn) := i;
            i := j+1;
        else weiter:=false; end if;
    else  if v.Sch < KK(j).Sch then
            KK(i) := KK(j);
            HeapIndex(KK(j).Kn) := i;
            i := j;
        else weiter:=false; end if;
    end if;
    j := i+i;
  end loop;
  KK(i):=v;
  HeapIndex(v.Kn) := i;
end sink;
```



```
function DeleteMin return Item is
Wurzel: Item;
begin
  if not Isempty then  Wurzel := KK(1);  Anzahl := Anzahl - 1;
  else Wurzel := KK(0);  end if;          -- dieser else-Fall tritt nie ein
  if not Isempty then  KK(1) := KK(Anzahl+1);
     sink(1, Anzahl); end if;
  HeapIndex(Wurzel.Kn) := 0;
  return Wurzel;
end DeleteMin;

-- Man beachte, dass wie üblich die zu entfernenden Elemente nicht gelöscht
-- werden, sondern dass nur der Index "Anzahl", der auf das letzte Element
-- im Heap zeigt, um 1 verringert wird. Die Manipulationen der Menge B
-- ziehen wir heraus; sie werden im Dijkstra-Algorithmus direkt ausgeführt.
-- In KK(0) speichern wir ein "null record", welches immer dann zurück
-- gegeben wird, wenn die Prioritätswarteschlange R leer ist; dieser Fall tritt
-- aber wegen  while R ≠ ∅ do  (= while not Isempty do) nicht ein.
```



```
procedure Aufsteigen(Los: MaxHeap) is      -- eine Hilfsprozedur
i, j: Natural;  v: Item;
begin  i := Los; j := i/2;      -- j zeigt auf den Elternknoten von i im Heap
  if  j = 0  then  HeapIndex(KK(1).Kn) := 1;
  else while  j > 0  and then  KK(i).Sch < KK(j).Sch  do
          v := KK(i);  KK(i) := KK(j);  KK(j) := v;
          HeapIndex(KK(i).Kn) := j;  HeapIndex(KK(j).Kn) := i;
          i := j;  j := i/2;  end loop;
  end if;
end Aufsteigen;

procedure Insert (K: Item) is
begin                          -- K am Ende einfügen und aufsteigen lassen
  Anzahl := Anzahl+1; KK(Anzahl) := K;
  HeapIndex(K.Kn) := Anzahl;
  Aufsteigen(Anzahl);
end Insert;
```



```
procedure DecreaseKey (K: Item) is
IndexK: MaxHeap;
begin  IndexK := HeapIndex(K.Kn);
    if K.Sch < KK(IndexK).Sch
       then KK(IndexK) := K;
       Aufsteigen (IndexK); end if;
end DecreaseKey;

begin Empty;                          -- Initialisierung des Pakets
      for q in Node loop HeapIndex(q) := 0; end loop;
end package Prio_Queue;
```

Mit diesem Paket lässt sich nun der Dijkstra-Algorithmus als Block (mit einem Unterblock und Prozedur) ausformulieren.


```
declare                          -- erforderliche Vorab-Deklarationen
  generic ... "generischer Teil wie oben angegeben";
  package Prio_Queue ... "Spezifikation wie oben angegeben";
  package body Prio_Queue ..."wie oben angegeben";
  type Knoten;  type NextKnoten is access Knoten;
  type Kante;  type NextKante is access Kante;
  type Knoten is record
       Id: Knotenname;
       Besucht: Boolean; zahl1, zahl2: Integer;
       Inhalt: <weitere Komponenten>;
       NKn: NextKnoten;    -- nächster Knoten (in Knotenliste)
       EIK: NextKante;     -- erste inzidente Kante
  end record;
  type Kante is record
       W: <Typ des Gewichts der Kanten>;
       EKn: NextKnoten;    -- Endknoten dieser Kante
       NKa: NextKante;     -- nächste Kante (bzgl. Knoten)
  end record;
```



```
  N, M: Natural;  Anfang: NextKnoten;
  Delta: array (Knoten, Knoten) of Float := (others => "∞");
  procedure Graph_Aufbau (Anker: in out NextKnoten) is
  "Prozedur, die einen Graphen einliest und/oder als Adjazenzliste
  aufbaut, auf die danach Anker zeigt; zugleich werden N und M als
  Zahl der Knoten und der Kanten bestimmt."
  end Graph_Aufbau;
  function G_korrekt(Anker: NextKnoten) return Boolean is
  ok: Boolean := true;  p: Nextknoten;  e: Nextkante;
  begin p := Anker;
      while p /= null loop  e := p.EIK;
          while e /= null loop
          Delta(p.all, e.EKn.all) := e.W; e:=e.NKa;
          if e.W < 0.0 then ok := false; end if;
      end loop; end loop;
  return ok;
  end G_korrekt;
```

G-korrekt überträgt die Kantenmarkierung nach Delta und prüft, ob alle Werte nichtnegativ sind. Man sollte noch prüfen, dass keine Schlingen und Mehrfachkanten vorliegen (selbst einfügen).


```
begin                -- Beginn des Anweisungsteils für den Graphen
  Graph_Aufbau (Anfang);      -- jetzt sind N und M bekannt
  if not G_korrekt then ... <"Fehlerfall behandeln">;
  else                        -- Deltatabelle ist nun gesetzt
  declare                     -- Beginn des Blockes für Dijkstra-Alg.
  package G_Prio_Queue is     -- Prio-Warteschlange für Knoten
      new Prio_Queue (Knoten, Float, N, "+", "<");
  with Ada.Integer_Text_IO; with Ada.Text_IO;
  with G_Prio_Queue; use G_Prio_Queue;
  Startknoten: Knoten;
  Zustand: array (Knoten) of (B, R, U) := (others => U);
                -- gibt an, ob ein Knoten in B, R oder U liegt
  D: pqk;                     -- zum Speichern der Ergebnisse
                              -- wegen "use" ist pqk hier sichtbar
  Zähler: Natural;            -- zählt die Knoten in B
```

```
procedure Dijkstra_with_Heap(Start: Knoten) is
X: Item; Y: Knoten;  Edge: NextKante;  H: Float;
begin                          -- entsprechend Verfahren in 11.2.4
   Edge := Start.EIK;          -- Initialisiere die Menge R (= KK).
