Übersicht

Greedy Algorithmen – Münzen wechseln

Es gibt diese acht Euromünzen:

Was ist die minimale Zahl von Münzen um 3.34 Euro zu zahlen?

Antwort:
Wir brauchen sechs Münzen:

Es ist unmöglich, weniger Münzen zu verwenden.

Greedy Algorithmen – Münzen wechseln

Es gibt diese acht Euromünzen:

Was ist die minimale Zahl von Münzen um 3.34 Euro zu zahlen?

Antwort:
Wir brauchen sechs Münzen:

Es ist unmöglich, weniger Münzen zu verwenden.

Münzwechsel – Ein Greedy-Algorithmus

Wir wollen den Betrag n wechseln:

Algorithmus

r := n;
while r > 0 **do**
  Choose biggest coin c with value(c) ≤ r;
  S := S ∪ { c};
  r := r − value(c)
od;
return S

Korrektheit

Lemma A

Sei C eine Münze und v ein Betrag, der mindestens so groß ist wie der Wert von C. Dann ist es suboptimal, v mit Münzen kleiner als C auszudrücken.

Beweise das Lemma für jede Münze von der kleinsten bis zur größten.
Nimm als Beispiel.
Sei v mindestens 1 EUR. Nehmen wir an, v kann mit genau k und kleineren Münzen für die verbleibenden $v - 50k$ Cents optimal bezahlt werden.
Da diese $v - 50k$ Cents optimal ausgezahlt werden, muß $v - 50k < 50$ und somit auch $100 \leq v < 50(k+1)$ gelten. Es folgt $k \geq 2$, ein Widerspruch zur Optimalität.

Korrektheit

Lemma A

Sei C eine Münze und v ein Betrag, der mindestens so groß ist wie der Wert von C. Dann ist es suboptimal, v mit Münzen kleiner als C auszudrücken.

Beweise das Lemma für jede Münze von der kleinsten bis zur größten.

Nimm  als Beispiel.

Sei v mindestens 1 EUR. Nehmen wir an, v kann mit genau k  und kleineren Münzen für die verbleibenden $v - 50k$ Cents optimal bezahlt werden.

Da diese $v - 50k$ Cents optimal ausgezahlt werden, muß $v - 50k < 50$ und somit auch $100 \leq v < 50(k+1)$ gelten. Es folgt $k \geq 2$, ein Widerspruch zur Optimalität.

Korrektheit

Theorem

Der Greedy-Algorithmus für den Münzwechsel ist optimal.

Beweis.

Nimm an, $C_1, C_2, \ldots, C_n$ ist eine Greedy-Lösung (mit $C_i \geq C_{i+1}$).
Zeige mit Induktion über k, daß eine optimale Lösung mit $C_1, C_2, \ldots, C_k$ beginnt.

Falls dem nicht so wäre, gäbe es eine optimale Lösung $C_1, C_2, \ldots, C_{k-1}, C'_k, \ldots, C'_m$ wobei $C_k > C'_i$ für $i = k, \ldots, m$.
Da $C'_k + \cdots + C'_m \geq C_k$ ist dies ein Widerspruch zu Lemma A. □

Korrektheit

Theorem

Der Greedy-Algorithmus für den Münzwechsel ist optimal.

Beweis.

Nimm an, $C_1, C_2, \ldots, C_n$ ist eine Greedy-Lösung (mit $C_i \geq C_{i+1}$).
Zeige mit Induktion über k, daß eine optimale Lösung mit $C_1, C_2, \ldots, C_k$ beginnt.

Falls dem nicht so wäre, gäbe es eine optimale Lösung $C_1, C_2, \ldots, C_{k-1}, C'_k, \ldots, C'_m$ wobei $C_k > C'_i$ für $i = k, \ldots, m$.

Da $C'_k + \cdots + C'_m \geq C_k$ ist dies ein Widerspruch zu Lemma A. □

Korrektheit

Theorem

Der Greedy-Algorithmus für den Münzwechsel ist optimal.

Beweis.

Nimm an, $C_1, C_2, \ldots, C_n$ ist eine Greedy-Lösung (mit $C_i \geq C_{i+1}$).
Zeige mit Induktion über k, daß eine optimale Lösung mit $C_1, C_2, \ldots, C_k$ beginnt.

