Algorithmen und Datenstrukturen

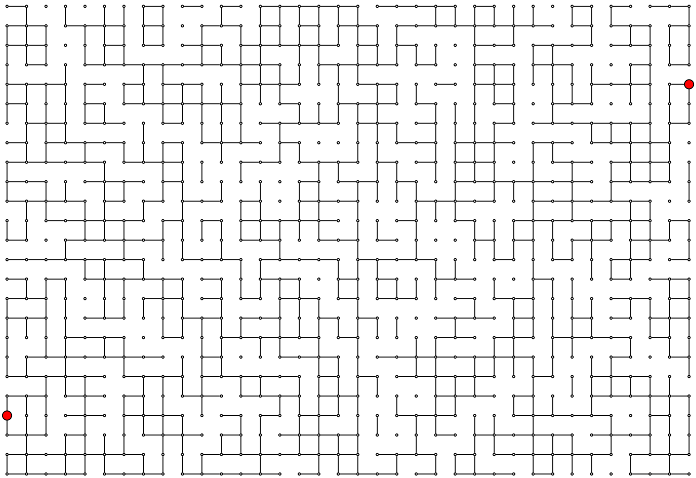
C3. Union-Find

Gabriele Röger

Universität Basel

Union-Find

Fragen



Sind die roten Knoten verbunden?

Wie viele Zusammenhangskomponenten hat der Graph?

Union-Find-Datentyp

Können Frage mit Hilfe folgendem Datentyp beantworten:

```
class UnionFind:
  # Initialisiert n Knoten mit Namen 0, ..., n-1
  def __init__(n: int) -> None

  # Fügt Verbindung zwischen v und w hinzu
  def union(v: int, w: int) -> None

  # Komponentenbezeichner für v
  def find(v: int) -> int

  # Sind v und w verbunden?
  def connected(v: int, w: int) -> bool

  # Anzahl der Zusammenhangskomponenten
  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.

(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).

(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

```python
class QuickFind:
    def __init__(self, no_nodes):
        self.id = list(range(no_nodes))
        self.components = no_nodes

    def find(self, v):
        return self.id[v]

    def union(self, v, w):
        id_v = self.find(v)
        id_w = self.find(w)
        if id_v == id_w:  # already in same component
            return
        # replace all occurrences of id_v in self.id with id_w
        for i in range(len(self.id)):
            if self.id[i] == id_v:
                self.id[i] = id_w
        self.components -= 1  # we merged two components
```

[0, 1, ..., no_nodes-1]

Quick-Find-Algorithmus (Fortsetzung)

```
    def connected(self, v, w):
        return self.find(v) == self.find(w)

    def count(self):
        return self.components
```

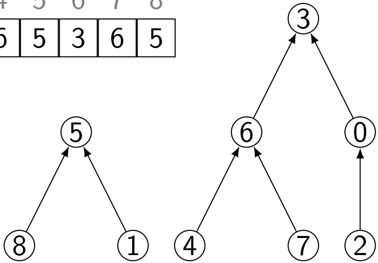
Aufwand?

- 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)

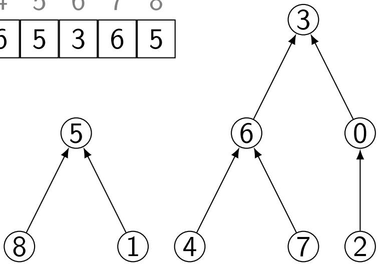
0	1	2	3	4	5	6	7	8
3	5	0	3	6	5	3	6	5



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

```python
class QuickUnion:
    def __init__(self, no_nodes):
        self.parent = list(range(no_nodes))
        self.components = no_nodes

    def find(self, v):
        while self.parent[v] != v:
            v = self.parent[v]
        return v

    def union(self, v, w):
        id_v = self.find(v)
        id_w = self.find(w)
        if id_v == id_w:  # already in same component
            return
        self.parent[id_v] = id_w
        self.components -= 1

    # connected und count wie bei QuickFind
```

Erste Verbesserung

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

Ranked-Quick-Union-Algorithmus

```python
class RankedQuickUnion:
    def __init__(self, no_nodes):
        self.parent = list(range(no_nodes))
        self.components = no_nodes
        self.rank = [0] * no_nodes  # [0, ..., 0]

    def union(self, v, w):
        id_v = self.find(v)
        id_w = self.find(w)
        if id_v == id_w:
            return
        if self.rank[id_w] < self.rank[id_v]:
            self.parent[id_w] = id_v
        else:
            self.parent[id_v] = id_w
            if self.rank[id_v] == self.rank[id_w]:
                self.rank[id_w] += 1
        self.components -= 1

    # 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

```python
class RankedQuickUnionWithPathCompression:
    def __init__(self, no_nodes):
        self.parent = list(range(no_nodes))
        self.components = no_nodes
        self.rank = [0] * no_nodes  # [0, ..., 0]

    def find(self, v):
        if self.parent[v] == v:
            return v
        root = self.find(self.parent[v])
        self.parent[v] = root
        return root

    # connected, count und union wie bei RankedQuickUnion
```

Diskussion

- Mit allen Verbesserungen erreichen wir beinahe konstante amortisierte Kosten für alle Operationen

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
  - α ist Umkehrfunktion einer Variante der Ackermann-Funktion
  - In der Praxis ist $\alpha(m,n) \leq 3$.

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
  - α 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

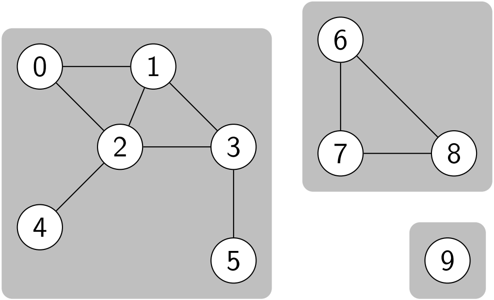
- Kapitel C2: Algorithmus `ConnectedComponents`, der auf Graphenexploration basiert
- Nach der Vorberechnung kosten Anfragen nur konstante Zeit.
- In der Praxis ist Union-Find meist schneller, da der Graph für viele Zwecke nicht vollständig aufgebaut werden muss.
- Ist der Graph schon aufgebaut, kann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

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.

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

- $P_1 = \{\{e_1, e_4\}, \{e_3\}, \{e_2, e_5\}\}$
- $P_2 = \{\{e_1, e_4, e_5\}, \{e_3\}\}$
- $P_3 = \{\{e_1, e_4, e_5\}, \{e_3\}, \{e_2, e_5\}\}$
- $P_4 = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_5\}\}$

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.

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

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

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.

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

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

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.

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

- $P_1 = \{\{e_1, e_4\}, \{e_3\}, \{e_2, e_5\}\}$ 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.
- $P_4 = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_5\}\}$

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.

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

- $P_1 = \{\{e_1, e_4\}, \{e_3\}, \{e_2, e_5\}\}$ 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

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

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.

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.

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

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.

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

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.