Algorithmen & Datenstrukturen

Herbst 2017

Vorlesung 14

```
MST-Kruskal(G)
    sortiere Kanten nach Gewicht: c(e_1) ≤ ... ≤ c(e_m)
    E' = ∅
    for k = 1..m
        if (V, E' ∪ {e_k}) ist kreisfrei // wenn nicht, verwirf Kante
            E' = E' ∪ {e_k}              // wähle Kante
    return (V, E')
```

Laufzeitanalyse

Wie prüfen wir Kreisfreiheit in jedem Schritt?

DFS/BFS in jedem Schritt: zu teuer

In jedem Schritt:

- haben wir einen Wald 
- neue Kante gibt keinen Kreis
  $\Leftrightarrow$ Anfangs-/Endknoten in verschiedenen Bäumen
- Kante einfügen: verschmilzt 2 Bäume

Beobachtung: Wald gibt Partition von V

Ziel: Designe ADT und Datenstruktur
für diese Operationen

Union-Find ADT für Partitionen

Menge $M = \{1, \dots, n\}$, Partition: $M = M_1 \cup \dots \cup M_k$
disjunkt, alle $M_i \neq \emptyset$

Beispiel: $n = 8$, $M_1 = \{1, 3, 5, 6\}$, $M_2 = \{2\}$, $M_3 = \{4, 7, 8\}$

Operationen:
- MakeSet(i): liefert neue Menge mit Element i und Namen i
- Find(x): liefert Namen der Menge die x enthält
- Union(i,j): vereinigt Mengen i und j; Resultat hat Namen j

Name einer Menge ist ein beliebiges Element

Im Beispiel oben könnte $M_1 = 3$, $M_2 = 2$, $M_3 = 7$ sein

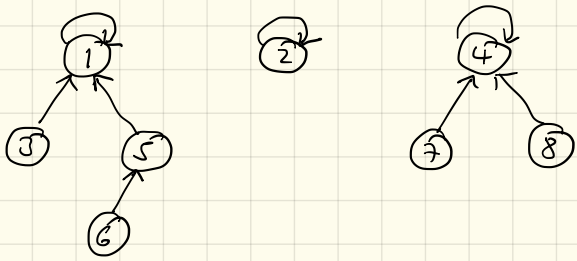
Mit diesem ADT wird MST-Kruskal:

```
MST-Kruskal(G)
    sortiere Kanten nach Gewicht: c(e_1) ≤ ... ≤ c(e_m)
    E' = ∅
    for v ∈ V: MakeSet(v)        // Initialer Wald / Partition
    for k = 1..m
        (u,v) = e_k
        if Find(u) ≠ Find(v)     // kein Kreis?
            Union(Find(u), Find(v))   // Verschmelze Bäume
            E' = E' ∪ {e_k}
    return (V, E')
```

Datenstruktur für Union-Find

Idee: Baum für jede Menge in Partition
- damit Find schnell wird
- Baum muss nicht binär sein

Beispiel:

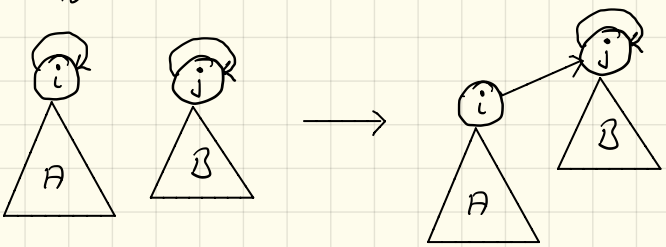


Knoten = Elemente der Menge
Wurzel = Name der Menge

MakeSet(i): 

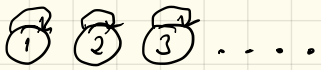
Find(x): Wurzel des Baums der x enthält

Union(i,j):



Kein Problem da Baum nicht binär sein muss

Laufzeit für Find? Höhe des Baums!

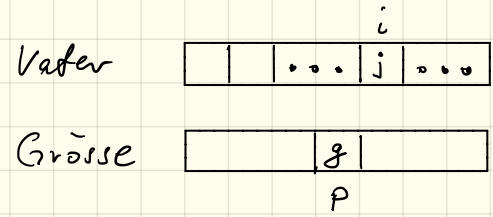
Kann entarten. z.B. 

....

mit Union(1,2), Union(2,3), ...

Idee: Bei Union(i,j) hänge kleineren Baum unter größeren
→ braucht Größeninformation (Größe = Anzahl Elemente)

Datenstruktur: 2 Arrays



Vater[i] ist Vater von i

Menge p hat Größe g

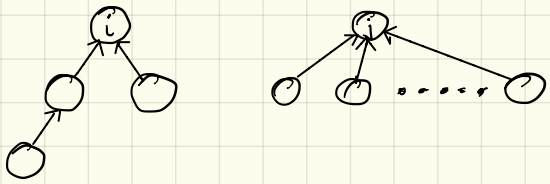
MakeSet(i): Vater[i] = i, Größe[i] = 1

Find(x): while Vater[x] ≠ x
x = Vater[x]
return x

Union(i,j): if Größe[i] > Größe[j]
vertausche i und j
Vater[i] = j
Größe[j] = Größe[i] + Größe[j]

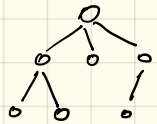
Warum Größe und nicht Höhe?

