Planung eines optimalen Telefonnetzes

Problemstellung

In einer gitterförmig aufgebauten Stadt ist eine Menge von Telefonknoten durch ihre ganzzahligen x-y-Koordinaten gegeben. Die Kosten für die Verbindung zweier Telefonknoten (x_1,y_1) und (x_2,y_2) wird mit Hilfe der sogenannten *Manhattan-Distanz* berechnet:

$$dist((x_1,y_1),(x_2,y_2)) = |x_1-x_2| + |y_1-y_2|$$

Die Abbildung 3.1 zeigt eine Stadt mit zwei blau gefärbten Telefonknoten. Die Knoten mit den Koordinaten (1,3) und (5,1) haben eine Manhattan-Distanz von dist = |1-5| + |3-1| = 6. Die Manhattan-Distanz drückt aus, dass Telefonleitungen nur längs von Straßen (horizontale und vertikale Linien) gelegt werden dürfen. Beispielsweise wäre die rote Linie der Länge 6 eine mögliche Verbindung zwischen den beiden Knoten.

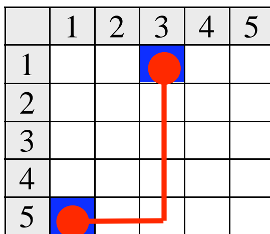

Abb. 3.1: Telefonnetz mit 2 Knoten

Zwei Knoten gelten als nicht direkt verbindbar, wenn ihre Manhattan-Distanz über einen Leitungsbegrenzungswert *lbg* liegt. Damit sind die Kosten *cost* für die Verbindung zweier Telefonknoten (x_1,y_1) und (x_2,y_2) wie folgt definiert:

$$cost(((x_1,y_1),(x_2,y_2)) = \begin{cases} dist((x_1,y_1),(x_2,y_2)), \text{ falls } dist((x_1,y_1),(x_2,y_2)) \leq lbg, \\ \infty, \text{ sonst} \end{cases}$$

Mit dem **Algorithmus von Kruskal** soll für eine Stadt mit einer gegebenen Menge von Telefonknoten ein optimales Telefonnetz, d.h. ein minimal aufspannender Baum, berechnet werden.

Beispielsweise ergibt sich für die Stadt mit 7 Knoten und lbg = 7 in Abbildung 3.2 einen minimal aufspannenden Baum (der Baum ist nicht eindeutig!) mit den Gesamtkosten von 3+3+5+3+2+2=18.

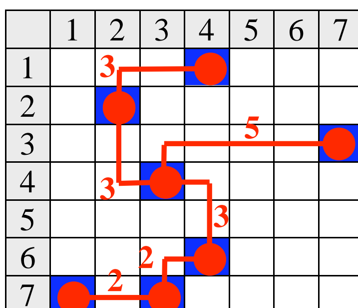

Abb. 3.2: Optimales Telefonnetz mit 7 Knoten

Zur Generierung hinreichend vieler und großer Testdatenmengen ist eine Funktion vorzusehen, mit der sich n zufällige Knoten in einem xMax*yMax großen Gitter erzeugen lassen.

Klasse TelNet

Die Klasse TelNet verwaltet die Menge aller Telefonknoten in einem STL-Vektor.

```
struct Knoten
{
      Knoten(int x, int y) : k_x(x), k_y(y) {}
      int k_x;
      int k_y;
};

typedef vector<Knoten> TelKnoten;
```

Prinzipiell ließe sich zum Speichern des Graphen einer der in Aufgabe 2 entwickelten Graphklassen verwenden. Da der Graph (in Abhängigkeit vom Leitungsbegrenzungswert) jedoch sehr dicht werden könnte, müsste eine speicheraufwendige Adjazenzmatrix oder -liste verwendet werden, die der Algorithmus aber gar nicht benötigt. Denn mit der Kostenfunktion *cost* gibt es eine einfache Möglichkeit, zu einem Knoten alle Nachbarknoten zu bestimmen.

Folgende Methoden sind in der Klasse vorzusehen:

- bool addTelKnoten(const Knoten& k);
  fügt einen neuen Telefonknoten k zum Netz dazu.
- bool remTelKnoten(const Knoten& k);
  löscht den Telefonknoten k aus dem Netz
- bool searchTelKnoten(const Knoten& k);
  prüft, ob ein Knoten k im Netz vorhanden ist.
- void generateTelKnoten (int n, int xMax, int yMax);
  erzeugt n zufällige Knoten aus einem xMax*yMax großen Gitter.
- const TelKnoten* getTelKnoten()
  Liefert alle Knoten des Netzes zurück.
- void setLbg(int lbg);
  Setzt den Leitungsbegrenzungswert auf lbg.
- int kosten(const Knoten& k1, const Knoten& k2);
  Liefert die Kosten der Verbindung von k1 nach k2 (siehe Funktion *cost*).
- const TelVerbindungen* berechneOptTelNet(int& c);
  berechnet einen minimal aufspannenden Baum nach dem Algorithmus von Kruskal und liefert diesen zurück. Falls kein aufspannender Baum existiert, wird 0 zurückgeliefert. Außerdem werden die Gesamtkosten des minimal aufspannenden Baums über den Referenzparameter c zurückgeliefert.