Falls dem nicht so wäre, gäbe es eine optimale Lösung $C_1, C_2, \ldots, C_{k-1}, C'_k, \ldots, C'_m$ wobei $C_k > C'_i$ für $i = k, \ldots, m$.
Da $C'_k + \cdots + C'_m \geq C_k$ ist dies ein Widerspruch zu Lemma A. □

Briefmarkensammeln

Funktioniert der Greedy-Algorithmus auch für Briefmarken aus Manchukuo?

Welche Briefmarken für einen 20 fen–Brief?

Der Greedy-Algorithmus führt nicht zu einer optimalen Lösung und findet manchmal gar keine!

Briefmarkensammeln

Funktioniert der Greedy-Algorithmus auch für Briefmarken aus Manchukuo?

Welche Briefmarken für einen 20 fen–Brief?

Der Greedy-Algorithmus führt nicht zu einer optimalen Lösung und findet manchmal gar keine!

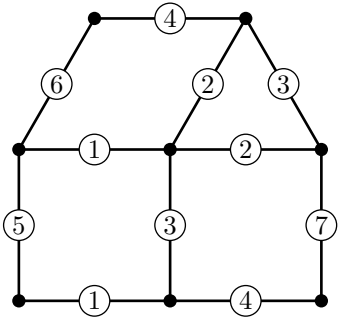
Briefmarkensammeln

Funktioniert der Greedy-Algorithmus auch für Briefmarken aus Manchukuo?

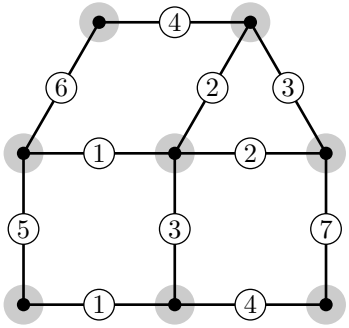
Welche Briefmarken für einen 20 fen–Brief?

Der Greedy-Algorithmus führt nicht zu einer optimalen Lösung und findet manchmal gar keine!

