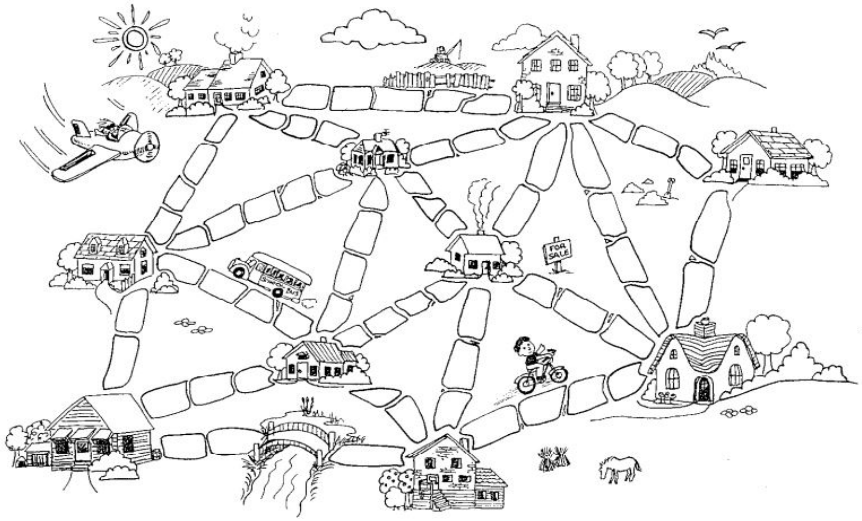
Problemstellung

Vor langer Zeit gab es eine Stadt, die keine Straßen hatte. Sich in dieser Stadt zu bewegen war insbesondere dann schwierig, wenn der Regen nach starken Gewittern die Wege sehr matschig machte. Fahrzeuge blieben stecken und jeder hatte ziemlich dreckige Schuhe. Daher entschied der Bürgermeister, einige Straßen zu asphaltieren. Allerdings wollte er nicht mehr Geld ausgeben als unbedingt nötig, damit auch noch ein Schwimmbad gebaut werden konnte.



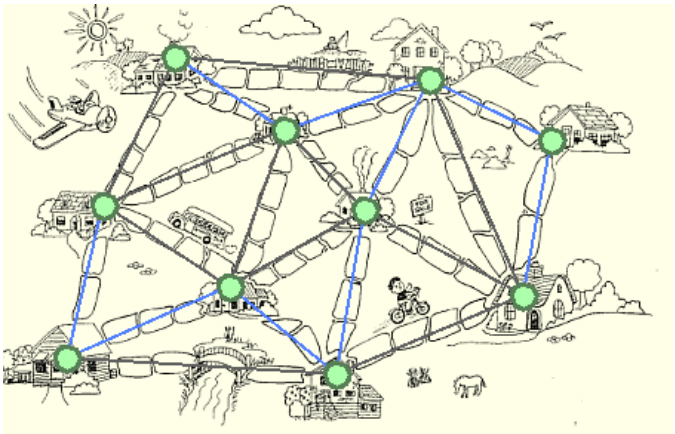
Quelle: Minimal Spanning Tree Activity, csunplugged.org (Lizenz: CC BY-NC-SA 4.0), URL: https://classic.csunplugged.org/minimal-spanning-trees/#The_Muddy_City (abgerufen 28.10.2020)

Also legte er zwei Bedingungen fest:

- Es müssen ausreichend Straßen asphaltiert werden, damit jeder von seinem Haus jedes andere Haus erreichen kann.
- Das Asphaltieren sollte so wenig kosten wie möglich. Die Anzahl der Pflastersteine ist das Maß für die Kosten.

Aufgaben zum Muddy-City-Problem:

1. *Finde die Wege, die alle Häuser miteinander verbinden und deren Asphaltierung am wenigsten kostet (=Pflastersteine beinhaltet).*



Die optimale Lösung besteht aus 23 Pflastersteinen.

2. *Welche Strategien hast du genutzt, um das Problem zu lösen?*

Modellierung

Die Ausgangssituation soll nun als Graph modelliert werden.

Modellierung

Knoten:

Die Knoten repräsentieren die .

Kanten:

Zwei Knoten sind durch Kanten verbunden, wenn

Minimaler Spannbaum (Minimal spanning tree)

Gegeben ist ein zusammenhängender, gewichteter, ungerichteter Graph.
Gesucht ist die Teilmenge der Kanten, so dass der Graph zusammenhängend bleibt und die Summe der Kantengewichte möglichst klein ist.