  TelVerbindungen kann als Liste von Kanten definiert werden, wobei eine Kante aus Anfangs- und Endknoten und ihre Kosten besteht:

```
struct Verbindung

      Knoten k1, k2;
      int c;
};

typedef list<Verbindung> TelVerbindungen;
```

Der Algorithmus von Kruskal verwendet zwei Hilfsdatenstrukturen:

- eine Union-Find-Struktur zur Speicherung einer Menge von aufspannenden Bäumen;
- eine Vorrangswarteschlange zur Speicherung aller Kanten mit schnellem Zugriff auf die billigste Kante.

Für beide Hilfsdatenstrukturen ist es geschickt, statt der Koordinaten der Knoten eine interne Nummerierung (1, 2, 3, …, n) zu verwenden. Da die Knoten in einem Vektor abgespeichert sind, kann für die interne Nummerierung einfach der Index verwendet werden.

Klasse Union-Find

Mit einem Objekt dieser Klasse lassen sich eine Menge von disjunkten Teilmengen $S_1, S_2, \ldots, S_k \subseteq \{1, 2, 3, \ldots, n\}$ verwalten. Folgende Methoden bietet die Klasse an.

- Konstruktor(int n)
  Initialisiert das Objekt mit den disjunkten Teilmengen {1}, {2}, …, {n}.
- int find(int e)
  liefert die Menge zurück, in der e Element ist.
- bool makeUnion(int e1, int e2)
  vereinigt die zwei Mengen, zu denen e1 bzw. e2 gehört, zu einer neuen Menge und liefert true zurück. Falls e1 und e2 zur gleichen Menge gehören, wird nicht vereinigt und false zurückgegeben.
- getNumberOfSets()
  liefert die Anzahl der Mengen zurück.

Vorrangswarteschlange

In der Vorrangswarteschlange werden alle gewichteten Kanten (Verbindungen mit Kosten) so verwaltet, dass auf die Kanten mit minimalen Gewicht effizient zugegriffen werden kann. Zwei Varianten bieten sich an:

- STL-Container priority_queue
- STL-Container vector, der nach dem Befüllen sortiert wird.

Bei beiden Varianten ist es geschickt, einen Typ für gewichtete Kanten mit internen Knoten-Nummern zu definieren und einen <-operator vorzusehen:

```
struct WEdge
{
      int i_u;
      int i_v;
      int w;
};

bool operator<(const WEdge& e1, const WEdge& e2)
```

Main

Folgendes Hauptprogramm definiert ein Telefonnetz wie in Abbildung 3.2 beschrieben, berechnet ein optimales Telefonnetz und gibt dieses aus.

```
#include <TelNet.h>

int main()
{
      TelNet meinTelNetz;

      meinTelNetz.addTelKnoten(Knoten(1,4));
      meinTelNetz.addTelKnoten(Knoten(2,2));
      meinTelNetz.addTelKnoten(Knoten(3,7));
      meinTelNetz.addTelKnoten(Knoten(4,3));
      meinTelNetz.addTelKnoten(Knoten(6,4));
      meinTelNetz.addTelKnoten(Knoten(7,1));
      meinTelNetz.addTelKnoten(Knoten(7,3));
      meinTelNetz.setLbg(7);

      int c;
      const TelVerbindungen* optTelNetz = meinTelNetz.berechneOptTelNet(c);

      if (optTelNetz == 0)
            cout << "Es existiert kein optimales Telefonnetz!" << endl;
      else
      {
            cout << "Optimales Telefonnetz:" << endl;
            cout << "Gesamt-Kosten: " << c << endl;
            cout << "Anzahl Verbindungen: " <<  optTelNetz->size() << endl;
            cout << "Verbindungen und Gewichte:" << endl;
            TelVerbindungen::const_iterator it;
            for (it = optTelNetz->begin(); it != optTelNetz->end(); it++)
            {
                  cout << it->k1.k_x << "," << it->k1.k_y << " -- ";
                  cout << it->k2.k_x << "," << it->k2.k_y;
                  cout << " Kosten: " << it->c << endl;
            }
      }

      return 0;
}
```

Testen

Testen Sie Ihr Programm für das Beispielnetz aus Abb. 3.2.

Generieren Sie dann unterschiedlich große Netze (z.B. 1000, 2000, 4000 Knoten) in einem 1000*1000 großen Gitter mit einem Leitungsbegrenzungswert lbg = 100. Ermitteln Sie jeweils die Kantenanzahl Ihres generierten Graphen und messen Sie die für die Berechnung eines optimalen Telefonnetzes benötigte CPU-Zeit. Bestätigt sich die theoretisch hergeleitete Laufzeitkomplexität?