9 Minimum Spanning Trees

Im Folgenden wollen wir uns genauer mit dem **„Minimum Spanning Tree“** -Problem auseinandersetzen.

9.1 MST-Problem

Gegeben ein ungerichteter Graph G = (V,E) und eine Gewichtsfunktion w

$$w : E \to \mathbb{R}$$

Man berechne einen minimalen aufspannenden Baum (MST) von G.

Definition 9.1 (Minimum Spanning Tree).
Sei G = (V,E) ein (ungerichteter) Graph, wobei jeder Kante e = (u,v) von E ein Gewicht w(u,v) zugeordnet ist.

1. Ein azyklischer Teilgraph T = (V, E') von G mit E' $\subseteq$ E, der alle Knoten V von G verbindet, bezeichnet man als *aufspannenden Baum*.

2. Das Gewicht von T ist die Summe der Kantengewichte:

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

3. Ein aufspannender Baum T von G mit minimalem Gewicht bezeichnet man als *minimalen aufspannenden Baum* (**M**inimum **S**panning **T**ree) von G.

Beispiel 9.1. Gegeben ein Graph G = (V,E). Die Knotenmenge V repräsentiere eine Menge von Spezies. Jede Kante e = (u,v) ist beschriftet mit der evolutionären Distanz w(u,v) (Kantengewicht) des Paares (u,v) von Spezies (siehe Abb 9.1)
Bestimmt man nun einen MST des Graphen G, erhält man einen Baum, mit Hilfe dessen man die evolutionäre Verwandtschaft zwischen den einzelnen Spezies erkennen kann.

9.2 Greedy-Strategie

Der einfachste Ansatz zur Berechnung eines Minimum Spanning Trees verwendet die sogenannte **Greedy-Strategie (Greedy Approach)** (greedy = gierig, gefräßig, habgierig):

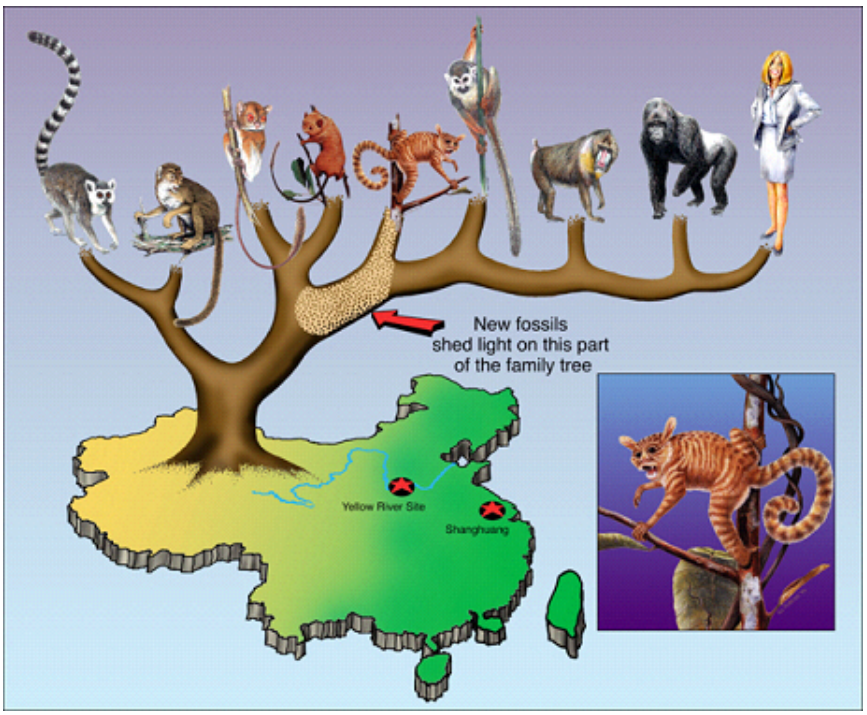


Abb. 9.1: Beispiel eines Stammbaumes von Affen. Am Abstand der Spezies kann man die evolutionäre Verwandtschaft erkennen.

