Programmieren C: Kruskal-Algorithmus

Praktisch:

Gegeben sind *Städte auf einer Landkarte*. Gesucht ist ein *Leitungsnetz*,
an das *alle Städte angeschlossen* sind, und das mit *minimaler Leitungslänge* auskommt.

Theoretisch:

Gesucht ist diejenige *Teilmenge der Kanten* eines *ungerichteten, kantengewichteten Graphen*,
die den *"Minimum Cost Spanning Tree"* zu diesem Graphen darstellt. *"Minimum Cost"* heißt,
dass die *Summe der Gewichte* (hier: Längen) der ausgewählten Kanten *minimal* ist.
"Spanning" heißt, dass die Lösung den Graph aufspannt, d.h. das *alle Knoten des Graphen*
über die ausgewählten Kanten *verbunden bzw. erreichbar* sind. Und *"Tree"* deshalb,
weil die Lösung immer ein Baum ist, d.h. *keine Zyklen enthält* (würde die Lösung
einen Zyklus enthalten, könnte man eine Kante des Zyklus weglassen, und hätte dadurch
eine billigere Kantenmenge gefunden, die immer noch alle Knoten verbindet).
Die Lösung für *n Knoten* hat daher *immer n-1 Kanten*.

Das Problem wäre zwar z.B. auch mittels Backtracking lösbar, aber es gibt dafür
sehr viel *effizientere Verfahren*, z.B. die *Algorithmen von Prim* und *von Kruskal*.
Für *"dünne" Graphen* (die im Vergleich zur Knotenzahl *nur wenige Kanten haben*) ist
der *Kruskal-Algorithmus* effizienter, für *"dichte" Graphen* (die in Relation zur Knotenzahl
viele oder sogar alle möglichen Verbindungen enthalten) ist der *Prim-Algorithmus* besser.

Unser Graph ist zwar dicht (denn wir betrachten alle möglichen Verbindungen,
d.h. von jeder Stadt prüfen wir die direkte Verbindung zu jeder anderen Stadt),
aber da eine effiziente Implementierung des Prim-Algorithmus etwas umfangreicher ist,
bleiben wir beim Kruskal-Algorithmus.

Lösungsidee:

Die Idee des Kruskal-Algorithmus ist ganz einfach:

- Betrachte *eine Verbindung nach der anderen*, und zwar *in der Reihenfolge ihrer Länge*
  (die *kürzesten zuerst*).
- Wenn die Verbindung *zwei Städte* verbindet, die *bisher noch nicht verbunden* sind,
  dann gehört die Kante *zur Lösung*, sonst wird die Verbindung ignoriert.
- *Wiederhole* das, bis *n-1 Verbindungen* für die Lösung gefunden sind.

Das Verfahren hat zwei Knackpunkte:

- Das ***Ermitteln der nächst-kürzesten Kante***. Hier sind wir faul:
  Wir *sortieren das Array aller Kanten* zu Beginn einmal nach Kantenlänge
  und gehen es dann der Reihe nach durch.

  Es gäbe *effizientere Verfahren* (weil wir ja bei weitem nicht alle Kanten brauchen,
  sondern *nur den vorderen Teil des Arrays* abarbeiten, bis genug Kanten gewählt sind:
  Die Sortierung der restlichen Kanten ist eigentlich unnötig),
  aber deren Implementierung ist uns *zu aufwändig*.
- Die ***Prüfung, ob eine Kante zwei noch nicht verbundene Städte verbindet***.

  Hier nutzen wir das effiziente Verfahren, die sogenannte *Union-Find-Datenstruktur*
  (allerdings nur in der Grundversion, wir lassen alle Optimierungen weg).

Die Union-Find-Idee:

- *In jeder Stadt* speichern wir einen zusätzlichen *Pointer auf eine andere Stadt*.
- Zu jedem Zeitpunkt gilt:
  - Jeder solche Pointer zeigt immer auf eine Stadt,
    zu der *schon eine Verbindung* hergestellt wurde (direkt oder indirekt).
  - In jeder *schon verbundenen Gruppe von Städten* gibt es *genau eine Stadt*,
    bei der dieser Pointer **NULL** ist.
  - Für *jede andere Stadt* in dieser Gruppe gilt: *Folgt man immer wieder dem Pointer*,
    kommt man irgendwann zu der Stadt mit dem **NULL**-Pointer.