   while Edge /= null loop
                  -- Alle Nachfolger des Startknotens Start kommen nach R
      Y := Edge.EKn.all;
      Zustand(Y) := R;
      Insert((X.Kn, Y, Delta(Start, Y)));
      Edge := Edge.NKa;
   end loop;
   Zustand(Start) := B;
   D(1) := (Start, Start, 0.0);
   Zähler := 1;
   while not Isempty loop                -- solange R nicht leer ist
      X := DeleteMin;  Zustand(X.Kn) := B;
      Zähler := Zähler+1;  D(Zähler) := X;
      Edge := X.Kn.EIK;
      while Edge /= null loop
         Y := Edge.EKn.all;
         if  Zustand(Y) /= B  then
            H := X.Sch + Delta(X.Kn, Y.all);
            if  Zustand(Y) = U  then
                  Insert((Y, X, H));  Zustand(Y) := R;
            else  DecreaseKey((Y, X, H));
         end if;
         Edge := Edge.NKa;
      end loop;
   end loop;
end Dijkstra_with_Heap;
```

```
   begin
      Startknoten := <"wähle einen Knoten aus">;
      Dijkstra_with_Heap (Startknoten);
      for i in 1..Zähler loop
         Put(D(i).Kn.Id); Put(D(i).Vor.Id); Put(D(i).Sch);
      end loop;
      ...
   end;                    -- Ende des inneren Blocks für den Dijkstra-Alg.
end if;
...
end;                    -- Ende des äußeren Blocks der Vorab-Deklarationen
```


11.2.8 Hinweise

1. Obiger Algorithmus wurde nicht getestet und kann daher (und wird vermutlich auch) noch einige Flüchtigkeitsfehler enthalten.

2. Da im Verfahren stets der Knoten X mit der geringsten Entfernung ausgewählt wird, gehört der Dijkstra-Algorithmus zu den Greedy-Verfahren, siehe 6.7.

3. Eine andere Implementierung (mit den sog. Fibonacci-Heaps) bringt die bessere Zeitkomplexität $O(m+n\cdot\log(n))$; allerdings werden die Konstanten hierbei größer, sodass sich diese Datenstruktur für die Praxis oft nicht lohnt.

4. Dringende Empfehlung für alle: "Opfern" Sie in der vorlesungsfreien Zeit drei Tage nur zu dem Zweck, den Dijkstra-Algorithmus in Ada zum Laufen gebracht zu haben. Anders werden Sie die Schwierigkeiten, die bereits mit dem "Programmieren im Kleinen" verbunden sind, kaum verinnerlichen. Auch erhalten Sie ein Gespür dafür, wie mühsam es ist, für die Praxis taugliche Bibliotheks-programme (oder objektorientierte Klassen) zu erstellen.

5. Für die Berechnung der kürzesten Wege zwischen allen Knotenpaaren verwendet man den sehr leicht zu implementierenden Floyd-Algorithmus. Dieser ist dem "Warshall-Algorithmus" zur Berechnung der transitiven Hülle (6.5.4) sehr ähnlich, siehe Übungen oder Literatur.

11.3 Minimale Spannbäume

Gegeben sei ein ungerichteter Graph $G=(V,E,\delta)$ mit $\delta: E\to\mathbb{R}$ (= Menge der reellen Zahlen).
Gesucht wird ein minimaler Spannbaum, also ein Baum $B=(V,E_B,\delta)$ mit gleicher Knotenmenge, der Teilgraph von G ist und dessen Gewicht $\delta(B)$ minimal ist bzgl. aller Spannbäume von G (siehe Def. 8.8.6).

Zur Lösung dieses Problems verwendet man den Prim-Algorithmus mit der gleichen Zeitkomplexität des Dijkstra-Algorithmus $O((n+m)\cdot\log(n))$ oder den Kruskal-Algorithmus mit der Zeitkomplexität $O(m\cdot\log(m))$.

J.B.Kruskal und R.C.Prim veröffentlichten ihre Algorithmen 1956 bzw. 1957.

11.3.1 Der Algorithmus von Prim

Der Prim-Algorithmus arbeitet genau wie der Dijkstra-Algorithmus, jedoch mit einem modifizierten Schlüssel *Sch*, der die Reihenfolge in der Prioritätswarteschlange festlegt. Während *bei Dijkstra* für alle $y\in R$ zu jedem Zeitpunkt gilt

$$Sch(y) = Min\{Sch(v) + \delta((v,y)) \mid \text{für alle Knoten } v\in B\},$$

verwendet man *beim Prim*-Algorithmus den kleinsten Abstand von y zu irgendeinem Knoten aus B, d. h., für alle $y\in R$ lautet jetzt der Schlüssel *Sch* zu jedem Zeitpunkt

$$Sch(y) = Min\{\delta((v,y)) \mid \text{für alle Knoten } v\in B\}.$$

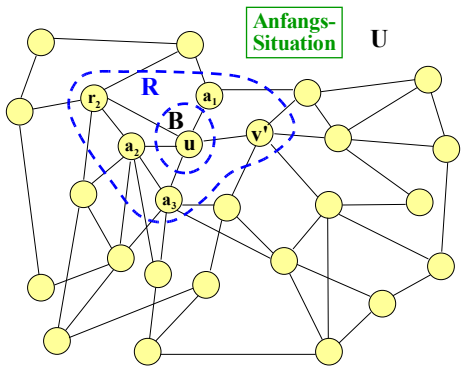
Mit dieser geringen Änderung geht das Programm 11.2.7 in den Prim-Algorithmus über. Auch hier entsteht ein Baum, da in die Menge B jeder Knoten mit genau einer Kante (zu seinem jeweiligen Vorgänger in B) aufgenommen wird.