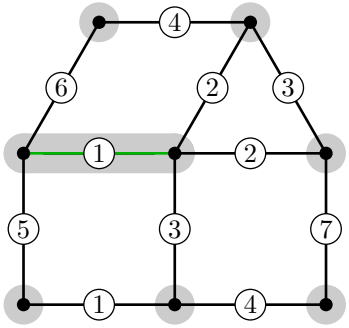
Kruskal: Minimaler Spannbaum



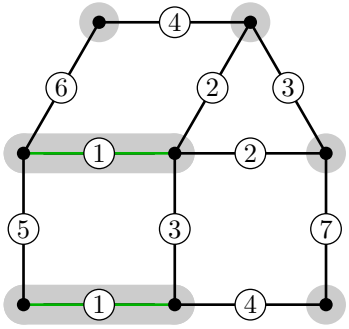
Kruskal: Minimaler Spannbaum



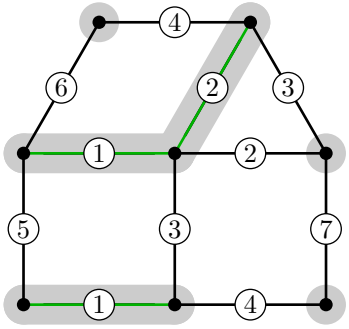
Kruskal: Minimaler Spannbaum



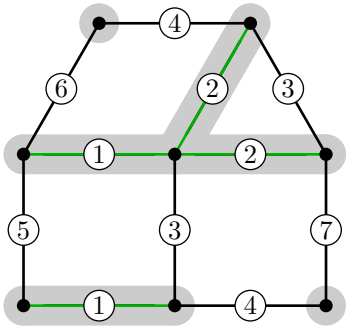
Kruskal: Minimaler Spannbaum



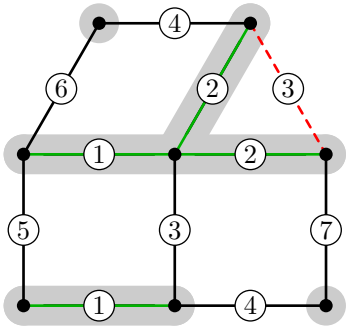
Kruskal: Minimaler Spannbaum



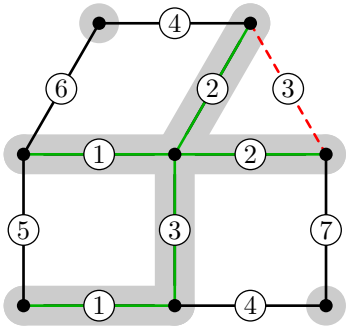
Kruskal: Minimaler Spannbaum



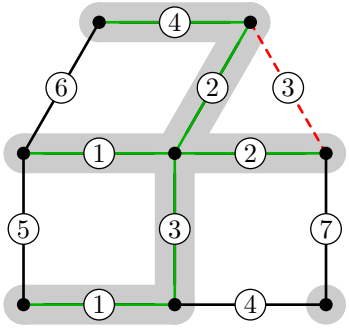
Kruskal: Minimaler Spannbaum



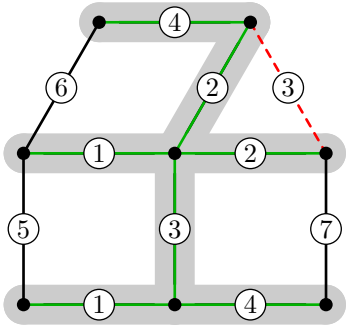
Kruskal: Minimaler Spannbaum



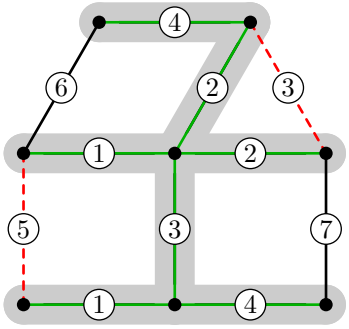
Kruskal: Minimaler Spannbaum



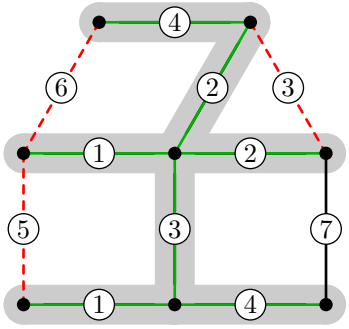
Kruskal: Minimaler Spannbaum



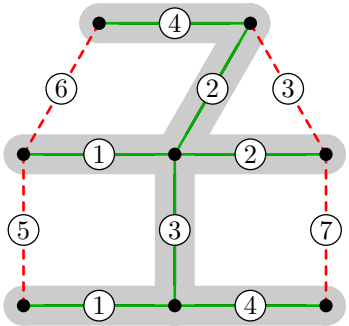
Kruskal: Minimaler Spannbaum



Kruskal: Minimaler Spannbaum



Kruskal: Minimaler Spannbaum



Kruskals Algorithmus – Implementierung

Algorithmus

function Kruskal(G, w) :
A := ∅;
for each vertex v in V[G] **do**
  Make_Set(v)
od;
sort the edges of E into nondecreasing order by weight;
for each edge { u, v}in E, nondecreasingly **do**
  if Find_Set(u) ≠ Find_Set(v) **then**
    A := A ∪ {{ u, v}};
    Union(u, v)
  fi
od;
return A

Korrektheit und Laufzeit der Implementierung

Die Korrektheit folgt als Korollar aus der Korrektheit des allgemeinen Greedy-Algorithmus und der Beobachtung zum graphischen Matroid.

Laufzeit mit $m = |E|$ und $n = |V|$ und $m \geq n - 1$:
n Make-Set-Operationen,
$O(m \log m)$ Zeit für das Sortieren der Kanten, und
$O(m)$ Find-Set- und Union-Operationen.

Mit einer geeigneten Union-Find-Implementierung zusammen $O(m \log m)$.

