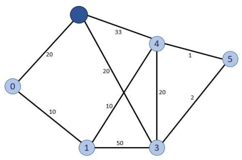
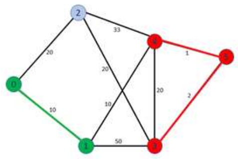
Minimum Cost Spanning Trees

MCST-(Minimale Spannbäume)- Bestimmt ein zusammenhängender azyklischen bewerteten Baum eines ungerichteten Graph G, bei dem die Summe aller Kantengewichte minimaler wird.





Praktische Anwendungen

MCSTs werden in verschieden Bereiche angewendet, um Z.B. Lösungen für diversen Problemen zu finden oder Daten Abzubilden.

- Entwurf von Netzwerken:
  Telefon, Rechner, Gas, Wasser, Transport.
- Schaltkreisen optimieren:
  für Platinen, Chips, elektrische Systeme.
- Clusteranalyse:
  Sozialmedien analysieren, Marktforschung,

Algorithmus von Prim

```
BEGIN
  T ← Ø
  FOR i ← 1, ..., n DO
     d(v[i]) ← ∞, parent(v[i]) ← NULL
  d(v[1]) ← 0, parent(v[1]) ← v[1]
  WHILE queue ≠ Ø DO
     u ← queue.extractMin()
     IF parent(u) ≠ u
        T.add({parent(u), u})
     FOR ALL {u, w} ∈ E do
        IF w ∈ queue AND l(u, w) < d(w) THEN
           d(w) ← l(u, w), parent(w) ← u
        ELSE IF parent(w) = NULL THEN
           d(w) ← l(u, w), parent(w) ← u
           queue.insert(w)
END
```

Der Baum ist leer und der Abstand aller Knoten wird auf unendlich gesetzt und die Knoten haben kein parent.

Ein Startknoten wird als Wurzel des MST gesetzt und in die Warteschlange eingefügt, dann wird der Abstand der Wurzel gleich 0 gesetzt und sich selbst als parent eingetragen.

Der Knoten mit dem kleinsten Wert wird aus der Warteschlange entfernt und seine Kante, welche ihn dem MST verknüpft, wird dazu hinzugefügt. Mit Ausnahme der Wurzel, die selber ihre parent ist.

Für jede angrenzenden Knoten werden potenzielle Änderungen beobachtet und wenn der Knoten schon zum MST hinzugefügt wurde oder der Knoten schon in der Warteschlange ist aber mit der bereits niedrigerer Kante, die mit dem Baum verbunden ist, dann wird diese Knoten nicht betrachtet.

Falls ein Knoten noch nicht zum MST hinzugefügt wurde, muss er dann später behandelt werden. Dieser wird in der Warteschlange hinzugefügt.

Wenn alle Kante behandelt sind ist der Algorithmus fertig.

Algorithmus von Kruskal

```
BEGIN
   T ← Ø
   queue ← sort {u, v} edges of E using l.
   FOREACH v in G.V
      make-tree(v);
   WHILE queue ≠ Ø AND trees-count > 1
DO
      {u, v} ← queue.extractMin()
      IF !(T ∪ {{u, v}} has cycle)
         T.add({u, v})
         merge(tree-of(u), tree-of(v))
END
```

Die Kanten werden nach ihrem Gewicht sortiert, so dass sie in konstanter Zeit aus der Warteschlange entfernt werden können.

Jeder Knoten ist ein loser Baum in Graph. Dann werden die sortierte Kante mit dem geringsten Gewicht aus der Warteschlange entfernt und nur markiert im Graph.

Falls diese Kante die Verbindung zwei Knoten ist, welche zum selben Baum gehören, wird sie ignoriert und nicht zum Baum hinzugefügt, sonst entsteht ein Kreis.

Falls die Endpunkte der Kante zu zwei verschiedene Bäume gehören dann werden sie vereinigt.

Der Prozess geschieht solange Kanten in der Warteschlage sind und mehr als Baum gibt.

Laufzeit der Algorithmen

Implementierung priority queue bei Prim

PQinsert(): V Knoten.

PQisempty(): V Knoten.

PQdelmin(): V Knoten.

PQdeckey():E Kanten.

	Priority Queues		
Operation	**Array**	**Binary heap**	**Fibonacci heap***
insert	N	log N	1
delete-min	N	log N	log N
decrease-key	1	log N	1
is-empty	1	1	1
Prim	V^2	E log V	E + V log V

Laufzeit bei Kruskal

- Kruskal: Analyse der Operationen O(E log V).
- Sort(): O(E log E) = O(E log V).
- UFinit(): V Singleton-Sets.
- UFfind(): höchstens einmal pro Kante.
- UFunion(): genau V –1.
- Bei bereits sortierte Kante: O(E log* V).