Genauer: Nur in der Prozedur Dijkstra_with_Heap ist die Zeile
"H := X.Sch + Delta(X.Kn, Y.all);"
zu ersetzen durch
H := Delta(X.Kn, Y.all);

Die Initialisierung der Schlüsselwerte im Aufruf "Insert" zu Beginn der Prozedur Dijkstra_with_Heap erfolgt mit den Delta-Werten und ist daher bereits korrekt.

Da der Graph beim Prim-Algorithmus ungerichtet, beim Dijkstra-Algorithmus dagegen gerichtet ist, muss jede ungerichtete Kante durch zwei gerichtete Kanten (mit gleichem δ-Wert) dargestellt sein, wenn wir das Programm ohne weitere Änderungen übernehmen wollen!

Wir demonstrieren den Prim-Algorithmus an einem Beispiel.

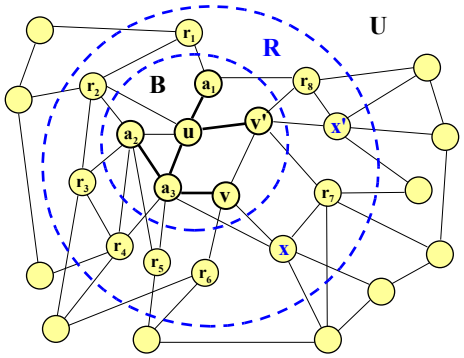


Ein Beispiel-Graph G = (V, E, δ) mit Startknoten u.
Anfangs werden B = {u} und R = {r_2, a_1, a_2, a_3, v'} = Menge der Nachbarknoten von u gesetzt. U = Menge der restlichen Knoten.

Jede ungerichtete Kante liegt im Programm als zwei gerichtete Kanten vor.

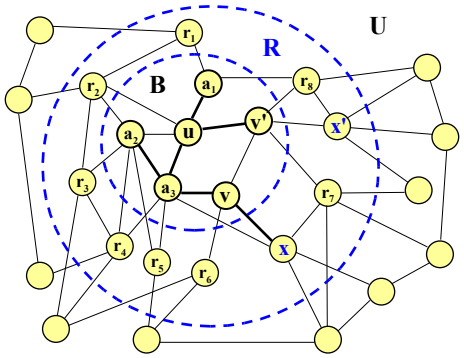
Die Werte der Kantenmarkierung δ sind hier nicht eingetragen.

Beachte: Die Menge der Kanten, die zwischen B und R verlaufen, trennt die Mengen B und V-B, d.h., jeder Weg von einem Knoten aus B zu einem Knoten aus V-B führt über mindestens eine solche Kante.



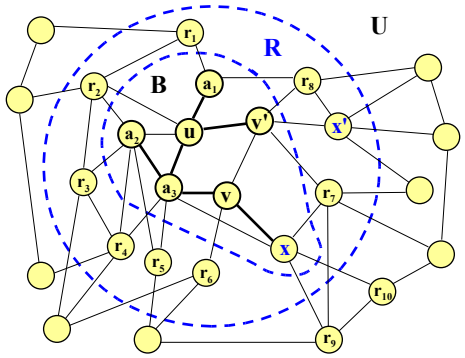
Der bereits erstellte Teil eines minimal spannen-den Baums ist mit fetten Kanten und fett umran-deten Knoten (B) dargestellt. Die kleinste Kante sei nun {v,x}.

Betrachte eine aktuelle Situation beim Prim-Algorithmus:
B = {u, a1, a2, a3, v, v'}, R = {x, x', r1, r2, r3, r4, r5, r6, r7, r8},
U = Menge der restlichen Knoten (diese haben hier keine Namen).



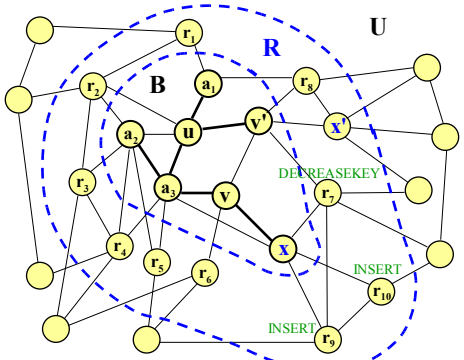
Der bereits erstellte Teil eines minimal spannen-den Baums ist mit fetten Kanten und fett umran-deten Knoten (B) dargestellt. Die kleinste Kante sei nun {v,x}. Dann wird x zusammen mit dem Verweis auf v und δ({v,x}) in B aufgenommen und die mit x ad-jazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüs-selwert *Sch* wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation beim Prim-Algorithmus:
B = {u, a1, a2, a3, v, v'}, R = {x, x', r1, r2, r3, r4, r5, r6, r7, r8},
U = Menge der restlichen Knoten (diese haben hier keine Namen).



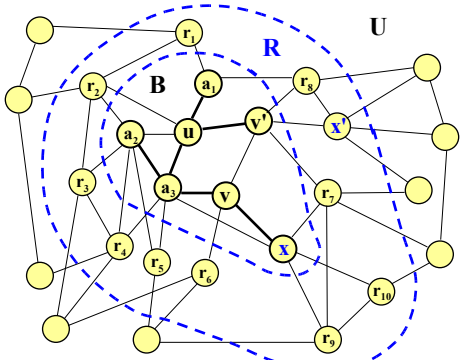
Der bereits erstellte Teil eines minimal spannen-den Baums ist mit fetten Kanten und fett umran-deten Knoten (B) dargestellt. Die kleinste Kante sei nun {v,x}. Dann wird x zusammen mit dem Verweis auf v und δ({v,x}) in B aufgenommen und die mit x ad-jazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüs-selwert *Sch* wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation beim Prim-Algorithmus:
B = {u, a1, a2, a3, v, v'}, R = {x, x', r1, r2, r3, r4, r5, r6, r7, r8},
U = Menge der restlichen Knoten (diese haben hier keine Namen).