  Jede Gruppe von verbundenen Städten wird also durch ihre **NULL**-Pointer-Stadt *repräsentiert*, und man kann für jede Stadt die repräsentative Stadt ihrer Gruppe *ermitteln*, indem man *den Pointern folgt*, bis man bei **NULL** ist.
- Am Anfang gibt es *noch keine Verbindungen*,
  d.h. jede Stadt bildet *ihre eigene Gruppe* und ist ihr eigener Repräsentant.
  Daher werden *am Anfang alle Pointer auf* **NULL** gesetzt.
- Wenn die gerade *zu prüfende Verbindung* die Städte *A* und *B* verbindet,
  dann berechnet man die *"repräsentativen Städte"* für *A* und für *B*,
  wir nennen sie einmal *RootA* und *RootB*.
  - Kommt dabei für *RootA* und *RootB* *dieselbe Stadt* heraus,
    dann gehören *A* und *B* zur *selben Gruppe*, sind also *schon verbunden*.
    Die Verbindung wird daher *ignoriert*.
  - Sind *RootA* und *RootB* *verschiedene Städte*,
    dann sind *A* und *B* *noch nicht verbunden*.
    Die Verbindung wird daher *in die Lösung aufgenommen*.

    Dann müssen die *beiden Gruppen* von *A* und *B* *zu einer Gruppe vereinigt* werden.
    Das erreicht man, indem man *in RootA* (dessen Pointer ja bisher noch **NULL** ist)
    einen Pointer auf RootB speichert. Damit wird *RootB* der Repräsentant
    der vereinigten Gruppe, und man landet bei *RootB,* wenn man
    die Pointer ausgehend von einer Stadt der bisherigen *A*-Gruppe verfolgt.

Im Detail:

- Unser Programm soll die Städte und die Verbindungen der Lösung
  mittels SDL-Grafik anzeigen.

  Unsere Städte haben daher *x/y-Koordinaten*, die ihrer *Pixel-Position* entsprechen,
  und unsere Verbindungen haben diejenige *Länge*,
  die sich *geometrisch aus den Koordinaten* der beiden verbundenen Städte ergibt.
- Zuerst einmal deklarieren wir zwei **`struct`**-*Typen*:
  - Für eine ***Stadt***: Zwei **`int`**'s für die *x- und y-Koordinate*
    und ein *Stadt-Pointer* für die *Union-Find-Logik*.
  - Für eine ***Verbindung***: *Zwei Stadt-Pointer* für die beiden verbundenen Städte
    und ein **`double`** für die *Länge*.

- Dann brauchen wir drei *Hilfsfunktionen*:
  - Eine für die *Ermittlung der "repräsentativen" Stadt*:

    Sie wird mit einem *Stadt-Pointer* aufgerufen und folgt in einer *Schleife* immer wieder dem Union-Find-Pointer der aktuellen Stadt, bis sie bei einer Stadt ist, deren Union-Find-Pointer **`NULL`** ist.

    Die Funktion liefert einen *Pointer* auf die *gefundene Stadt als Returnwert*.
  - Eine *Vergleichsfunktion* für das Sortieren des Verbindungs-Arrays mit **`qsort`** (welche Parameter und welchen Returntyp muss diese Funktion haben?).

    Die beiden Pointer-Parameter sind in Wirklichkeit *Pointer auf Verbindungen*, und die *Längen-Member* der beiden Verbindungen werden *verglichen*: Ist die *erste kürzer*, liefert die Funktion **`-1`**, ist die *zweite kürzer*, liefert sie **`1`**, und sonst (bei *Gleichheit*) liefert sie **`0`**.
  - Und eine für die *Berechnung der Verbindungslängen*:

    Sie wird mit *zwei Stadt-Pointern* aufgerufen und liefert einen **`double`**-Returnwert: Deren *Entferung*, berechnet *aus der Differenz der x- und der y-Koordinaten mittes Pythagoras* (die Wurzel-Funktion heißt **`sqrt`** und kommt aus **`math.h`**).
- Dann kommt schon das ***Hauptprogramm***.
  Es wird mit *einer positiven, ganzen Zahl* auf der Befehlszeile aufgerufen (prüfe das!): Der *Anzahl n* der Städte.
- Als erstes legen wir ein *Array von n Stadt-Strukturen* mit **`malloc`** an (da das Städte- und das Verbindungs-Array in Summe sehr groß sein können, sollten wir dafür *keine lokalen Arrays* verwenden).
  Prüfe das **`malloc`** auf Fehler!