In jedem Schritt wird mithilfe einer Heuristik unter mehreren Kanten die (scheinbar) beste Kante ausgewählt.
Dabei ist zu beachten, daß eine einmal getroffene Auswahl nicht mehr rückgängig gemacht werden kann. Aus diesem Grund liefert dieser Ansatz in vielen Anwendungen nur suboptimale Ergebnisse und nicht die (globale) optimale Lösung. Es existieren jedoch Greedy-Algorithmen, die die optimale Lösung des MST-Problems berechnen. Ein solcher Algorithmus wird im Folgenden diskutiert.
Wir betrachten einen generischen Algorithmus, der folgende Iterations-Invariante aufrechterhält: *Wir berechnen iterativ eine Kantenmenge A für die gilt: Vor und nach jeder Iteration ist A eine Teilmenge der Kanten eines MST.* Falls die Kantenmenge A nach einer gewissen Zahl von Iterationen einen aufspannenden Baum bildet, erhält man einen MST.

Definition 9.2.
Sei G = (V,E) ein Graph und A die Teilmenge eines MST von G.
Eine Kante (u,v) $\in$ E bezeichnet man als *sichere* Kante, wenn gilt:

$$A \cup \{(u,v)\} \text{ ist Teilmenge eines MST von G.}$$

In jedem Schritt des generischen Algorithmus fügen wir eine sichere Kante hinzu. Damit ergibt sich folgender Algorithmus:

Generic-MST(G,w):

```
(1) A = ∅
(2) Solange die Kanten in A keinen aufspannenden Baum bilden:
(3)      Suche eine sichere Kante (u,v) für A.
(4)      A = A ∪ {(u,v)}
(5) Gib A zurück
```

Beweis der Korrektheit.

Initialisierung:	Für die leere Menge ist die Invariante erfüllt.
While-Schleife:	Es werden nur sichere Kanten hinzugefügt. Damit folgt aus der Definition einer sicheren Kante, daß die Invariante erfüllt ist.
Programmende:	Alle Kante gehören zu einem MST und bilden einen aufspannenden Baum. Damit ist ein MST berechnet !

■

9.3 Kruskal's Algorithmus

In diesem Abschnitt wird Kruskal's Algorithmus vorgestellt. Doch zuerst müssen einige wichtige Begriffe eingeführt werden, die zum Verständnis dieses Algorithmus wichtig sind.

Definition 9.3.

1. Ein *Cut* (S,V\S) eines ungerichteten Graphen G = (V,E) ist eine Partition von V in zwei disjunkte Teilmengen S und V\S.

2. Eine Kante (u, v) *kreuzt* den Cut (S,V\S), wenn gilt:

$$(u \in S \wedge v \in V\backslash S) \vee (v \in S \wedge u \in V\backslash S)$$

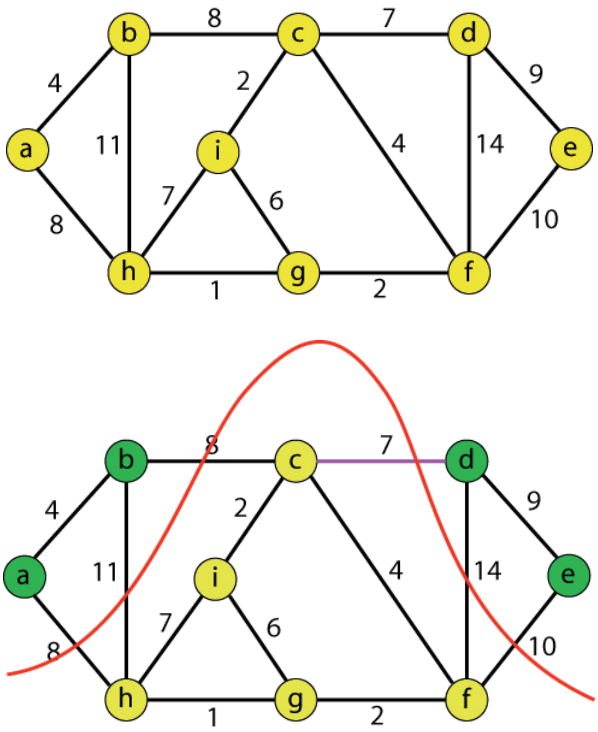
3. Ein Cut (S,V\S) *respektiert* eine Menge A⊆E, wenn gilt:

$$\nexists (u, v) \in (S, V\backslash S) \quad \text{mit } (u, v) \text{ kreuzt } (S, V\backslash S)$$

4. Eine Kante (u,v) eines Cuts (S,V\S) bezeichnet man als *leicht*, wenn gilt:

$$\text{(u,v) kreuzt nicht (S,V\S)} \quad \wedge \quad w((u,v)) \text{ ist minimal}$$

Beispiel 9.2.



Die rote Linie symbolisiert den Cut ({abde},{cfghi}). Die Kante (c,d) mit dem Gewicht 7 ist die leichte Kante dieses Cuts.

Satz 9.1 (Identifizierung sicherer Kanten).
Sei G = (V,E) ein zusammenhängender, ungerichteter Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$. *Sei A eine Kantenmenge, die Teilmenge eines minimalen aufspannenden Baumes von G ist. Sei (S,V\S) ein Cut der A respektiert und sei (u,v) eine leichte Kante, die (S,V\S) kreuzt. Dann gilt: (u,v) ist eine sichere Kante für A.*

Beweis. Sei T ein MST, der A aber nicht (u,v) enthält. Wir zeigen nun, daß es einen MST T' gibt, der sowohl A als auch (u,v) enthält.
Wir betrachten den Pfad von u nach v im MST T. Auf diesem Pfad muss mindestens eine Kante (x,y) existieren, die den Cut (S,V\S) kreuzt. Dies gilt aufgrund der Eigenschaft eines Cuts zusammen mit der Tatsache, daß (u,v) diesen Cut nach Voraussetzung kreuzt. Ausserdem gilt: (x,y)∉A, weil A den Cut respektiert.
Entfernt man nun (x,y) aus dem MST T, so zerfällt dieser in zwei Teilbäume. Fügt man die Kante (u,v) hinzu, so erhält man einen neuen Baum T':

$$T' = (T \backslash \{(x,y)\}) \cup \{(u,v)\}$$

Als nächstes zeigen wir, daß T' ein MST ist. Da (u,v) eine leichte Kante des Cuts (S,V\S) ist und (x,y) den Cut kreuzt, gilt:

$$w((u,v)) \leq w((x,y))$$

Daraus folgt:

$$w(T') = w(T) - w((x,y)) + w((u,v))$$

$\Rightarrow \qquad w(\text{T'}) \leq w(\text{T})$

Da T ein MST ist, muss folglich auch T' ein MST sein.
Bleibt noch zu zeigen, daß (u,v) eine sichere Kante für A ist. Da A $\subseteq$ T, folgt A $\subseteq$ T'. Damit ist auch A$\cup\{$(u,v)$\}$ $\subseteq$ T'. Da T' ein MST von G ist , ist folglich (u,v) eine sichere Kante für A. ■

Kommen wir nun zum eigentlichen Algorithmus von Kruskal. Die Grundidee dieses Algorithmus liegt darin, daß durch Hinzunahme von sicheren Kanten eine Menge von Bäumen solange ausgebaut wird, bis ein einziger minimaler aufspannender Baum entstanden ist.
Dazu führt man eine Menge aller Kanten, die nach aufsteigenden Kantengewichten sortiert werden, ein. Die Kanten werden innerhalb des Algorithmus beginnend mit der Kante mit dem kleinsten Gewicht abgearbeitet. Des Weiteren werden zur Initialisierung alle Knoten als Teilbäume betrachtet. Verbindet man zwei Teilbäume durch Hinzufügen einer sicheren Kante miteinander, so werden die entsprechenden Knotenmengen dieser Teilbäume zu einer Knotenmengen, also einem neuen Baum, verschmolzen. Diese sichere Kante wird zum entstehenden MST hinzugefügt.
Formuliert man dies im Pseudocode, erhält man folgenden Algorithmus:

Kruskals-MST(G,w):

```
(1) A = ∅
(2) Für jeden Knoten v∈V: Bilde_Menge(v)
(3) Sortiere alle Kante von E bezüglich der Gewichte
(4) Für jede Kante (u,v)∈E (in aufsteigender Reihenfolge):
(5)      Falls Menge(u) ≠ Menge(v):
(6)          A = A ∪ {(u,v)}
(7)          Vereinige Menge(u) mit Menge(v)
(8) Gib A zurück
```

Beweis der Korrektheit. Der Cut (Menge(u), V\Menge(u)) respektiert A und (u,v) ist eine leichte Kante, die (Menge(u), V\Menge(u)) kreuzt, falls (Menge(u) $\neq$Menge(v)). Damit folgt nach Satz 9.1, daß (u,v) eine sichere Kante ist. ■

Laufzeit des Algorithmus

Im Folgenden wollen wir uns die Laufzeit des Kruskal-Algorithmus näher ansehen.

Satz 9.2 (Laufzeit des Kruskal-Algorithmus). *Der Algorithmus von Kruskal zur Berechnung eines MSTs eines Graphen G=(V,E) mit n Knoten und m$\geq$n Kanten hat eine Laufzeit $\mathcal{O}$(m log(n)).*

Beweis. Das Sortieren aller Kanten von E kann man in $\mathcal{O}$(m log(m)). Für jede Kante in E muss ein Gleichheitstest durchgeführt werden. Ausserdem muss die Vereinigung aller Knotenmengen zu einer Menge erfolgen. Damit ergibt sich die Laufzeit eines MST von:

$$\text{MST(n,m)} = \mathcal{O}(\text{m log(m)} + \text{m} * \text{Gleichheitstest} + \text{n} * \text{Vereinige})$$

Mittels effizienter Datenstrukturen für die Mengenoperationen, z.B. mittels Heaps, kann man die Gleichheitstest und die Vereinige-Operationen in Zeit $\mathcal{O}(\log(n))$ durchführen. Damit folgt für die Laufzeit des MST:

$$\text{MST(n,m)} = \mathcal{O}(\text{m log(m)} + \text{m log(n)} + \text{n log(m)})$$

Da $\log(m) \leq \log(n^2) = 2\log(n) \in \mathcal{O}(\log(n))$, folgt weiter:

$$\text{MST(n,m)} = \mathcal{O}(\text{m log(m)} + \text{n log(n)})$$

und mit der Voraussetzung, daß $m \geq n$, folgt die Behauptung. ■

9.4 Prim's Algorithmus

In diesem Abschnitt wollen wir uns mit einem weiteren MST-Algorithmus, dem Algorithmus von Prim auseinander setzen. Dieser Algorithmus verfolgt einen ähnlichen Ansatz wie der Dijkstra-Algorithmus zur Suche eines kürzesten Weges in einem Graph (siehe Kap. 10.3 auf Seite 165).
Damit ergibt sich folgende Grundidee von Prim's Algorithmus: Man hat eine Menge A aus Kanten vom Graph G, die immer einen Baum bildet. Durch Hinzunahme von sicheren Kanten erhält man schließlich einen minimalen aufspannenden Baum.
Hierzu verwenden wir wiederum Satz 9.1, wobei bei einem Graphen G = (V,E) als Cut $(Q, V \setminus Q)$, gewählt wird. Dabei bilden die Knoten im Baum A die Menge Q.
Eine sichere Kante (u,v) ist eine Kante mit minimalem Gewicht, d.h. wir benötigen eine effiziente Datenstruktur zur Bestimmung des Knoten $v \in V \setminus Q$), der eine Kante mit minimalem Gewicht zu einem Knoten $u \in Q$ besitzt. Dazu verwenden wir eine Priority-Queue (z.B. Fibonacci-Heap siehe Kap.**??**). Zusammen mit dem entsprechenden Knoten v wird die Länge der aktuell kürzesten Kante von V zum bereits berechneten MST-Teilbaum A gespeichert. Die Länge repräsentiert die Priorität eines Elements in der Priority-Queue. Damit ergibt sich folgender Algorithmus im Pseudocode:

Prims-MST(G,w,r):

```
 (1) Q = V[G]
 (2) Für jeden Knoten u∈Q: Schlüssel[u] = ∞
 (3) Schlüssel[Wurzel] = 0
 (4) Vorgänger[Wurzel] = NIL
 (5) Solange Q ≠ ∅:
 (6)      u = Minimum(Q)
 (7)      Für alle adjazenten Knoten v von u:
 (8)          Wenn v∈Q und w(u,v) < Schl"ussel[v]:
 (9)               Schlüssel[v] = w(u,v)
(10)               Vorgänger[v] = u
```

Der Algorithmus liefert für jeden Knoten v seinen Vorgänger u = `Vorgänger[v]`.

Beweis der Korrektheit. In jedem Schritt wird eine leichte Kante (u,v), die den Cut (A,V\A) kreuzt, zu A hinzugefügt. Ausserdem respektiert der Cut (A,V\A) logischerweise A, womit nach nach Satz 9.1 folgt, daß (u,v) eine sichere Kante ist. Damit terminiert der Algorithmus und die Kanten in A bilden einen minimalen aufspannenden Baum. ■

Laufzeit des Algorithmus

Im Folgenden wollen wir uns die Laufzeit von Prim's Algorithmus näher ansehen.

Satz 9.3 (Laufzeit von Prim's Algorithmus). *Der Algorithmus von Prim zur Berechnung eines MSTs eines Graphen G=(V,E) mit n Knoten und m Kanten hat unter Verwendung eines Min-Heaps als Priority-Queue die Laufzeit $\mathcal{O}$(m log(n)). Verwendet man als Datenstruktur einen Fibonacci-Heap (siehe Kap.* **??***), so ist die amortisierte Laufzeit $\mathcal{O}$(m + n log(n)).*

Beweis. i) Verwendung eines Min-Heaps als Priority-Queue. Damit findet die Initialisierung in Zeile 1-4 in $\mathcal{O}$(V) Zeit statt. Die Schleife in Zeile 5 wird |V|-mal durchlaufen, wobei jedes Extract-Min von Q in einem binären Heap $\mathcal{O}$(log(V)) Zeit benötigt. Damit ist die gesamte Zeit für alle Aufrufe von Zeile 6 in $\mathcal{O}$(V log(V)). Die Schleife in Zeile 7 wird insgesamt $\mathcal{O}$(E)-mal ausgeführt. Zeile 9 enthält implizit eine Decrease-Key Operation im Heap, die bei einem binären Heap in der Zeit $\mathcal{O}$(log(V)) implementiert werden kann. Daraus folgt als gesamte Laufzeit des Algorithmus: $\mathcal{O}$(V log(V) + E log(V)) = $\mathcal{O}$(E log(V)).

ii) Verwendung eines Fibonacci-Heaps als Priority Queue. Wie in Kap. **??** gezeigt wurde, ist die Laufzeit der Extract-Min Operation in $\mathcal{O}$(log(V)) amortisierter Zeit und die Laufzeit der Decrease-Key Operation in $\mathcal{O}$(1) amortisierter Zeit. Damit erhält man als gesamte Laufzeit des Algorithmus mit Fibonacci-Heap $\mathcal{O}$(E + V log(V)).

■