Der bereits erstellte Teil eines minimal spannen-den Baums ist mit fetten Kanten und fett umran-deten Knoten (B) dargestellt. Die kleinste Kante sei nun {v,x}. Dann wird x zusammen mit dem Verweis auf v und δ({v,x}) in B aufgenommen und die mit x ad-jazenten Knoten, die nicht in B liegen, werden in R aufgenommen oder ihr Schlüs-selwert *Sch* wird ggfls. erniedrigt (falls sie schon in R liegen).

Betrachte eine aktuelle Situation:
B = {u, a1, a2, a3, v, v'}, R = {x, x', r1, r2, r3, r4, r5, r6, r7, r8},
U = Menge der restlichen Knoten (diese haben hier keine Namen).



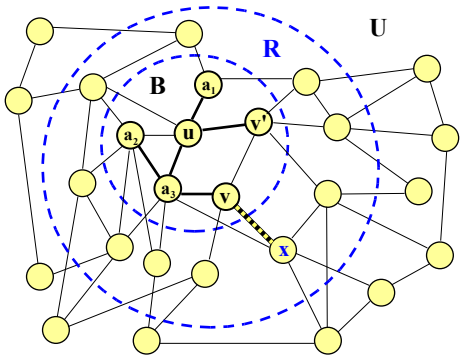
Der bereits erstellte Teil eines minimal spannen-den Baums ist mit fetten Kanten und fett umran-deten Knoten (B) dargestellt. Die kleinste Kante sei nun ... .

Wir erhalten die nächste Situation:
B = {u, a1, a2, a3, v, v', x}, R = {x', r1, r2, r3, r4, r5, r6, r7, r8, r9, r10},
U = Menge der restlichen Knoten (diese haben hier keine Namen).

11.3.2 Wie kann man beweisen, dass dieser Algorithmus tatsächlich einen minimalen Spannbaum konstruiert?

Mit Hilfe der Eigenschaft, dass die Kanten zwischen B und R die Mengen B und V-B trennen, geht dies sehr einfach (siehe Folie mit der Anfangssituation oben).

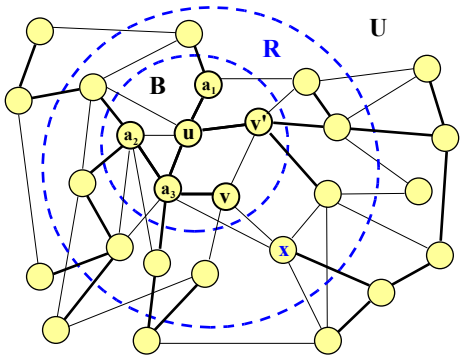
Korrektheitsbeweis: Zunächst sei der Graph zusammenhängend, denn sonst besitzt er keinen Spannbaum.
Annahme, wir hätten für den minimalen Spannbaum eine kleinste Kante nicht verwenden dürfen. D.h., in keinem minimalen Spannbaum kommt die Kante {v,x} vor, obwohl sie in einer Situation des Prim-Algorithmus kleiner gleich allen anderen Kanten, die von B nach R führten, war und folglich vom Prim-Algorithmus ausgewählt wurde. Wir betrachten eine Situation aus dem obigen Beispiel zur Illustration:



Der bereits erstellte Teil eines minimal spannen-den Baums ist mit fetten Kanten und fett umran-deten Knoten (B) dargestellt. Die kleinste Kante von B nach R sei nun {v,x}.

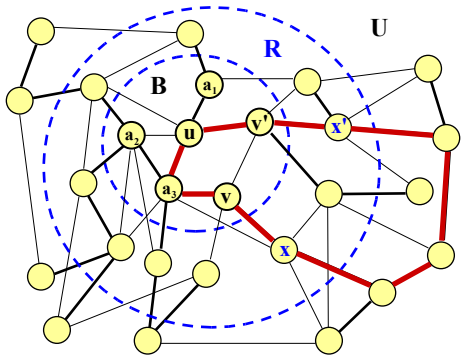
Wir nehmen nun an: Diese Kante {v,x} ist in keinem minimalen Spannbaum enthalten.

In dieser aktuellen Situation des Prim-Algorithmus sei der Schlüssel *Sch* von x minimal und die Kante {v,x} habe den kleinsten δ-Wert aller Kanten zwischen B und R.



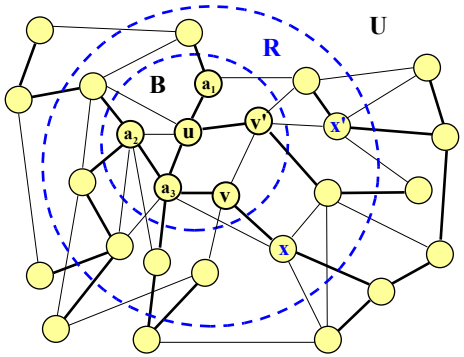
Ein minimaler Spannbaum wird mit fetten Kanten dargestellt.

Sei dies ein minimaler Spannbaum, in dem die Kante {v,x} nicht enthalten ist. Fügt man nun die Kante {v,x} zu diesem Spannbaum hinzu, so muss ein Zyklus entstehen, auf dem v und x liegen.



Wir haben {v,x} hinzugefügt. Der entstandene Zyklus ist mit dickeren Kanten hervorgehoben.

Da v in B und x in V-B lag, muss es genau eine Kante {v',x'} auf diesem Zyklus geben, die zum Zeitpunkt, als {v,x} ausgewählt wurde, ebenfalls vorhanden war, d.h., es gilt $\delta(\{v,x\}) \le \delta(\{v',x'\})$.



Wir lassen nun {v',x'} weg. Der neue Spannbaum ist mit fetten Kanten dargestellt.

