

# Algorithmen und Datenstrukturen

## C3. Union-Find

Gabriele Röger

Universität Basel

# Algorithmen und Datenstrukturen

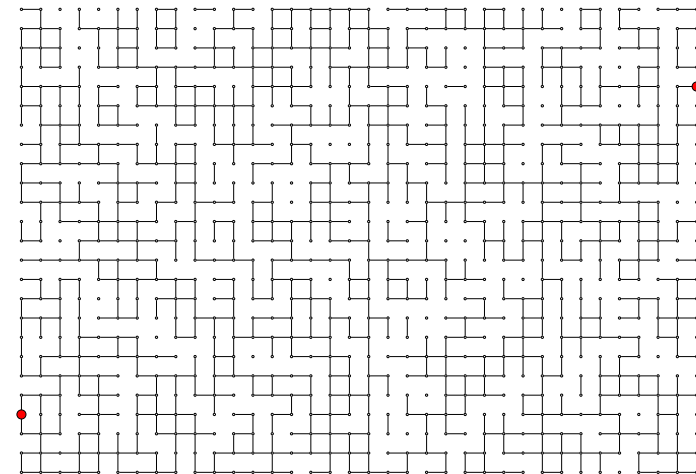
## — C3. Union-Find

### C3.1 Union-Find

### C3.2 Zusammenhangskomponenten und Äquivalenzklassen

## C3.1 Union-Find

## Fragen



Sind die roten Knoten verbunden?  
Wie viele Zusammenhangskomponenten hat der Graph?

## Union-Find-Datentyp

Können Frage mit Hilfe folgender Datenstruktur beantworten:

```

1 class UnionFind:
2     # Initialisiert n Knoten mit Namen 0, ..., n-1
3     def __init__(n: int) -> None
4
5     # Fügt Verbindung zwischen v und w hinzu
6     def union(v: int, w: int) -> None
7
8     # Komponentenbezeichner für v
9     def find(v: int) -> int
10
11    # Sind v und w verbunden?
12    def connected(v: int, w: int) -> bool
13
14    # Anzahl der Zusammenhangskomponenten
15    def count() -> int

```

## (Etwas) naiver Algorithmus: Quick-Find

- ▶ Für  $n$  Knoten: Array `id` der Länge  $n$
- ▶ Eintrag an Stelle  $i$  ist Bezeichner der Zusammenhangskomponente, in der Knoten  $i$  liegt.
- ▶ Anfänglich liegt jeder Knoten (alleine) in seiner eigenen Zusammenhangskomponente (insgesamt  $n$  Stück).
- ▶ Aktualisiere das Array bei jedem Aufruf von `union`.

## Quick-Find-Algorithmus

```

1 class QuickFind:
2     def __init__(self, no_nodes):
3         self.id = list(range(no_nodes))
4         self.components = no_nodes
5
6     def find(self, v):
7         return self.id[v]
8
9     def union(self, v, w):
10        id_v = self.find(v)
11        id_w = self.find(w)
12        if id_v == id_w: # already in same component
13            return
14        # replace all occurrences of id_v in self.id with id_w
15        for i in range(len(self.id)):
16            if self.id[i] == id_v:
17                self.id[i] = id_w
18        self.components -= 1 # we merged two components

```

[0, 1, ..., no\_nodes-1]

## Quick-Find-Algorithmus (Fortsetzung)

```

20    def connected(self, v, w):
21        return self.find(v) == self.find(w)
22
23    def count(self):
24        return self.components

```

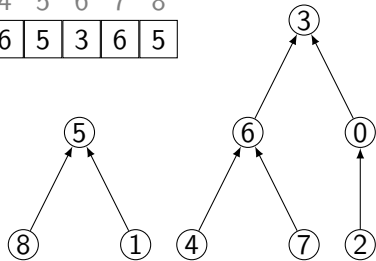
Aufwand?

- ▶ Kostenmodell = Anzahl Arrayzugriffe
- ▶ ein Arrayzugriff für jeden Aufruf von `find`
- ▶ zwischen  $n + 3$  und  $2n + 1$  Arrayzugriffe für jeden Aufruf von `union`, der zwei Komponenten vereinigt

## Etwas besserer Algorithmus: Quick-Union

- ▶ (implizite) Baumstruktur zur Repräsentation jeder Zusammenhangskomponente
- ▶ Repräsentiert durch Array mit Eintrag des Elternknotens (Wurzel: Referenz auf sich selbst)

