RWTH Aachen **SoSe 2007**
Lehrgebiet Theoretische Informatik Blatt 12
Kneis—Mölle—Richter—Rossmanith—Thiemann 2.7.2007

Übungen zur Vorlesung Datenstrukturen und Algorithmen

T32

Beweisen Sie die Korrektheit des Algorithmus von Prim. Es soll angenommen werden, daß der Eingabegraph zusammenhängend ist und seine Kantengewichte durch die Funktion *weight* kodiert sind.

```
procedure Prim(s):
Q ← new Priority−Queue();
for v in V do Q.insert(v) with key ∞; parent[v] ← v od;
Q.decrease_key(s) to 0;
while Q ≠ ∅ do
  v ← Q.extract_min;
  forall u ∈ Q adjacent tov do
    if weight(v,u)<Q.key(u) then Q.decrease_key(u) toweight(v,u); parent[u] ← v fi
  od
od
```

Lösungsvorschlag:
Wir betrachten die Zeitpunkte $i \in \{1, \dots, n\}$ nach dem i-ten Durchlauf der Zeile

```
v:=Q.extract_min;
```

Ferner bezeichne T_i den zum Zeitpunkt i erwägten Spannbaum, der im obigen Algorithmus durch die *parent*-Zeiger der Knoten aus der jeweiligen Menge $V \setminus Q$ beschrieben wird. Insbesondere gilt somit $T_1 = (\{s\}, \emptyset)$.
Wir beweisen die folgenden zwei Invarianten via Induktion:

1. T_i ist ein minimaler Spannbaum für die Knoten in $V \setminus Q$.
2. T_i ist Teil eines minimalen Spannbaums für den ganzen Graphen.

Für $i = 1$ sind beide Aussagen offenkundig wahr. Für $i = n$ impliziert die erste Invariante die Korrektheit des Algorithmus.
Es sei nun $2 \le i \le n$. Wir nehmen an, daß die beiden Invarianten für $i - 1$ gelten. Nach Voraussetzung ist also T_{i-1} ein minimaler Spannbaum für die Knoten, die im letzten Schritt in $V \setminus Q$ lagen. Ferner existiert nach Voraussetzung ein minimaler Spannbaum T^* für den Gesamtgraphen, der ein Oberbaum von T_{i-1} ist.
Sei F_i der Teilgraph von $T*$, der nicht T_i enthält, F_i ist dann ein Wald.
Weil T^* ein Baum ist, führt in T^* jeweils genau eine Kante $e = \{v, u\}$ aus dem Teilbaum T_{i-1} in jede Komponente von F_{i-1}. Wird nun einer der entsprechenden Knoten u aus der Priority-Queue entfernt und es gilt $parent[u] = v$, so gelten offensichtlich beide Invarianten, da mit $e = \{v, u\}$ eine Kante von T^* ausgewählt wurde.

Wird hingegen keine der oben genannten Kanten, sondern eine andere Kante e' gewählt, so müssen wir das folgende Austauschargument für T^* anwenden:

Sowohl e' als auch eine der oben genannten Kanten e führen in dieselbe Komponente von F_{i-1}. Es sei $T^* = (V, E^*)$. Der Graph $(V, E^* \cup \{e'\})$ besteht aus dem Baum T_{i-1} und dem Wald F_{i-1} sowie den beiden Kanten e und e', von denen jede den Baum mit der entsprechenden Komponente verbindet. Aufgrund der minimalen Kantenwahl kann e' kein höheres Gewicht als e haben. Andererseits kann e' auch kein kleineres Gewicht als e haben, weil $T^{**} = (V, (E^* \cup \{e'\}) \setminus \{e\})$ sonst ein billigerer Spannbaum als T^* wäre (Widerspruch zur Minimalität). Somit können wir T^* durch T^{**} ersetzen und erhalten wieder die Gültigkeit der beiden Invarianten.

T33

Geben Sie eine Folge von Union- und Find-Operationen an, die zu einem Baum der Höhe vier führt. Verwenden Sie dabei sowohl die Rangheuristik als auch Pfadkompression.

Lösungsvorschlag:

Wir wollen hier annehmen, daß $union(a, b)$ so implementiert ist, daß die Wurzel von b unter die Wurzel von a gehängt wird.

Nur Unionoperationen: $(1, 2)$, $(3, 4)$, $(5, 6)$, $(7, 8)$, $(1, 3)$, $(5, 7)$, $(1, 5)$

H27 (10 Punkte)

Zeigen oder widerlegen Sie die Korrektheit des folgenden Greedy-Algorithmus zum Finden von minimalen Spannbäumen auf Graphen:

Beginne mit n isolierten Zusammenhangskomponenten, die jeweils einen der Knoten des Eingabegraphen enthalten. Solange es noch zwei Komponenten A, B gibt, verbinde diese mit einer Kante minimalen Gewichts unter allen Kanten zwischen A und B.

Lösungsvorschlag:

Dieser Algorithmus ist inkorrekt, wie das folgende Gegenbeispiel zeigt. Der Graph bestehe aus den Knoten A, B, C, die ein Dreieck bilden. Die Kante $\{A, B\}$ koste 100 Euro, die anderen jeweils 1 Cent. Dem Algorithmus steht es nun frei, die 100-Euro-Kante zu wählen, wenn er die Komponenten $\{A\}$ und $\{B\}$ zu verbinden wünscht.

H28 (10 Punkte)

Modifizieren Sie die Klasse *Partition* aus der Vorlesung so, daß zusätzlich die Technik *union-by-rank* (also die Rangheuristik) verwendet wird.

```
public class Partition {
  int[] s;
  public Partition(int n) {
    s = new int[n];
    for(int i=0; i<n; i++) s[i]=i;
  }
  public int find(int i) {
    int p=i, t;
    while(s[p] ≠ p) p=s[p];
    while(i ≠ p) { t=s[i]; s[i]=p; i=t; }
    return p;
  }
  public void union(int i, int j) { s[find(i)] = find(j); }
}
```

Lösungsvorschlag:

Ungetestet, aber vermutlich korrekt:

```
public class Partition {
  int[] s;
  int[] r;
  public Partition(int n) {
    s = new int[n];
    r = new int[n];
    for(int i=0; i<n; i++) { s[i]=i; r[i]=0; }
  }
  public int find(int i) {
    int p=i, t;
    while(s[p] ≠ p) p=s[p];
    while(i ≠ p) { t=s[i]; s[i]=p; i=t; }
    return p;
```

```
  }
  public void union(int i, int j) {
    int iroot, jroot;
    iroot = find(i); jroot = find(j);
    if (r[iroot]>r[jroot]) { s[jroot] = iroot; return; }
    if (r[iroot]<r[jroot]) { s[iroot] = jroot; return; }
    r[iroot]++; s[jroot] = iroot;
  }
}
```