Ersetzt man also {v', x'} im minimalen Spannbaum durch {v, x}, so erhält man wieder einen Spannbaum, dessen Gewicht um den Wert $\delta(\{v', x'\}) - \delta(\{v, x\}) \geq 0$ kleiner ist!

Dieser neue Spannbaum, in dem {v, x} vorkommt, besitzt also ein Gewicht, das kleiner oder gleich dem des zuerst betrachteten minimalen Spannbaums ist.

Folglich gibt es doch einen minimalen Spannbaum, der die Kante {v, x} enthält, im Widerspruch zur Annahme. Daher gibt es also stets einen minimalen Spannbaum, in dem die vom Prim-Algorithmus ausgewählten Kanten vorkommen. Der Prim-Algorithmus liefert also stets einen minimalen Spannbaum. ■

Natürlich kann es mehrere minimale Spannbäume zu einem Graphen geben, wenn es Kanten mit gleichem δ-Wert im Graphen gibt. Dann treten eventuell in der Prioritätswarteschlange mehrere Knoten mit gleichem minimalem Schlüssel *Sch* auf. Egal welchen solchen Knoten man dann wählt, man erhält am Ende stets einen minimalen Spannbaum.

11.3.3 Der Algorithmus von Kruskal

Der Kruskal-Algorithmus sortiert die Kanten bzgl. δ, beginnt dann mit der kleinsten Kante und nimmt jeweils die nächste Kante hinzu, sofern diese Kante keinen Zyklus mit den bereits ausgewählten Kanten bildet. Auf diese Weise entsteht ein minimaler Spannbaum.

Die Zeitkomplexität beträgt $O(m \cdot \log(m))$ mit $m = |E|$.

Dies gilt aber nur bei geeigneter Implementierung, insbesondere muss es eine Liste oder ein Feld der Kanten geben (hier empfiehlt sich eine Inzidenzlistendarstellung des Graphen, in der die Knoten und die Kanten in eigenen Listen hintereinander stehen und Verweise zu den Endknoten von der Kanten- in die Knotenliste existieren).

"Einziges" Problem hierbei:

Wie stellt man fest, ob eine Kante mit den bereits ausgewählten Kanten einen Zyklus bildet?

Lösung: Man bildet ein System von Mengen aus Knoten und fasst jeweils in einer Menge genau die Knoten zusammen, die untereinander durch bereits ausgewählte Kanten verbunden sind.

Für jede Kante prüft man dann, ob ihre Endknoten in der gleichen Menge liegen. Falls ja, so verwirft man die Kante, denn dann bewirkt diese Kante einen Zyklus; falls nein, so nimmt man die Kante zu den Baumkanten hinzu und vereinigt die beiden Mengen, in denen die Endknoten lagen, zu einer neuen Menge.

Kruskal-Algorithmus für einen minimalen Spannbaum:

Gegeben sei ein ungerichteter Graph $G = (V, E, \delta)$.

1. Sortiere alle Kanten nach ihrem δ-Wert.
2. Bilde für jeden Knoten $y \in V$ die Menge {y};
   BK := ∅; -- dies wird die Menge der Baumkanten
3. while (die Kantenliste ist nicht leer) and (|BK| < n-1) do
   Entferne die kleinste Kante e von der Kantenliste;
   if die Endknoten x und y der Kante e liegen in verschiedenen Mengen M und N then
   bilde M := M∪N; vergiss N; BK := BK∪{e} fi
   od;

Die Kanten von BK bilden einen minimalen Spannbaum.

11.3.4 *Korrektheit des Kruskal-Algorithmus:*

Der Beweis, dass dieses Vorgehen einen minimalen Spannbaum liefert, wird ähnlich wie der Beweis zum Prim-Algorithmus geführt. Annahme, eine Kante e, die vom Kruskal-Algorithmus ausgewählt wurde, würde in keinem minimalen Spannbaum sein, dann fügt man sie zu einem minimalen Spannbaum hinzu, wodurch ein Zyklus entsteht, auf dem mindestens eine Kante mit nicht-kleinerem δ-Wert liegen muss; gegen diese tauscht man die Kante e aus und erhält einen minimalen Spannbaum, in dem die Kante e doch vorkommt.

(Hinweis: Gäbe es auf dem Zyklus außer der Kante e nur Kanten mit kleinerem δ-Wert, so könnte die Kante e nicht vom Kruskal-Algorithmus ausgewählt worden sein, da alle anderen Kanten vorher hätten ausgewählt worden sein müssen und e dann einen Zyklus bewirkt hätte.) ■

Der Algorithmus stellt nebenbei auch fest, ob der gegebene Graph zusammenhängend war: Dies trifft genau dann zu, wenn ein Spannbaum konstruiert wird, wenn also am Ende genau n-1 Kanten ausgewählt wurden.

Das Vereinigen der beiden Mengen wurde hier so realisiert, dass eine der Mengen (hier: M) neu auf M∪N gesetzt wird, während die andere Menge entfernt wird. Dies kann man natürlich in dieser Weise nicht implementieren, da das Löschen viel zu viel Aufwand erfordern würde. Vielmehr fasst man die beiden Mengen zusammen und vergisst deren alte Bedeutung.

11.3.5 Der Kruskal-Algorithmus braucht zwei Operationen:

FIND(x) = Stelle die Menge fest, in der das Element x liegt.

UNION(M, N) = Vereinige die beiden Mengen M und N, nenne das Ergebnis wieder M und vergiss N.

Hiermit lautet der Kruskal-Algorithmus:

Sortiere alle Kanten nach ihrem δ-Wert in eine Liste L;

for all $y \in V$ do $M_y := \{y\}$ od; BK := leere Liste;

while not isempty(L) and (|BK| < n-1) do
  e := DELETEMIN(L); x und y seien die Endknoten von e;
  M := FIND(x); N := FIND(y);
  if M ≠ N then UNION(M, N); füge e zu BK hinzu fi
od;

Das sog. UNION-FIND-Problem fragt nach einer effizienten Implementierung der Operationen UNION und FIND unter der Nebenbedingung, dass je O(n) dieser Operationen auf einer Gesamtmenge von n Elementen durchgeführt werden müssen.