Weiterführende Fragen

Aufgaben:

1. *Wende einen Algorithmus zur Bestimmung des minimalen Spannbaums auf die Karte der größten Städte in Deutschland an (02_deutschlandkarte.csv). Beachte, dass nur markierte Kanten sichtbar sind, da es zu viele Kanten gibt, um sie alle übersichtlich darzustellen. Vergleiche mit dem Autobahnnetz in Deutschland. Nenne Ursachen für Unterschiede.*
2. *Öffne den Stadtplan von Baden-Baden (03_badenbaden.csv) und die Karte mit den Fährstrecken (04_inseln.csv). Gib je eine praktische Aufgabenstellung für diese Graphen an, bei denen ein minimaler Spannbaum die beste Lösung ist.*

   Stadtplan: z.B. Wasserrohre, Gasleitungen, Netzwerkkabel verlegen.
   Inselkarte: Beschränkung auf Fährverbindungen mit einer möglichst kurzen Gesamtstrecke, so dass trotzdem alle Inseln verbunden sind.
3. *Untersuche, ob der Algorithmus zur Bestimmung des minimalen Spannbaums auch mit negativen Kantengewichten zurechtkommt.*

   Negative Kantengewichte stellen kein Problem dar, da es beim Algorithmus nur auf die Sortierreihenfolge der Kanten ankommt und nicht auf den Wert des Gewichts. Diese Sortierung funktioniert auch bei negativen Kantengewichten.
4. *Begründe, warum es sich bei den Algorithmen zur Bestimmung des minimalen Spannbaums um Greedy-Algorithmen handelt.*

   Ein Greedy-Algorithmus ist der Kruskal-Algorithmus, da in jedem Schritt die günstigste Kanten ausgewählt wird, die nicht zu einem Zyklus führt.
   Prim ist ein Greedy-Algorithmus, da immer die kürzeste Kante gewählt wird, um den Baum zu erweitern.
5. *Der Algorithmus zur Bestimmung des minimalen Spannbaums ist ein Greedy-Algorithmus liefert aber trotzdem die optimale Lösung und keine Näherungslösung. Informiere dich z.B. bei Wikipedia über den Beweis zur Korrektheit.*

Beim Traveling Salesman Problem (TSP) sucht man nach der kürzesten Route durch eine gegebene Liste von Städten, die ein Handlungsreisender besuchen muss (vgl. Problemstellung 2, AB Repräsentation von Graphen). Am Ende soll er wieder zu Hause ankommen.

Für dieses Problem kennt man keinen effizienten Algorithmus. Man kann aber die Algorithmen zur Bestimmung eines MST abändern, so dass sie eine Näherungslösung für das TSP darstellen.

6. *Beschreibe die Veränderungen die am Algorithmus für den MST notwendig sind, um das TSP zu lösen. Du kannst dazu im Graphentester die Algorithmen "TSP (Greedy: Knoten)" und "TSP (Greedy: kürzeste Kante)" zu Hilfe nehmen.*

   Der Prim-Algorithmus vereinfacht sich, da die kürzeste Kante von einem markierten zu einem unmarkierten Knoten für das TSP nicht im ganzen bisherigen Baum gesucht werden muss, sondern am Ende der Route liegen muss. Man muss also nur die ausgehenden Kanten des Routenendes zu allen noch nicht markierten Knoten betrachten (vgl. TSP-Greedy im Graphentester). Am Ende wird die Rundreise durch eine Kanten zwischen den beiden Blättern des Baums geschlossen.

   Der Kruskal-Algorithmus wird etwas komplizierter, da man beim Zusammenführen zweier Teilbäume oder Vergrößern eines Teilbaums darauf achten muss, dass keine innere Knoten sondern nur Blätter verwendet werden dürfen. Außerdem muss zum Schließen der Rundreise am Ende einmal erlaubt werden, einen Zyklus zu bilden.

Beschreibung des Algorithmus (Kruskal)

Der Algorithmus verfolgt die Idee, immer die kürzeste Kante des Graphen dem Baum hinzuzufügen, wenn dadurch nicht ein Zyklus entsteht und die Kante damit überflüssig ist. Es entstehen dabei zunächst viele einzelne Bäume (ein Wald), die sukzessive zu einem einzigen Baum zusammengeführt werden. Um die Zyklen leicht erkennen zu können, werden die Knoten jedes einzelnen Baums in einer eigenen Farbe eingefärbt.

Zunächst werden die Kanten also nach ihrem Gewicht sortiert und dann für jede Kante folgende Regeln beachtet:

- haben beide Knoten noch keine Farbe (Farbe 0), dann gehören sie bisher zu keinem Baum und werden beide in einer noch nicht benutzen Farbe gefärbt.
- hat ein Knoten eine Farbe und der andere noch nicht, wird der Baum des farbigen Knotens erweitert. Die Farbe wird für den noch nicht gefärbten Knoten übernommen.
- haben beide Knoten eine unterschiedliche Farbe, dann werden zwei Bäume zu einem vereinigt. Alle Knoten des zweiten Baums werden mit der Farbe des ersten eingefärbt.
- haben beide Knoten die gleiche Farbe, gehören sie zum gleichen Baum. Die Kante darf nicht eingefügt werden, da ein Zyklus entstehen würde.