Mache Dir an diesem Beispiel klar: Union(i,j) mit



4 Knoten k Knoten

Satz: Jeder so erzeugte Baum der Höhe h hat mindestens 2^h Elemente
($\Rightarrow$ Find ist $O(\log n)$)



Höhe 2
7 Elemente

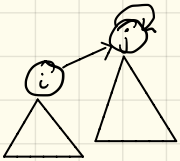
Beweis: MakeSet(i): Höhe 0
1 Element ✓

Zu zeigen: Union erhält die Eigenschaft

Annahme i: hat Höhe h_1 und $g(i) \geq 2^{h_1}$ Elemente
j: " h_2 und $g(j) \geq 2^{h_2}$ "

Ohne Einschränkung $g(i) \leq g(j)$

Union (i,j):



Höhe $h = \max(h_1+1, h_2)$
Größe: $g(i) + g(j)$

Fall 1: $h = h_2$

$$g(i) + g(j) \geq g(j) \overset{\text{Annahme}}{\geq} 2^{h_2} = 2^h$$ ✓

Fall 2: $h = h_1 + 1$

$$g(i) + g(j) \geq 2\,g(i) \overset{\text{Annahme}}{\geq} 2 \cdot 2^{h_1} = 2^h$$ ■

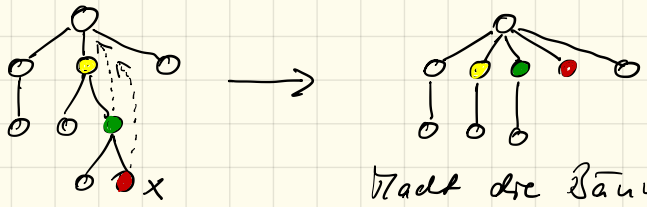
Also:

Union (i,j)	$O(1)$
Find (x)	$O(\log n)$

Die Höhe kann noch weiter verringert werden, wenn man nach jedem Find alle besuchten Knoten an die Wurzel hängt (macht Find asymptotisch nicht teurer): **Pfadverkürzung**

```
Find(x)
    y = x                          // merke x in y
    while Vater[x] ≠ x
        x = Vater[x]
    while Vater[y] ≠ y             // jetzt ist x Wurzel, 2. Durchlauf
        z = Vater[y]
        Vater[y] = x               // hänge an Wurzel
        y = z
    return x
```

Beispiel:



Macht die Bäume sehr schnell flach

Durchschnittliche Kosten von m Operationen (Find, Union) auf n Elementen ist $O(\alpha(m,n))$

α: inverse Ackermann-Funktion

α wächst <u>sehr</u> langsam, $\alpha(m,n) \leq 5$ für alle praktischen Größen

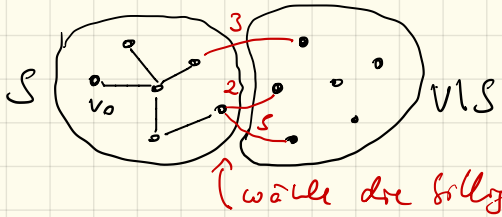
Laufzeit MST-Kruskal:

Sortieren	$O(m \log m)$
Rest	$O(m \alpha(m,n))$
Gesamt	$O(m \log m) = O(m \log n)$

da Graph zusammenhängend

MST-Algorithmus von Jarník/Prim/Dijkstra (1930, 57, 59)

Idee: lasse einen Spannbaum wachsen von einem $v_0 \in V$



verwendet nur Auswahlregel

(wähle die billigste)

$S = \{v_0\}$, $E = \emptyset$
while $|S| < n$
  wähle billigste (u,v) mit $u \in S$, $v \notin S$ (oder umgekehrt)
  $S = S \cup (\{u,v\} \cap V \setminus S)$
  $E' = E' \cup \{(u,v)\}$
return (V, E')

Ist korrekt da Auswahlregel für $S = V$.

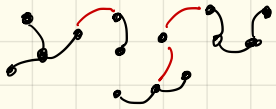
Verbesserungen analog zu Dijkstra kürzester Weg

$\Rightarrow O(m + n \log n)$ (mit Fibonacci Heaps)

MST-Algorithmus von Boruvka (1926)

Idee: In jedem Schritt wähle zu jedem Baum im Wald die billigste Kante die ihn mit einem anderen verbindet

neu gewählt (mit Auswahlregel)



6 Bäume ⇒ zwischen 6/2 und 6-1 Kanten werden gewählt

⇒ 3 mindestens halbiert sich in jedem Schritt

$E' = \emptyset$, $T = (V, E')$ // Anfangswald

repeat

  berechne Zusammenhangskomponenten von T // BFS

  für jede Komponente berechne billigste Kante zu einer anderen Komponente (wenn Auswahl nimm die mit kleinstem inzidenten Knoten) // damit // kein Kreis

  füge diese zu E'

until T hat eine Komponente

return T

Laufzeit: BFS $O(n+m) = O(n)$ (da Graph zusammenhängend)

billigste Kanten $O(m)$

⇒ $O(m \log n)$ (da $\leq \log_2 n$ Iterationen)

Geht auch mit Union-Find

Geeignet für parallele Implementierung