0	1	2	3	4	5	6	7	8
3	5	0	3	6	5	3	6	5



- ▶ Wurzelknoten dient als Bezeichner der Zusammenhangskomponente

## Quick-Union-Algorithmus

```

1 class QuickUnion:
2     def __init__(self, no_nodes):
3         self.parent = list(range(no_nodes))
4         self.components = no_nodes
5
6     def find(self, v):
7         while self.parent[v] != v:
8             v = self.parent[v]
9         return v
10
11    def union(self, v, w):
12        id_v = self.find(v)
13        id_w = self.find(w)
14        if id_v == id_w: # already in same component
15            return
16        self.parent[id_v] = id_w
17        self.components -= 1
18
19        # connected und count wie bei QuickFind
  
```

## Erste Verbesserung

- ▶ **Problem bei Quick-Union:** Bäume können zu Ketten entarten  
→ `find` benötigt lineare Zeit in der Grösse der Komponente.
- ▶ **Idee:** Hänge in `union` flacheren Baum an Wurzel des tieferen Baums

## Ranked-Quick-Union-Algorithmus

```

1 class RankedQuickUnion:
2     def __init__(self, no_nodes):
3         self.parent = list(range(no_nodes))
4         self.components = no_nodes
5         self.rank = [0] * no_nodes # [0, ..., 0]
6
7     def union(self, v, w):
8         id_v = self.find(v)
9         id_w = self.find(w)
10        if id_v == id_w:
11            return
12        if self.rank[id_w] < self.rank[id_v]:
13            self.parent[id_w] = id_v
14        else:
15            self.parent[id_v] = id_w
16            if self.rank[id_v] == self.rank[id_w]:
17                self.rank[id_w] += 1
18        self.components -= 1
19
20        # connected, count und find wie bei QuickUnion
  
```

## Zweite Verbesserung

### Pfadkompression

- ▶ **Idee:** Hänge in `find` alle traversierten Knoten direkt an die Wurzel um
- ▶ Wir aktualisieren die Höhe des Baumes bei der Pfadkompression nicht.
  - ▶ Wert von `rank` kann von tatsächlicher Höhe abweichen.
  - ▶ Deshalb heisst er auch **Rang** (rank) statt Höhe.

## Ranked-Quick-Union-Algorithmus mit Pfadkompression

```

1 class RankedQuickUnionWithPathCompression:
2     def __init__(self, no_nodes):
3         self.parent = list(range(no_nodes))
4         self.components = no_nodes
5         self.rank = [0] * no_nodes # [0, ..., 0]
6
7     def find(self, v):
8         if self.parent[v] == v:
9             return v
10        root = self.find(self.parent[v])
11        self.parent[v] = root
12        return root
13
14        # connected, count und union wie bei RankedQuickUnion

```

## Diskussion

- ▶ Mit allen Verbesserungen erreichen wir **beinahe konstante amortisierte Kosten** für alle Operationen
- ▶ **Genauer:** [Tarjan 1975]
  - ▶  $m$  Aufrufe von `find` bei  $n$  Objekten (und höchstens  $n - 1$  Aufrufe von `union`, die zwei Komponenten vereinigen)
  - ▶  $O(m\alpha(m, n))$  Arrayzugriffe
  - ▶  $\alpha$  ist Umkehrfunktion einer Variante der **Ackermann-Funktion**
  - ▶ In der Praxis ist  $\alpha(m, n) \leq 3$ .
- ▶ **Trotzdem:** es kann keinen Union-Find-Algorithmus geben, der lineare Zeit garantieren kann. (unter „Cell-Probe“-Berechnungsmodell)

## Vergleich mit explorationsbasiertem Verfahren

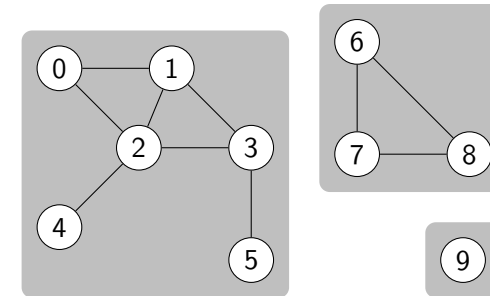
- ▶ **Kapitel C2:** Algorithmus **ConnectedComponents**, der auf **Graphenexploration** basiert
- ▶ Nach der Vorberechnung kosten Anfragen nur konstante Zeit.
- ▶ In der Praxis ist Union-Find meist schneller, da der Graph für viele Zwecke nicht vollständig aufgebaut werden muss.
- ▶ Ist der Graph schon aufgebaut, kann Graphenexploration besser sein.
- ▶ Weiterer Vorteil von Union-Find
  - ▶ **Online-Verfahren**
  - ▶ problemloses Hinzufügen weiterer Kanten

## C3.2 Zusammenhangskomponenten und Äquivalenzklassen

## Wiederholung: Zusammenhangskomponenten

### Ungerichteter Graph

- ▶ Zwei Knoten  $u$  und  $v$  sind genau dann in der gleichen **Zusammenhangskomponente**, wenn es einen Pfad zwischen  $u$  und  $v$  gibt (= Knoten  $u$  und  $v$  **verbunden** sind).