Außer in Fall 4 wird außerdem die Kante markiert, da sie zum minimalen Spannbaum gehört.

Beschreibung des Algorithmus (Prim)

Dieser Algorithmus startet mit einem beliebigen Knoten. Dieser wird markiert. Sukzessive werden an diesen Knoten weitere Knoten angebunden und so allmählich ein immer größerer Baum aufgebaut. Dabei wird derjenige Knoten dem Baum hinzugefügt, zu dem die kürzeste Kante vom bisherigen Baum führt. Diese Kante wird markiert.

Auch hier startet man damit, die Kanten nach ihrem Gewicht zu sortieren. Nun sucht man in dieser Liste nach einer Kante, bei der ein Knoten markiert ist und der andere nicht. Dabei kann man alle Kanten, bei denen beide Knoten markiert sind, streichen, da sie nie mehr gebraucht werden.

Hat man eine derartige Kante gefunden, wird sie markiert und aus der Liste entfernt. Der neu angebundene Knoten wird ebenfalls markiert. Diese Schritte wiederholt man, bis alle Knoten angebunden sind, also n-1 mal bei n Knoten.

Pseudocode des Algorithmus (Kruskal)

```
Minimal spanning tree:
Setze farbnummer auf 1
Hole eine Liste aller Kanten und sortiere sie aufsteigend
Wiederhole für jede Kante
  k1 = Startknoten, k2 = Zielknoten der Kante
  Falls Farbe von k1==0 und Farbe von k2==0
    Setze Farbe beider Knoten auf farbnummer
    Markiere die Kante
    Erhöhe die Farbnummer um 1
  Sonst
    Falls Farbe eines Knotens==0
      Setze Farbe dieses Knotens auf die des anderen
      Markiere die Kante
    Sonst
      Falls beide Knoten unterschiedliche Farben haben
        Wiederhole für jeden Knoten k im Graph
          Falls Farbe von k die Farbe von k2 hat
            Setze Farbe von k auf die Farbe von k1
          Ende-Falls
        Ende-Wiederhole
        Markiere die Kante
      Sonst
        Lösche Kante
      Ende-Falls
    Ende-Falls
```

Pseudocode des Algorithmus (Prim)

```
Minimal spanning tree:
Hole eine Liste aller Kanten und sortiere sie aufsteigend
Setze einen beliebigen Knoten auf markiert
Wiederhole n-1 mal
  Wiederhole für jede Kante ka der Liste
    Falls Markierung des Startknotens != Markierung des Zielknotens
      naechsteKante = ka
      brich Schleife ab
    Ende-Falls
  Ende-Wiederhole
  Markiere naechsteKante und lösche sie aus der Liste
  Markiere Start- und Zielknoten
Ende Wiederhole
```

Quelltext des Algorithmus (Kruskal)

```
int farbe = 1;
List<Kante> kanten = g.getAlleKanten();
List<Knoten> knoten = g.getAlleKnoten();
Collections.sort(kanten);

for (Kante k: kanten) {
  int f1 = k.getStart().getFarbe();
  int f2 = k.getZiel().getFarbe();
  if(f1 == 0 && f2 == 0) {
    k.getStart().setFarbe(farbe);
    k.getZiel().setFarbe(farbe);
    k.setMarkiert(true);
    farbe++;
  } else
  if(f1 == 0) {
    k.getStart().setFarbe(f2);
    k.setMarkiert(true);
  } else
  if(f2 == 0) {
    k.getZiel().setFarbe(f1);
    k.setMarkiert(true);
  } else
  if(f1 == f2) {
    k.setGeloescht(true);
  } else
  {
    for(Knoten k1 : knoten) {
      if(k1.getFarbe() == f2) k1.setFarbe(f1);
    }
    k.setMarkiert(true);
  }
}
```

Quelltext des Algorithmus (Prim)

```
List<Knoten> knoten = g.getAlleKnoten();
List<Kante> kanten = g.getAlleKanten();
Collections.sort(kanten);
knoten.get(0).setMarkiert(true);

for(int i = 1; i<knoten.size(); i++) {
  Kante ka=null;
  for(Kante ka2 : kanten) {
    if(ka2.getStart().isMarkiert() != ka2.getZiel().isMarkiert()) {
      ka = ka2;
      break;
    }
  }

  ka.setMarkiert(true);
  kanten.remove(ka);
  ka.getStart().setMarkiert(true);
  ka.getZiel().setMarkiert(true);
}
```