  Wir *initialisieren* die Elemente auch gleich:
  - Als *x- und y-Koordinate* nehmen wir eine *Zufallszahl innerhalb der Fenstergröße* (lass rundherum *ein paar Pixel Rand*, damit beim Zeichnen der Städte nicht über den Rand gezeichnet wird, und sorge dafür, dass das Programm bei jedem Lauf *andere* Zufallszahlen ermittelt).
  - Der *Union-Find-Pointer* wird am Anfang auf **`NULL`** gesetzt.

  Ich empfehle, *für alle Schleifen* über das Städte- oder das Verbindungs-Array *Pointer statt* **`int`***'s als Schleifenvariable* zu verwenden, das macht den Code kürzer und leserlicher, wenn die Array-Elemente Strukturen sind!

  Zeig die Städte auch gleich *grafisch* an (mit einem kleinen Kreis oder Kreuzchen).
- Dann kommt das *Array der Verbindungs-Strukturen*, ebenfalls *dynamisch* angelegt.

  Unsere Verbindungen sind *richtungslos*, zwischen zwei Städten *A* und *B* gibt es daher *nicht zwei* Verbindungen (*A*-->*B* und *B*-->*A*), *sondern nur eine* (bei mir liegt deshalb die *erste Stadt* einer jeden Verbindung im Städte-Array *immer weiter hinten* als die zweite Stadt der Verbindung).
  Weiters speichern wir keine Verbindungen zu sich selbst.
  Für *n* Städte brauchen wir daher ein Array mit *n*(n-1)/2* Verbindungen (Warum?).

Auch die Verbindungen werden gleich *initialisiert*.
Dazu brauchst du *zwei geschachtelte Schleifen über das Städte-Array*:
Die äußere läuft *vom zweiten Array-Element bis zum Array-Ende*,
die innere durchläuft alle Elemente des Städte-Arrays *vor diesem Element*.
In der Schleife speichern wir im aktuellen Element des Verbindungs-Arrays
die beiden *Stadt-Pointer* (=Schleifenvariablen) und deren *Entfernung* (Hilfsfunktion),
dann wird die aktuelle Verbindungsarray-Position eins weitergezählt.

- Als nächstes kommt der **`qsort`**-Aufruf für das *Verbindungs-Array*.
- Dann kommt der eigentliche ***Kruskal-Algorithmus***:

  Er besteht aus einer *Schleife über das Verbindungs-Array*. Die Schleife *endet*,
  sobald *n-1 Verbindungen zur Lösung* genommen wurden (zähl mit!).

  In der Schleife holen wir uns *die beiden Städte der aktuellen Verbindung*
  und berechnen für beide ihre *"repräsentative Stadt"* (Hilfsfunktion).

  Sind die beiden repräsentativen Städte *verschieden* (siehe oben!),
  dann gehört die *Verbindung zur Lösung*:

  - Wir *zählen* sie.
  - Wir *zeichnen eine Linie zwischen den beiden Städten der Verbindung*
    (*warte* nach jedem Anzeigen ein paar Millisekunden,
    damit man der Lösung beim Wachsen zusehen kann).
  - Wir *speichern im Pointer-Member* der einen "repräsentativen Stadt"
    einen Pointer auf die andere "repräsentative Stadt" (wie oben beschrieben).
- *Summiere* in der Schleife auch die *Gesamtlänge* aller Verbindungen in der Lösung,
  gib die Summe nach der Schleife aus.
- Für Linux- und Mac-Benutzer: Das Programm verwendet Funktionen aus **`math.h`** .
  Beim Linken ist daher die Option **`-lm`** nötig.