1 Minimale Spannbäume ohne LEDA

Wir wollen einen Algorithmus zur Berechnung minimaler Spannbäume in ungerichteten Graphen mit ganzzahligen Kantengewichten ohne Verwendung von LEDA implementieren. Der Eingabegraph soll aus einer Textdatei eingelesen werden, deren Name dem Programm als Kommandozeilenargument übergeben wird. Das Gewicht eines minimalen Spannbaums soll über den Standardausgabe-Stream `cout` ausgegeben werden.

1.1 Einlesen einer Textdatei

Um in C++ eine Datei einzulesen, kann sehr komfortabel die in `fstream.h` definierte Klasse `ifstream` verwendet werden: bei der Deklaration wird einfach der Dateiname dem Konstruktor übergeben, und die in der Textdatei enthaltenen Zahlen können einfach mit dem `>>`-Operator gelesen werden.

```
int main(int argc, char **argv) {
   ...
   int n,m;
   ifstream I(argv[1]);
   I >> n >> m; // Knotenzahl n und Kantenzahl m einlesen
   ...
}
```

Bei größeren Dateien läßt sich die Einlesezeit unter Umständen verkürzen, indem man nicht jeden einzelnen Wert mit dem `>>`-Operator aus der Datei liest, sondern die Datei blockweise lädt und dann im Hauptspeicher in Werte zerlegt.

1.2 Algorithmus von Kruskal

Da die Implementierung von Priority-Queues mittels Fibonacci-Heaps ohne LEDA sehr aufwendig ist, bietet es sich an, den Algorithmus von Kruskal für die Implementierung ohne LEDA auszuwählen. Bei der Implementierungsvariante ohne Priority-Queue müssen lediglich am Anfang alle Kanten nach ihrem Gewicht sortiert werden. Dazu kann entweder die Funktion `qsort` aus `stdlib.h` verwendet werden oder eine eigene Implementierung eines effizienten Sortierverfahrens (Quicksort, Heapsort, Mergesort, etc.).

In der Hauptschleife des Algorithmus von Kruskal werden die Kanten dann der Reihe nach daraufhin geprüft, ob ihre Endpunkte schon in einem gemeinsamen Teilbaum enthalten sind. (Falls nein, werden die Teilbäume vereinigt und die Kante in den Spannbaum eingefügt.) Zu diesem Zweck muß eine sogenannte Union-Find-Struktur verwaltet werden, die in Abschnitt 2 erklärt wird.

2 Union-Find-Strukturen[1]

2.1 Grundlagen

Das "Union–Find–Problem" besteht darin, disjunkte Mengen unter den Operationen

- *makeset:* Schaffen einer neuen Menge, die nur das gegebene Element enthält
- *union:* Vereinigen zweier gegebener Mengen
- *find:* Bestimmung der Menge, die das gegebene Element enthält

zu verwalten. Zur Implementierung soll der Algorithmus in jeder der Mengen ein sogenanntes "*kanonisches Element*" unterhalten, das zur Identifikation der betreffenden Menge dient. Als ein solches Element kann ein beliebiges Datenelement verwendet werden, denn die Mengen sind als disjunkt vorausgesetzt, und somit ist ein Element für seine Menge stets charakteristisch.

Operationen:

makeset(x): Schaffe eine neue Menge, die nur das Element x enthält. Dieses wird natürlich als kanonisches Element verwendet. Es ist in keiner anderen Menge enthalten.

find(x): Gib das kanonische Element zu derjenigen Menge zurück, die x enthält.

union(x,y): Vereinige die Mengen mit den kanonischen Elementen x und y und zerstöre die beiden Mengen. Wähle das kanonische Element für die Resultatmenge neu und gib es zurück. Voraussetzung: $x \neq y$.

Implementierung: Jede Menge wird als In–Tree dargestellt, dessen Knoten die Elemente enthalten. Die Wurzel enthält das kanonische Element. Somit besitzt jeder Knoten x einen Zeiger zu seinem Vater $p(x)$; Der Zeiger der Wurzel zeigt auf die Wurzel selbst. Unter Berücksichtigung dieser Konzepte kann nun die Implementierung der Funktionen angegeben werden:

makeset(x): Definiere $p(x) := x$ und gib x zurück;

find(x): Folge, ausgehend vom Knoten mit Inhalt x, den Zeigern bis zur Wurzel des Baums und gib das kanonische Element zurück;

union(x,y): Definiere $p(x) := y$ und gib y zurück; Das heißt, der Baum mit Wurzel x wird an den mit Wurzel y angehängt, so daß y als ein weiteres Kind x erhält.

Bemerkung: Mit dieser Implementierung von *union* entartet ein Baum im worst case linear, so daß *find* die Komplexität $O(n)$ erhält, wenn n *makeset*–Operationen zum Aufbau des Baums verwendet wurden, der Baum also n Elemente enthält. Dies läßt sich durch Anwendung zweier Heuristiken, der *Pfadkompression* und dem Schema *Union–by–Rank* noch erheblich verbessern:

[1] Dieser Abschnitt ist dem Skript *Datenstrukturen* von Jens Ernst entnommen: http://wwwmayr.in.tum.de/skripten/ead_ws9798.ps

Pfadkompression: Die Struktur des Baums wird während der *find(x)*–Operation verändert. Dabei wird ausgenutzt, daß der Algorithmus beim Hochlaufen der In–Pointer in Richtung Wurzel nicht nur für x, sondern auch für alle anderen Elemente auf dem Weg von x zur Wurzel erfährt, welcher Menge sie angehören. Deshalb werden ihre Zeiger nach Bestimmung der Wurzel direkt auf das kanonische Element gerichtet.