Als gute Implementierung gilt die Darstellung jeder Menge als Baum, dessen Kanten von jedem Knoten zu seinem Vorgänger gerichtet sind und in dessen Wurzel zusätzlich die Anzahl der Elemente dieses Baums vermerkt ist (s.u.). Dann lässt sich

FIND(x) als Bestimmung der Wurzel des Baums, in dem sich x befindet, und

UNION(M,N) als Anhängen des kleineren Baums an die Wurzel des größeren Baums realisieren.

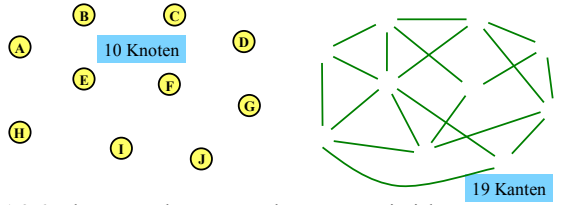
FIND(x) ≠ FIND(y) läuft dann auf die einfache Abfrage, ob beide Wurzeln der gleiche Knoten sind, hinaus.

Mit dieser Darstellung benötigt jedes FIND (x) höchstens log(n) Vergleiche (warum? Beachten Sie, dass stets der kleinere an den größeren Baum gehängt wird), und UNION kann in konstant vielen Schritten ausgeführt werden. Der eigentliche Kruskal-Algorithmus benötigt also höchstens $O(m \cdot \log(n))$ viele Schritte, weil die while-Schleife bis zu m Mal durchlaufen wird und jedes FIND höchstens log(n) Schritte erfordert.

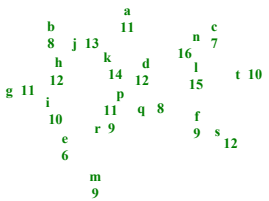
Nach Kapitel 10 kostet das Sortieren der m Kanten $O(m \cdot \log(m))$ Schritte. In der Praxis ist n in der Regel kleiner als m, daher wird die Komplexität des Kruskal-Algorithmus durch das Sortieren der m Kanten dominiert. Insgesamt ergibt sich in dem normalen Fall $n \leq m$ somit eine Zeitkomplexität von

$$O(m \cdot \log(m) + m \cdot \log(n)) = O(m \cdot \log(m)).$$

Hinweis:
"Verflacht" man die Bäume bei jeder FIND-Operation (sog. "Pfadkompression"), dann braucht man für alle UNION- und FIND-Operationen nur $O(m \cdot \log^*(n))$ Schritte. (Siehe Vorlesungen zur Algorithmik im weiteren Studium oder siehe Literatur; log* wurde in 6.1.4 behandelt.)
Die Zeitkomplexität wird hierdurch allerdings nicht verringert, da das Sortieren der Kanten unverändert $O(m \cdot \log(m))$ Schritte erfordert.

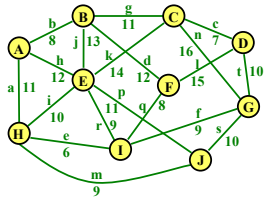


11.3.6 Erläuterung der Datenstrukturen am Beispiel.



Benennung der 19 Kanten und Gewichte $\delta : E \to \mathbb{R}$

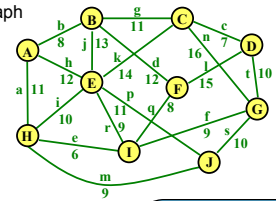
Der Graph G = (V, E, δ: E → ℝ):



Erste Aufgabe: Bilde die sortierte Liste der Kanten. Ergebnis:

(e,6), (c,7), (b,8), (q,8), (f,9), (m,9), (r,9), (i,10), (s,10), (t,10), (a,11), (g,11), (p,11), (d,12), (h,12), (j,13), (k,14), (l,15), (n,16).

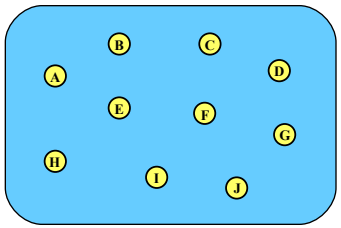
Graph



Sortierte Kantenliste

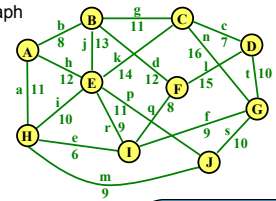
(e,6),
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

"Arbeitsfläche":
Ein Wald isolierter Knoten



Bilde nun die einelementigen Mengen M_y für jeden Knoten y:

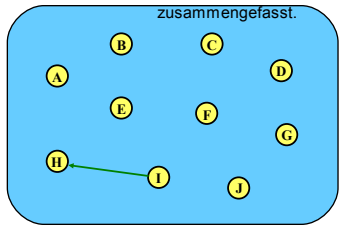
Graph



Sortierte Kantenliste

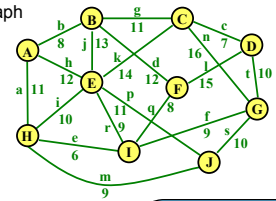
(e,6),
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte die erste Kante der sortierten Liste, also **(e,6)**:
(Die Ausrichtung der Kanten ist bei den Mengen zunächst willkürlich, später wird stets der kleine-re an den größeren Baum gehängt.)



Im Wald werden zwei Knoten zu einem Baum zusammengefasst.

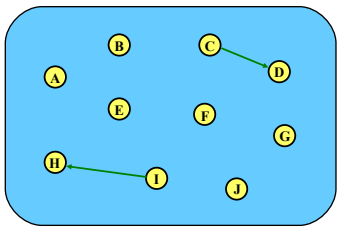
Graph



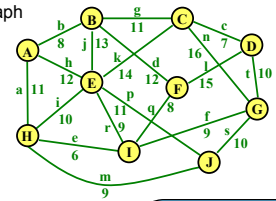
Sortierte Kantenliste

(e,6), +
(c,7),
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die zweite Kante der sortierten Liste, also **(c,7)**:
(Die Ausrichtung der Kanten ist bei den Mengen zunächst willkürlich, später wird stets der kleine-re an den größeren Baum gehängt.)



