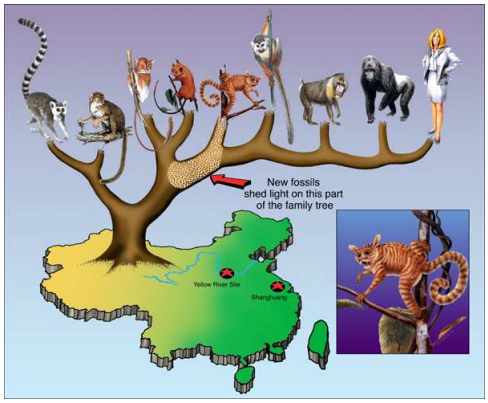
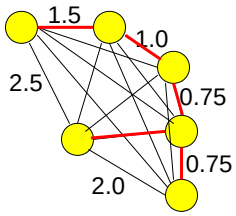
Minimum Spanning Tree

Minimum Spanning Tree: MST





Gegeben ein Graph G = (V, E). Die Knotenmenge V repräsentiere eine Menge von Spezies. Jede Kante e = (v,u) ist beschriftet mit der evolutionäre Distanz w(v,u) (Kantengewicht) des Paares (v, u) von Spezies.

Man bestimme den „minimalen aufspannenden Baum“ des Graphen.

Minimum Spanning Tree

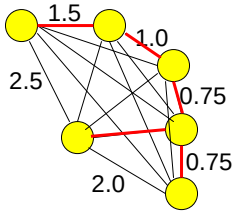
Definition [1]:

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

[a] Eine azyklischer Teilgraph T von G, der alle Knoten V von G verbindet, heißt ein auf-spannender Baum.

[b] Das Gewicht von T ist die Summe der Kanten-gewichte:

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



[c] Ein aufspannender Baum T von G mit minimalem Gewicht wird als minimaler aufspannender Baum von G (Minimum Spanning Tree) bezeichnet.

MST-Problem:

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

$$w : E \rightarrow \Re$$

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

Minimum Spanning Tree

Die Algorithmen zur Berechnung eines MST verwenden die sogenannte **Greedy-Strategie (Greedy Approach) (**greedy = gierig, gefräßig, habgierig)**.** In jedem Schritt des Algorithmus hat man die Wahl zwischen mehreren Kanten. Die Greedy-Strategie wählt die Kante, die in diesem Moment die (scheinbar) beste Wahl darstellt. In vielen Anwendungen liefern Greedy-Strategien nur suboptimale Resultate und nicht die (globale) optimale Lösung. Es existieren jedoch Greedy-Algorithmen, die die optimale Lösung des MST-Problem berechnen.

Wir werden zunächst einen generischen Algorithmus diskutieren, der die 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 also die Kantenmenge A nach einer gewissen Zahl von Iterationen einen aufspannenden Baum bildet, so haben wir einen MST berechnet.

Minimum Spanning Tree

In jedem Schritt fügen wir eine Kante (u,v) zu A hinzu, so dass $A \cup \{(u,v)\}$ auch diese Invariante erfüllt, d.h., eine Teilmenge der Kanten eines MSTs ist. Wir nennen eine solche Kante eine sichere Kante.

Generic-MST(G,w):

(1) $A = \emptyset$
(2) Solange die Kanten in A keinen aufspannenden Baum bilden:
(3) Suche eine sichere Kante (u,v) für A.
(4) $A = A \cup \{(u,v)\}$.
(5) Gib A aus (return A;).

Initialisierung:	Für die leere Menge ist die Invariante erfüllt.
Solange-Schleife (while):	Es werden nur sichere Kanten hinzugefügt.
Programmende:	Alle Kanten gehören zu einem MST und alle Kanten zusammen bilden einen auf-spannenden Baum => MST berechnet!

Wie findet man sichere Kanten?

Minimum Spanning Tree

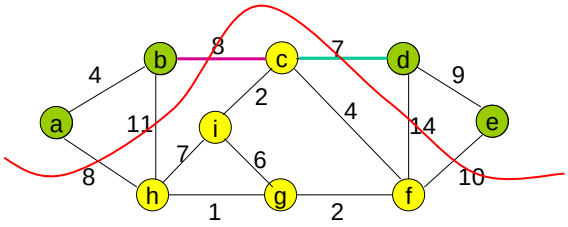
Definition [2]:

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

Eine Kante (u,v) kreuzt den Cut (S,V \ S), wenn ein Endpunkt in S und der andere in V \ S ist.