Union–by–Rank: Hierfür wird eine Freiheit in der Spezifikation von *union* ausgenutzt, nämlich die Tatsache, daß es keine Rolle spielt, welcher der zu vereinigenden Bäume an welchen angehängt wird. Um die Komplexität von *find* zu reduzieren, wollen wir erreichen, daß die *union*–Operation die Bäume nicht unnötig hoch werden läßt. Dazu speichern wir zusammen mit jedem Knoten x eine ganze Zahl *Rang(x)*, die eine obere Schranke für die *Höhe* von x angibt. Es sei bereits jetzt darauf hingewiesen, daß die Aktualisierung dieser Werte außerordentlich wenig Aufwand bereitet: Nur solange x die Wurzel, also kanonisches Element ist, kann sich an seiner Höhe überhaupt etwas ändern. (Ganz anders verhält es sich mit der Tiefe der einzelnen Knoten.)
Bei *makeset(x)* wird *Rang(x)* mit 0 vorbelegt, bei *union(x,y)* werden *Rang(x)* und *Rang(y)* verglichen. Falls *Rang(x)<Rang(y)*, so wird $p(x) := y$ gesetzt und y zurückgegeben. Das bedeutet, daß grundsätzlich der weniger hohe Baum an den höheren angehängt wird. Falls aber *Rang(x)=Rang(y)*, so wird $p(x) := y$ gesetzt und danach *Rang(y)* um 1 erhöht. Die Erhöhung des Rangs ist bei den anderen Fällen nicht nötig.

Algorithmen:

makeset(x):

```
p(x):=x;
Rang(x):=0;
```

find(x): (mit Pfadkompression)

```
if (x!=p(x)) then p(x):=find(p(x)) fi;
return p(x);
```

union(x,y): (Union–by–Rank)

```
if (Rang(x)>Rang(y)) then
   p(y):=x; return x;
elsif (Rang(x)<Rang(y)) then
   p(x):=y; return y;
elsif (Rang(x)=Rang(y)) then
   p(x):=y; Rang(y)++; return y;
fi
```

Durch Union–by–Rank und Pfadkompression erreichen wir die folgende amortisierte Komplexität: Die Laufzeit für $m \geq n$ *union*– und *find*–Operationen auf einer Grundmenge von n Elementen ist $O(m\alpha(m,n))$, wobei α im wesentlichen eine Inverse der Ackermann-Funktion ist und extrem langsam wächst.

Tip: In C++ ist `union` ein reserviertes Wort, man muß die union-Funktion also anders nennen!

3 Minimale Spannbäume mit Nebenbedingung

Nun wollen wir das folgende erweiterte Spannbaum-Problem für einen gegebenen ungerichteten Graphen $G = (V, E)$ mit Kantengewichten $c : E \to \mathbb{N}$ lösen: für jede Kante $e \in E$ soll das Gewicht eines Spannbaums für G berechnet werden, der die Kante e enthält und unter allen solchen Spannbäumen minimales Gewicht hat.

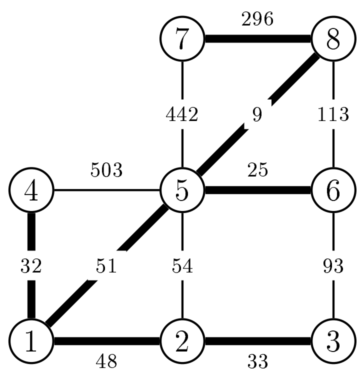
Eine Methode, die auf jeden Fall funktionieren würde, ist die folgende: für jede Kante e lassen wir den Algorithmus von Kruskal einmal durchlaufen, wobei wir nur als allererste Kante statt der billigsten Kante des Graphen die Kante e in den Spannbaum einfügen. Dadurch bekämen wir die gewünschten minimalen Spannbäume mit Nebenbedingungen, die Laufzeit wäre aber mindestens quadratisch, d.h. $\Omega(m^2)$, da wir m-mal alle Kanten durchgehen müßten.

Die folgenden Tips sollten helfen, eine effizientere Methode zu implementieren:

- Sei T ein minimaler Spannbaum und $w(T)$ sein Gewicht. Dann ist für alle Kanten e von T bereits klar, daß das minimale Gewicht eines e enthaltenden Spannbaums $w(T)$ ist; für diese Kanten muß also nichts mehr zusätzlich berechnet werden.

- Sei e nun eine Kante des Graphen, die nicht in T enthalten ist. Durch Einfügen von e in T entsteht ein Kreis. Sollte es nicht möglich sein, nur durch Betrachten der Kanten auf diesem Kreis herauszufinden, wie ein minimaler e enthaltender Spannbaum aussieht?

Beispiel:



Minimaler Spannbaum: Gewicht 494

Kante	Spannbaum
48	494
32	494
51	494
33	494
54	497
93	536
503	946
25	494
442	640
9	494
113	582
296	494