## Zusammenhangskomponenten: Eigenschaften

- ▶ Die Zusammenhangskomponenten definieren eine **Partition** der Knoten:
  - ▶ Jeder Knoten ist in einer Zusammenhangskomponente.
  - ▶ Kein Knoten ist in mehr als einer Zusammenhangskomponente.
- ▶ „ist verbunden mit“ ist **Äquivalenzrelation**
  - ▶ **reflexiv**: Jeder Knoten ist mit sich selbst verbunden.
  - ▶ **symmetrisch**: Ist  $u$  mit  $v$  verbunden, dann ist  $v$  mit  $u$  verbunden.
  - ▶ **transitiv**: Ist  $u$  mit  $v$  verbunden und  $v$  mit  $w$  verbunden, dann ist  $u$  mit  $w$  verbunden.

## Partition allgemein

### Definition (Partition)

Eine **Partition** einer endlichen Menge  $M$  ist eine Menge  $P$  nicht-leerer Teilmengen von  $M$ , so dass

- ▶ jedes Element von  $M$  in einer Menge in  $P$  vorkommt:  
 $\bigcup_{S \in P} S = M$ , und
- ▶ die Mengen in  $P$  paarweise disjunkt sind:  
 $S \cap S' = \emptyset$  für  $S, S' \in P$  mit  $S \neq S'$ .

Die Mengen in  $P$  heißen **Blöcke**.

$$M = \{e_1, \dots, e_5\}$$

- ▶  $P_1 = \{\{e_1, e_4\}, \{e_3\}, \{e_2, e_5\}\}$  ist eine Partition von  $M$ .
- ▶  $P_2 = \{\{e_1, e_4, e_5\}, \{e_3\}\}$  ist keine Partition von  $M$ .
- ▶  $P_3 = \{\{e_1, e_4, e_5\}, \{e_3\}, \{e_2, e_5\}\}$  ist keine Partition von  $M$ .
- ▶  $P_4 = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_5\}\}$  ist eine Partition von  $M$ .

## Äquivalenzrelation allgemein

### Definition (Äquivalenzrelation)

Eine **Äquivalenzrelation** auf einer Menge  $M$  ist eine **symmetrische, transitive und reflexive** Relation  $R \subseteq M \times M$ .

Wir schreiben  $a \sim b$  für  $(a, b) \in R$  und sagen  **$a$  ist äquivalent zu  $b$** .

- ▶ **symmetrisch:**  $a \sim b$  impliziert  $b \sim a$
- ▶ **transitiv:**  $a \sim b$  und  $b \sim c$  impliziert  $a \sim c$
- ▶ **reflexiv:** für alle  $e \in M$ :  $e \sim e$

## Äquivalenzklassen

### Definition (Äquivalenzklassen)

Sei  $R$  eine Äquivalenzrelation auf der Menge  $M$ .

Die **Äquivalenzklasse** von  $a \in M$  ist die Menge

$$[a] = \{b \in M \mid a \sim b\}.$$

- ▶ Die Menge aller Äquivalenzklassen ist eine Partition von  $M$ .
- ▶ **Umgekehrt:**  
Für Partition  $P$  definiere  $R = \{(x, y) \mid \exists B \in P : x, y \in B\}$   
(also  $x \sim y$  genau dann, wenn  $x$  und  $y$  im gleichen Block).  
Dann ist  $R$  eine Äquivalenzrelation.
- ▶ Können Partitionen als Äquivalenzklassen betrachten und umgekehrt.

## Union-Find und Äquivalenzen

- ▶ **Gegeben:** endliche Menge  $M$ ,  
Sequenz  $s$  von Äquivalenzen  $a \sim b$  über  $M$
- ▶ Fasse Äquivalenzen als Kanten in Graphen mit Knotenmenge  $M$  auf.
- ▶ Die Zusammenhangskomponenten entsprechen den Äquivalenzklassen der **feinsten Äquivalenzrelation**, die alle Äquivalenzen aus  $s$  enthält.
  - ▶ keine „unnötigen“ Äquivalenzen

Wir können die **Union-Find-Datenstruktur** zur **Bestimmung der Äquivalenzklassen** verwenden.