Korrektheit und Laufzeit der Implementierung

Die Korrektheit folgt als Korollar aus der Korrektheit des allgemeinen Greedy-Algorithmus und der Beobachtung zum graphischen Matroid.
Laufzeit mit $m = |E|$ und $n = |V|$ und $m \geq n - 1$:
n Make-Set-Operationen,
$O(m \log m)$ Zeit für das Sortieren der Kanten, und
$O(m)$ Find-Set- und Union-Operationen.

Mit einer geeigneten Union-Find-Implementierung zusammen $O(m \log m)$.

Union-Find: naiv

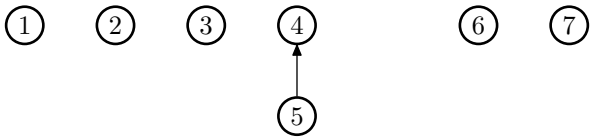
union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$

Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

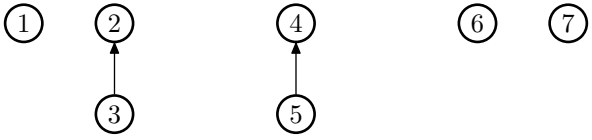
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$



Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$



Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$

Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$

Union-Find: union by rank

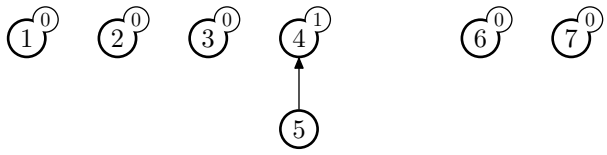
union(a, b): Verwende die ranghöhere Wurzel

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$

Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

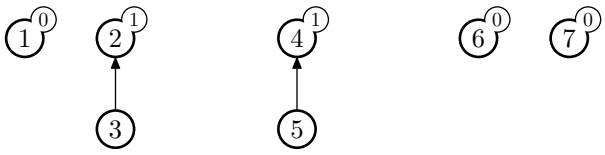
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

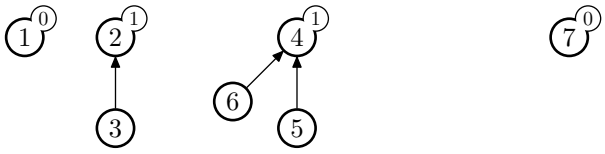
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

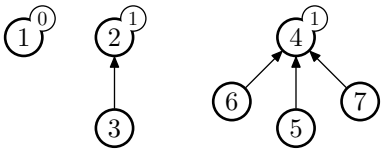
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

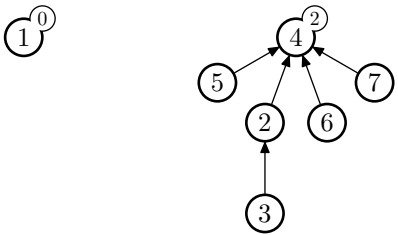
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

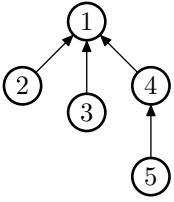
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: Pfadkompression

find(*a*): Komprimiere durchlaufene Pfade

Beispiel: *find*(4)



Union-Find

Algorithmus

procedure Make_Set(x) :
p[x] := x;
rank[x] := 0

- Wir betrachten jeweils eine Menge $\{0, \ldots, n-1\}$
- Ein Array für die Eltern eines für den Rang
- Ein Baum pro Menge repräsentiert durch die Wurzel
- Wurzel hier kurzgeschlossen

Union-Find

Algorithmus

function Find_Set(x) :
if $x \neq p[x]$ **then** $p[x] := \text{Find\_Set}(p[x])$ **fi**;
return p[x];

Algorithmus

procedure Union(x, y) :
$x := \text{Find\_Set}(x)$;
$y := \text{Find\_Set}(y)$;
if $\text{rank}[x] > \text{rank}[y]$ **then** $p[y] := x$
else $p[x] := y$;
if $\text{rank}[x] = \text{rank}[y]$ **then** rank[y]++**fi**;
fi;

Java

```
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); }
}
```

Union-Find – Analyse

Zunächst keine Pfadkompression.

Lemma

Falls eine Union-Find-Datenstruktur einen Baum mit m Elementen enthält, dann ist seine Höhe höchstens $\log m + 1$.

Beweis.