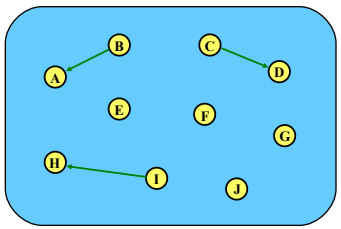
Graph



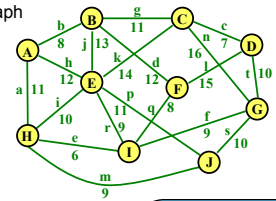
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8),
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die dritte Kante der sortierten Liste, also **(b,8)**:



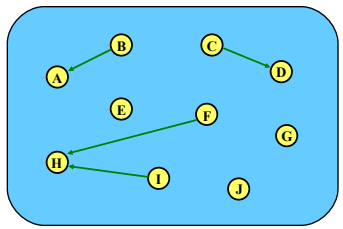
Graph



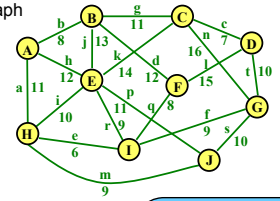
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8),
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Betrachte nun die vierte Kante **(q,8)**. Hänge den kleineren Baum "F" an den größeren, der zu I gehört.

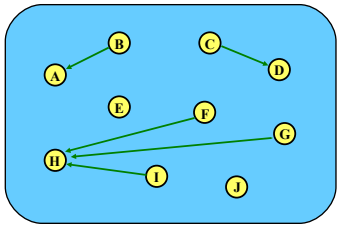


Graph



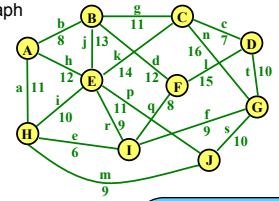
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9),
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



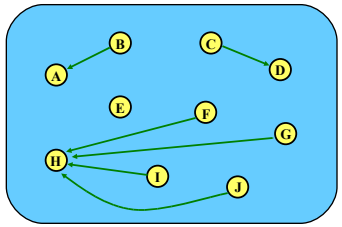
Betrachte nun die fünfte Kante **(f,9)**. Hänge den kleineren Baum "G" an den größeren, der zu I gehört.

Graph



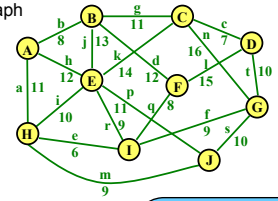
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9),
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



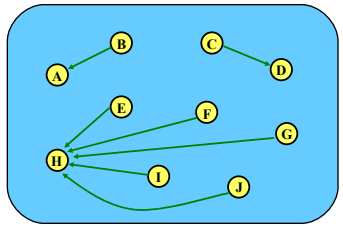
Betrachte nun die sechste Kante **(m,9)**. Hänge den kleineren Baum "J" an den größeren, der zu H gehört.

Graph



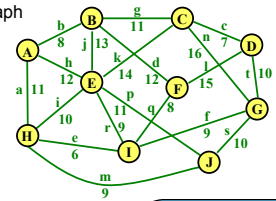
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9),
(i,10),
(s,10),
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



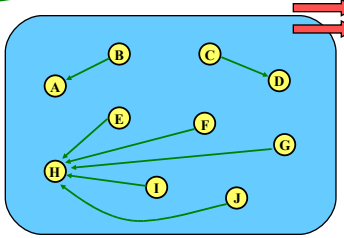
Betrachte nun die siebente Kante **(r,9)**. Hänge den kleineren Baum "E" an den größeren, der zu I gehört.

Graph



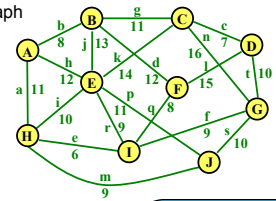
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10),
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



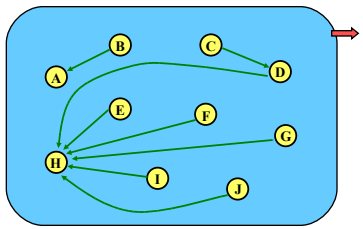
Betrachte nun die achte und neunte Kante. Jedes Mal trifft man auf "H", d. h., diese Kanten führen zu Zyklen.

Graph



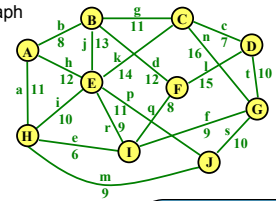
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



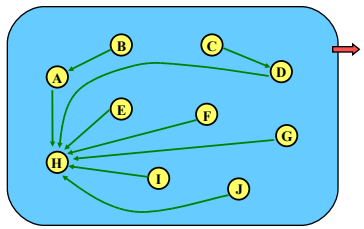
Betrachte nun die zehnte Kante **(t,10)**. Nun wird der Baum "D" an den größeren Baum "H" gehängt.

Graph



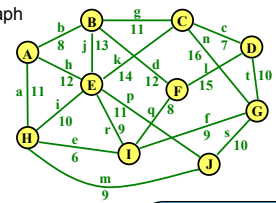
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11),
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



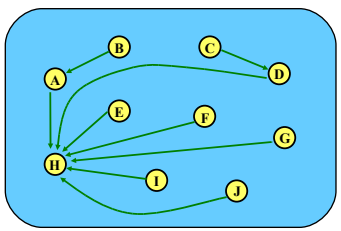
Betrachte jetzt die elfte Kante **(a,11)**. Nun wird der Baum "A" an den Baum "H" gehängt - und *wir sind fertig*, da n-1 Kanten gefunden wurden.