Ein Cut (S, V \ S) respektiert eine Kantenmenge A, wenn keine Kante aus A den Cut kreuzt.

Eine leichte Kante eines Cuts ist eine Kante, die den Cut kreuzt und ein minimales Gewicht besitzt.



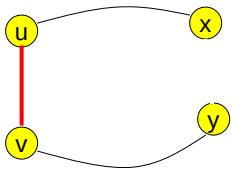
Minimum Spanning Tree

Der folgende Satz liefert eine Regel zur Identifizierung von sicheren Kanten.

Satz [1]:

Sei G=(V,E) ein zusammenhängender, ungerichteter Graph G=(V,E) mit Gewichtsfunktion w: E->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 ist (u,v) eine sichere Kante für A.

Beweis: Sei T ein MST, der A enthält. Wir nehmen an, dass T 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 es mindestens eine Kante (x,y) geben, die den Cut (S, V \ S) kreuzt.
Die Kante ist nicht in A, da A den Cut respektiert.

Entfernt man (x,y) aus dem MST T, so zerfällt dieser in zwei Teilbäume.

Addiert man die Kante (u,v), so erhält man einen neuen MST T‘:

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

Minimum Spanning Tree

Wir zeigen nun, dass T‘ ein MST ist: Da (u,v) eine leichte Kante des Cuts (S,V\S) ist und (x,y) auch den Cut kreuzt, gilt:

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

Hieraus folgt:

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

$w(T') \leq w(T)$

Da T ein MST ist, muss folglich auch T‘ ein MST sein.
Es bleibt zu zeigen, dass (u,v) eine sichere Kante für A ist.
Da $A \subseteq T$, ist $A \subseteq T'$. Also ist auch $A \cup \{(u,v)\} \subseteq T'$.
Da T‘ ein MST von G ist, ist folglich (u,v) eine sichere Kante für A. ■

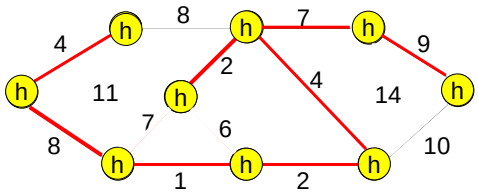
Minimum Spanning Tree

Kruskals Algorithmus zur Berechnung eines MST:

Der Algorithmus verwaltet eine Menge von Bäumen, die durch Hinzunahme weiterer sicherer Kanten solange ausgebaut werden, bis ein einziger minimaler aufspannender Baum berechnet wurde. Die Knoten eines jeden Teilbaums bilden eine Menge. Zu Beginn des Algorithmus ist jeder Knoten ein Baum (eine Menge).

Die Kantenmenge wird vor Ausführung der eigentlichen Iteration nach aufsteigendem Kantengewicht sortiert. Die Kanten werden in dieser Reihenfolge abgearbeitet, beginnend mit der Kante mit kleinstem Gewicht.

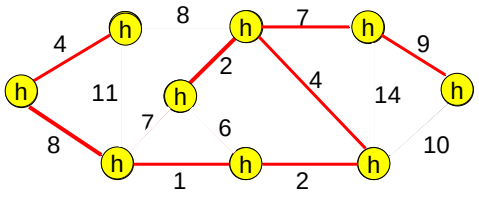
Werden zwei Teilbäume durch Hinzufügen einer (sicheren) Kante miteinander verbunden, so werden die entsprechenden Mengen miteinander zu einer neuen Menge (einem neuen Baum) verschmolzen und die Kante wird in A aufgenommen.



Verbindet die aktuelle Kante zwei Knoten des gleichen Teilbaums (der gleichen Menge), so wird die Kante nicht zu A hinzugefügt.

MST-KRUSKAL(G,w):

(1) A = ∅
(2) Für jeden Knoten v∈ V: Bilde_Menge(v)
(3) Sortiere alle Kanten 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.



Korrektheit: Berechnet MST-KRUSKAL wirklich einen MST von G?

(1) A = ∅
(2) Für jeden Knoten v∈ V: Bilde_Menge(v)
(3) Sortiere alle Kanten 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.

