Union-Find: naiv

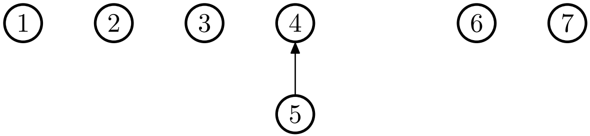
union(a, b): Hänge *find*(a) bei *find*(b) ein

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

Union-Find: naiv

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

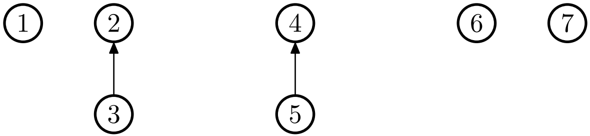
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$...



Union-Find: naiv

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

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



Union-Find: naiv

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

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

Union-Find: naiv

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

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

Union-Find: union by rank

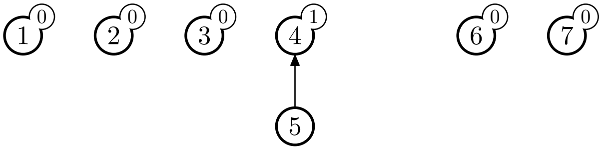
union(a, b): Verwende die ranghöhere Wurzel

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

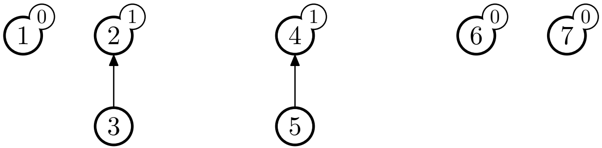
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

$union(a, b)$: Verwende die ranghöhere Wurzel

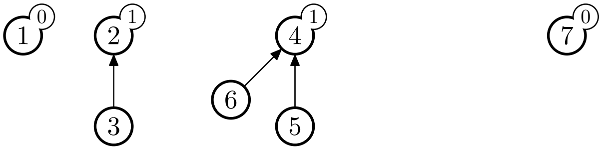
Beispiel: $union(5, 4)$, $union(3, 2)$, $union(5, 6)$, $union(4, 7)$, $union(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

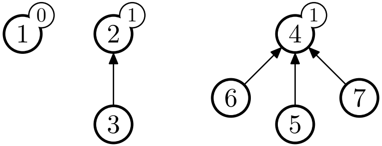
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

$union(a, b)$: Verwende die ranghöhere Wurzel

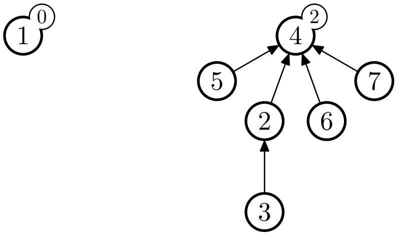
Beispiel: $union(5, 4)$, $union(3, 2)$, $union(5, 6)$, $union(4, 7)$, $union(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

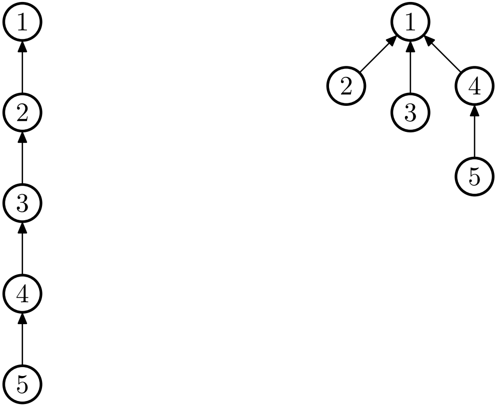
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



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** $p[x] :=$ Find_Set(p[x]) **fi**;
return p[x];

Algorithmus

procedure Union(x, y) :
$x :=$ Find_Set(x);
$y :=$ Find_Set(y);
if $rank[x] > rank[y]$ **then** $p[y] := x$
  else $p[x] := y$;
    if $rank[x] = rank[y]$ **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 $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. $\square$

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 $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. $\square$

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

$\square$

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...

Übersicht

Algorithmische Geometrie

Probleme der Algorithmischen Geometrie haben üblicherweise diese Eigenschaften:

- Die Eingabe besteht aus Punkten, Segmenten, Kreisbögen usw. in der euklidischen Ebene.
- Die Fragestellung ist relativ einfach.
- Sehr große Eingaben müssen bewältigt werden.

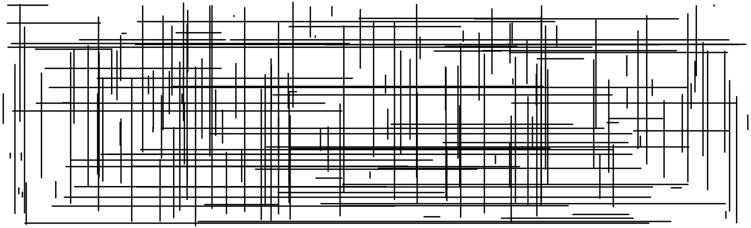
Anwendungen beispielsweise im VLSI-Design.

Übersicht

Schnitte von Segmenten

Eingabe: Horizontale und vertikale Segmente

Ausgabe: Paare von Segmenten, die sich schneiden



Naiver Algorithmus:

Teste alle Paare, ob sie sich schneiden.

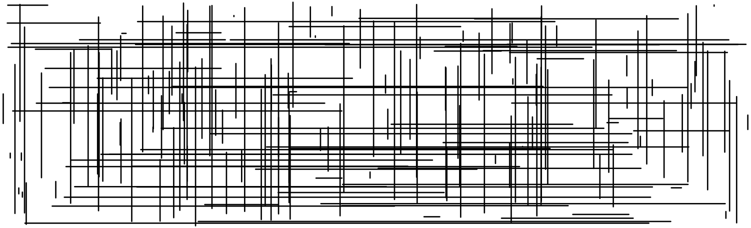
Laufzeit: $\Theta(n^2)$

Variante: Finde heraus, ob es einen Schnitt gibt.

Schnitte von Segmenten

Eingabe: Horizontale und vertikale Segmente

Ausgabe: Paare von Segmenten, die sich schneiden



Naiver Algorithmus:

Teste alle Paare, ob sie sich schneiden.

Laufzeit: $\Theta(n^2)$

Variante: Finde heraus, ob es einen Schnitt gibt.