Wird ein Element neu hinzugefügt, dann ist die Höhe des Baums $1 = \log(1) + 1$.
Werden zwei Bäume zu einem kombiniert, dann ist die Höhe anschließend unverändert, außer die Höhen beider Bäume waren exakt gleich h.
Falls die Bäume vorher k und m Elemente enthielten, galt $h \leq \log(k) + 1 \leq \log(m) + 1$.
Daher $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. □

Union-Find – Analyse

Zunächst keine Pfadkompression.

Lemma

Falls eine Union-Find-Datenstruktur einen Baum mit m Elementen enthält, dann ist seine Höhe höchstens $\log m + 1$.

Beweis.

Wird ein Element neu hinzugefügt, dann ist die Höhe des Baums $1 = \log(1) + 1$.
Werden zwei Bäume zu einem kombiniert, dann ist die Höhe anschließend unverändert, außer die Höhen beider Bäume waren exakt gleich h.
Falls die Bäume vorher k und m Elemente enthielten, galt $h \leq \log(k) + 1 \leq \log(m) + 1$.
Daher $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. □

Union-Find – Analyse

Zunächst keine Pfadkompression.

Lemma

Falls eine Union-Find-Datenstruktur einen Baum mit m Elementen enthält, dann ist seine Höhe höchstens $\log m + 1$.

Beweis.

Wird ein Element neu hinzugefügt, dann ist die Höhe des Baums $1 = \log(1) + 1$.
Werden zwei Bäume zu einem kombiniert, dann ist die Höhe anschließend unverändert, außer die Höhen beider Bäume waren exakt gleich h.
Falls die Bäume vorher k und m Elemente enthielten, galt $h \leq \log(k) + 1 \leq \log(m) + 1$.
Daher $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. □

Union-Find – Analyse

Zunächst keine Pfadkompression.

Lemma

Falls eine Union-Find-Datenstruktur einen Baum mit m Elementen enthält, dann ist seine Höhe höchstens $\log m + 1$.

Beweis.

Wird ein Element neu hinzugefügt, dann ist die Höhe des Baums $1 = \log(1) + 1$.
Werden zwei Bäume zu einem kombiniert, dann ist die Höhe anschließend unverändert, außer die Höhen beider Bäume waren exakt gleich h.
Falls die Bäume vorher k und m Elemente enthielten, galt $h \leq \log(k) + 1 \leq \log(m) + 1$.
Daher $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. □

Union-Find – Analyse

Zunächst keine Pfadkompression.

Lemma

Falls eine Union-Find-Datenstruktur einen Baum mit m Elementen enthält, dann ist seine Höhe höchstens $\log m + 1$.

Beweis.

Wird ein Element neu hinzugefügt, dann ist die Höhe des Baums $1 = \log(1) + 1$.
Werden zwei Bäume zu einem kombiniert, dann ist die Höhe anschließend unverändert, außer die Höhen beider Bäume waren exakt gleich h.
Falls die Bäume vorher k und m Elemente enthielten, galt $h \leq \log(k) + 1 \leq \log(m) + 1$.
Daher $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. □

Union-Find – Analyse

Theorem

In einer anfangs leeren Union-Find–Datenstruktur mit Rangheuristik werden m Operationen in $O(m \log m)$ Zeit ausgeführt.

Beweis.

- Es gibt stets höchstens m Elemente
- Die Höhe aller Bäume ist durch $\log(m) + 1$ beschränkt
- Union und Find benötigt also nur $O(\log m)$ Zeit

□

Union-Find – Analyse

Theorem

In einer anfangs leeren Union-Find–Datenstruktur mit Rangheuristik werden m Operationen in $O(m \log m)$ Zeit ausgeführt.

Beweis.

- Es gibt stets höchstens m Elemente
- Die Höhe aller Bäume ist durch $\log(m) + 1$ beschränkt
- Union und Find benötigt also nur $O(\log m)$ Zeit

□

Rang und Pfadkompression

Mittels amortisierter Analyse (Tarjan 1975): m Operationen in $O(m\alpha(m))$ mit $\alpha(m)$ funktionale Inverse der Ackermannfunktion

Tarjan 1979, Fredman, Saks 1989: Das ist optimal!

Beweis recht kompliziert...