Satz [1]:

Sei G=(V,E) ein zusammenhängender, ungerichteter Graph G=(V,E) mit Gewichtsfunktion w: E->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 ist (u,v) eine sichere Kante für A.

Da der Cut (Menge(u), V \ Menge(u)) A respektiert und (u,v) eine leichte Kante ist, die (Menge(u), V \ Menge(u)) kreuzt (falls Menge(u) ≠ Menge(v)), ist (u,v) eine sichere Kante für A.

Minimum Spanning Tree

11

```
// LEDA Implementierung von MST-Kruskal

list<edge> MIN_SPANNING_TREE(      const graph& G,
                                   int (*cmp)(const edge&, const edge &))  {
        list <edge> T;                      //  LEDA Listen
        node_partition P(G);
        list<edge>  L  = G.all_edges( );   // LEDA Listen
        L.sort(cmp);
        edge e;

        forall(e,L) {
                node u = source(e);
                node v = target(e);
                if (! P.same_block( u,v)) {
                        T.append(e);              // LEDA: anstatt push_back
                        P.union_blocks(u,v);
                }
        }
        return T;
}
```

Minimum Spanning Tree

12

Laufzeit MST-KRUSKAL

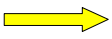
(1)	A = ∅	O(1)
(2)	Für jeden Knoten v∈ V: Bilde_Menge(v)	O(nlog(n))
(3)	Sortiere alle Kanten von E bezüglich der Gewichte.	O(m log(m))
(4)	Für jede Kante (u,v) ∈ E	O(m)
(5)	Falls Menge(u) ≠ Menge(v):	O(m*Gleichheitstest)
(6)	A = A ∪ {(u,v)}	O((n-1)*1)
(7)	Vereinige Menge(u) mit Menge(v)	O((n-1) *Vereinige)
(8)	Gib A zurück.	O(1)

Als Laufzeit MST für einen Graphen mit n Knoten und m Kanten erhält man :

MST(n,m) = O(m log(m) + m * Gleichheitstest + n * Vereinige)

Mittels effizienter Datenstrukturen für die Mengenoperationen (Heaps), kann man die Gleichheitstests und die Vereinige-Operationen in Zeit O(log(n)) durchführen. Hieraus folgt:

MST(n,m) = O(m log(m) + n log(n))



Minimum Spanning Tree

Da $\log(m) \leq \log(n^2) = 2\log(n) \in O(\log(n))$ ist, folgt für die Laufzeit weiter:

$$MST(n,m) = O(m \log n + n \log n)$$

Da der Graph nur dann einen aufspannenden Baum besitzt, wenn die Zahl der Kanten m größer gleich n-1 ist, können wir die obige Laufzeit noch vereinfachen:

$$MST(n,m) = O(m \log n)$$

Satz [2]:

Der Algorithmus von Kruskal zur Berechnung eines MSTs eines Graphen G=(V,E) mit n Knoten und $m \geq n$ Kanten hat Laufzeit $O(m \log n)$.

Minimum Spanning Tree

Prim's Algorithmus zur Berechnung eines MST:

Bei Prim's Algorithmus bilden die Kanten in A immer einen Baum.
Zu diesem Baum werden solange sichere Kanten hinzugefügt, bis man einen (minimalen) aufspannenden Baum hat.
Um die sicheren Kanten zu finden, wenden wir wiederum Satz [1] an:

Satz [1]:

Sei G=(V,E) ein zusammenhängender, ungerichteter Graph G=(V,E) mit Gewichtsfunktion w: E->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 ist (u,v) eine sichere Kante für A.

Als Cut wählt Prim's Algorithmus (Q, V \ Q), wobei Q die Menge der Knoten im Baum A ist.

Eine sichere Kante (u,v) ist daher eine Kante mit minimalen Gewicht, die einen Knoten u von Q mit einem Knoten v von V\Q verbindet.

Wir benötigen daher eine effiziente Datenstruktur zur Bestimmung des Knotens $v \in V\setminus Q$, der den minimalen Abstand (Kante mit minimalem Gewicht) zu einem Knoten $u \in Q$ besitzt.

Minimum Spanning Tree

Prim's Algorithmus zur Berechnung eines MST:

Die Rückgabe des MST erfolgt über das Knotenfeld „parent". Hier wird für jeden Knoten der Elternknoten im berechneten MST gespeichert.