Graph



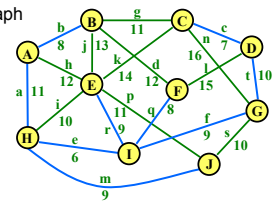
Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11), +
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).



Ergebnis: Die mit "+" markierten 9 Kanten **e, c, b, q, f, m, r, t** und **a** bilden einen minimalen Spannbaum.

Graph



Sortierte Kantenliste

(e,6), +
(c,7), +
(b,8), +
(q,8), +
(f,9), +
(m,9), +
(r,9), +
(i,10), -
(s,10), -
(t,10), +
(a,11), +
(g,11),
(p,11),
(d,12),
(h,12),
(j,13),
(k,14),
(l,15),
(n,16).

Ergebnis: Die mit "+" markierten 9 Kanten **e, c, b, q, f, m, r, t** und **a** bilden einen minimalen Spannbaum.

Zugleich wird bestätigt, dass der Graph zusammenhängend ist.

Ende des Beispiels.

Zur Datenstruktur: Verwandele einen Wald in einem Baum.

Zunächst bildet jeder Knoten des Graphen einen eigenen einelementigen Baum ohne Kanten (= isolierter Knoten). Man nehme die nächste Kante e der sortierten Kantenliste; ihre Endpunkte seien x und y. Von x ausgehend durchläuft man dessen Baum und endet in einem Wurzel-Knoten Z1; von y ausgehend landet man in einem Wurzel-Knoten Z2. Sind Z1 und Z2 verschieden, so hängt man den kleineren der beiden Bäume mit den Wurzeln Z1 und Z2 an den größeren (man verzeigert hier zur Wurzel hin!) und e wird zu den Kanten, die später einen minimalen Spannbaum bilden, hinzugenommen. Anderenfalls sind x und y bereits durch Kanten des Spannbaums verbunden und die aktuelle Kante e wird daher verworfen.

Programmieren Sie dies!

11.4 Maximales Matching

Vorbemerkung: Das Problem, zu speziellen Graphen ein maximales Matching zu finden, ist unter dem Namen Heiratsproblem bekannt. Seine Lösung ist unter den Problemen aufgeführt, die als "Algorithmus der Woche" im Informatikjahr 2006 ins Netz gestellt wurden, siehe:

http://www.informatikjahr.de/

Das Heiratsproblem lautet: Gegeben sind zwei disjunkte Mengen D und H ('Damen' und 'Herren') und eine Relation $E \subseteq \{\{x,y\} \mid x \in D \text{ und } y \in H\}$. Gesucht ist eine möglichst große Teilmenge F von E, in der jedes Element von D und von H höchstens einmal vorkommt. Die Relation E wird als "wechselseitig sympathisch" und die Menge F als Menge von Hochzeiten aufgefasst.

Definition 11.4.1:
$G=(V,E)$ sei ein ungerichteter Graph.
a) Eine Teilmenge der Kanten $F \subseteq E$ heißt Matching, wenn je zwei verschiedene Kanten von F disjunkt sind, d.h., für alle $\{x,y\}, \{x',y'\} \in F$ gilt:

aus $\{x,y\} \neq \{x',y'\}$ folgt $\{x,y\} \cap \{x',y'\} = \emptyset$.

b) Ein Matching $F \subseteq E$ heißt maximal, wenn es kein Matching $F' \subseteq E$ mit $|F| < |F'|$ gibt.

Man könnte meinen, das Problem, ein maximales Matching zu finden, sei einfach: Ausgehend von der leeren Menge nehme man schrittweise immer wieder eine Kante hinzu, die zwei noch nicht ausgewählte Knoten miteinander verbindet. Dies führt in der Regel jedoch nicht zu einem maximalen Matching (selbst ein Beispiel konstruieren!).

Definition 11.4.2: Ein ungerichteter Graph $G=(V,E)$ heißt bipartit, wenn seine Knotenmenge V so in zwei disjunkte Teilmengen D und H zerlegt werden kann, dass Kanten nur von einer zur anderen Teilmenge führen, d.h., $V = D \cup H$, $D \cap H = \emptyset$ und $E \subseteq \{\{x,y\} \mid x \in D \text{ und } y \in H\}$. Man schreibt dann auch $G=(D \cup H, E)$ oder $G=(D,H;E)$.

Das Problem, ein maximales Matching in einem bipartiten Graphen zu finden, bezeichnet man als *Heiratsproblem*. Es wurde 1935 von dem englischen Mathematiker Philip Hall charakterisiert.

Bemerkung: Diese Definitionen lassen sich leicht auf gerichtete Graphen übertragen, indem die Begriffe für die zugehörigen ungerichteten Versionen (siehe 3.8.5 c) gelten.

11.4.3

siehe zugehöriger Algorithmus der Woche unter

www.informatikjahr.de

Dieser Algorithmus wurde in der Vorlesung vorgestellt und erläutert. Zeitkomplexität: $O(n \cdot m)$.

Abschließende Hinweise zu weiteren Graphalgorithmen

Beispiele solcher Algorithmen:
Zusammenhangskomponenten
Färbbarkeit
größte vollständige Teilgraphen (Cliquenproblem)
minimale Rundreise (TSP = travelling salesman problem)
zweit-, dritt-, ..., k-t kürzeste Wege
Baumisomorphie (gerichtet / ungerichtet)
Graphisomorphie
Teilgraph-Isomorphie
Maximaler Fluss in einem Graphen-Netzwerk
(Stichwort: MaxFlow = MinCut).
Eine Informatik-Anwendung: Schicke maximal viele Datenpakete durch ein Netzwerk. Wie hoch ist der Fluss? Kann man eine feste Empfangszeit garantieren? Wie hängt dies von der Zahl der Empfänger ab? ....