Die Gewichte der Kanten werden in unserer Implementierung mittels eines Kantenfelds „weight" an die Prozedur übergeben.

Die MST-Berechnung geht von dem Knoten r aus.

```
void  MST-PRIM(        const graph&  G, node  r,
                       const edge_array<float>&  weight,
                       node_array<node>&  parent)   {
```

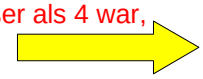
Minimum Spanning Tree

Prim's Algorithmus zur Berechnung eines MST:

```
p_queue<float, node> PQ;
```

Zur Speicherung der Knoten, die als nächstes über eine sichere Kante zum Baum hinzugefügt werden können, verwenden wir eine LEDA-Priority-Queue (Fibonacci-Heap-Implementierung). 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 (Items) in der Priority-Queue.

```
pq_item  it = PQ.insert(6, v);    //  fügt ein neues Element hinzu
float  length = PQ.del_min( );    //  löscht das Element mit min. Priorität
pq_item  it = PQ.find_min( );     //  gibt das Element mit min. Priorität zurück
float length = PQ.prio(it);       //  gibt die Priorität von Element it zurück
node u  =  PQ.inf(it);            //  gibt den Knoten von Element it zurück
PQ.decrease_p(it, 4);             //  verkleinert die Priorität von it auf 4, falls
                                  //  die Priorität von it größer als 4 war,
                                  //  sonst Fehlermeldung
```



Minimum Spanning Tree

17

Prim's Algorithmus zur Berechnung eines MST:

```
void  MST-PRIM(          const graph&  G, node  r,
                         const edge_array<float>&  weight,
                         node_array<node>&  parent)   {

    p_queue<float, node> PQ;
    //  Welche Knoten sind bereits abgearbeitet?
    node_array<bool>   ready(G, false);
    // Zu jedem Knoten in der PQ speichern wir das entsprechende Element.
    node_array<pq_item>  I(G);

    edge e;
    forall_adj_edges(e , r)  {
          I[G.target(e)] = PQ.insert(weight[e], G.target(e));
          parent[G.target(e)] = r;                                  O(n log n)
    }
    ready[r] = true;
```

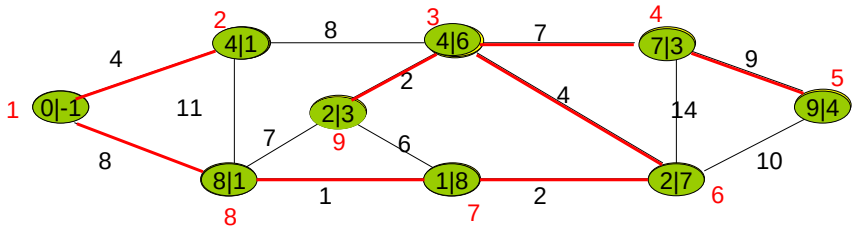
Minimum Spanning Tree

18

```
          while (! PQ.empty())  {
                    node u  = PQ.inf(PQ.find_min());  O(n * find_min) = O(n log n)
                    PQ.del_min( );
                    ready[u] = true;

                    forall_adj_edges(e,u)  {
                              node v = G.target(e);
                              if  ( ! ready[v])  {
                                        if ( parent[v] == nil ) {
    O(n * insert) = O(n log n)                  I[v] = PQ.insert(weight[e], v);
                                                parent[v] = u;
                                        }
                                        else if  ( weight[e] < PQ.prio(I[v])) {
    O(m * decrease_p) = O(m)                    PQ.decrease_p(I[v], weight[e]);
                                                parent[v] = u;
                                        }
                              }
                    }
          }
}
```

Minimum Spanning Tree



Minimum Spanning Tree

Satz [3]:

Der Algorithmus MST_PRIM berechnet den MST eines Graphen G=(V,E) mit n Knoten und m Kanten (falls man einen Min-Heap) verwendet in Zeit O(mlog(n)). Verwendet man als Datenstruktur einen sogenannten Fibonacci-Heap, so ist die amortisierte Laufzeit O(m + n log(n)).

Bemerkung: Fibonacci-Heaps erlauben das Einfügen von neuen Elementen und die Minimumsberechnung in einer n-elementigen Menge in amortisierter Zeit O(log(n)). Die anderen Operationen im Algorithmus MST_PRIM können in amortisierter Zeit O(1) durchgeführt werden.