# Algorithms and Data Structures

## C3. Disjoint-set Data Structure/Union-Find

Gabriele Röger

University of Basel

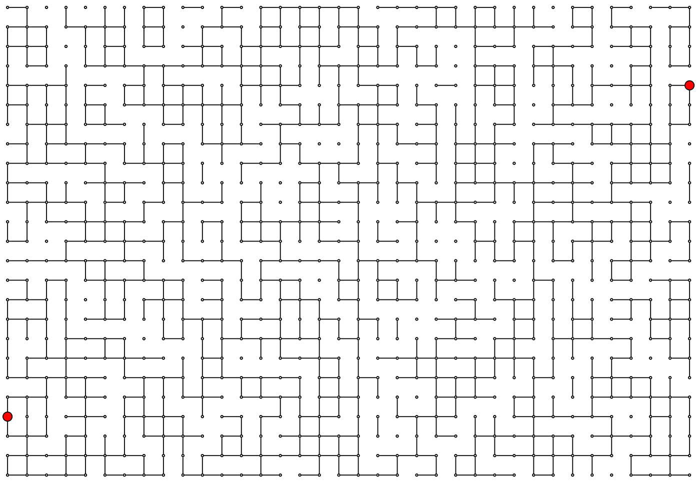
May 8, 2024

# Algorithms and Data Structures

May 8, 2024 — C3. Disjoint-set Data Structure/Union-Find

## C3.1 Union-Find

## Questions



Are the red vertices connected?

How many connected components does the graph have?

## Connected Components as Disjoint Sets

Set of conn. components as collection of disjoint sets of objects.

- One set for all vertices of one connected component.
- Operations:
  - Union: Given two objects, merge the sets that contain them into one.
    Introduce a new edge between the given vertices, connecting their connected components.
  - Find: Given an object, return a representative of the set that contains it.
    Given a vertex, return a representative vertex for its connected component.
    - Must return the same representative for all objects in the set.
    - The representative may only change if set gets merged.
    - Two objects are in the same set (two vertices are connected) if find returns the same representative for them.
  - Count: Return the number of sets
    Return the number of connected components.

## Union-Find Data Type

```
class UnionFind:
  # Initialization for n objects (with names 0, ..., n-1).
  def __init__(n: int) -> None

  # Merge the sets containing objects v and w.
  def union(v: int, w: int) -> None

  # Representative for set containing v.
  # May change if set is merged by call of union,
  # but not otherwise.
  def find(v: int) -> int

  # Number of sets.
  def count() -> int
```

## (Somewhat) Naive Algorithm: Quick-Find

- For $n$ objects: Array `representative` of length $n$.
- Entry at position $i$ is representative of the set containing $i$.
- Initially, every object is (alone) in its own set, and thus its representative.
- Update the array in every call of `union`.

## Quick-Find Data Structure

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

    def count(self):
        return self.components

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

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

## Quick-Find Data Structure (Continued)

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

Running time?

- Cost model = number of array accesses
- one access for every call of `find`
- between $n+3$ and $2n+1$ accesses
  for every call of `union` that merges two components

## Better: Quick-Union aka Disjoint-set Forest

- (implicit) tree for representing each set
- represented as array with parent nodes as entries (root: reference to itself)

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

$$\begin{tikzcd}
 & & & & & 3 & \\
 & 5 & & & 6 \arrow[ur] & & 0 \arrow[ul] \\
8 \arrow[ur] & & 1 \arrow[ul] & 4 \arrow[ur] & & 7 \arrow[ul] & 2 \arrow[u]
\end{tikzcd}$$

- Root node serves as representative of the set.

## Quick-Union Data Structure

```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):
        repr_v = self.find(v)
        repr_w = self.find(w)
        if repr_v == repr_w:  # already in same component
            return
        self.parent[repr_v] = repr_w
        self.components -= 1

    # count as in QuickFind
```

## First Improvement

- Problem with Quick-Union: Trees can degenerate into chains.
  $\rightarrow$ `find` requires linear time in the size of the set.
- Idea: In union the root of the tree with lower height becomes a child of the root of the higher tree.

## Ranked Quick-Union Algorithm

```
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):
        repr_v = self.find(v)
        repr_w = self.find(w)
        if repr_v == repr_w:
            return
        if self.rank[repr_w] < self.rank[repr_v]:
            self.parent[repr_w] = repr_v
        else:
            self.parent[repr_v] = repr_w
            if self.rank[repr_v] == self.rank[repr_w]:
                self.rank[repr_w] += 1
        self.components -= 1

    # connected, count and find as in QuickUnion
```

## Second Improvement

Path Compression

- Idea: During find, reconnect all traversed nodes to the root.
- We do not update the height of the tree during path compression.
    - Value of `rank` can deviate from the actual height.
    - That's why it is called rank and not height.

## Ranked Quick-Union Algorithm with Path Compression

```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 and union as in RankedQuickUnion
```

## Discussion

- With all improvements, we achieve almost constant amortized cost for all operations.
- More precisely: [Tarjan 1975]
  - $m$ calls of `find` for $n$ objects (and at most $n-1$ calls of `union`, merging two components)
  - $O(m\alpha(m,n))$ array accesses
  - $\alpha$ is inverse of a variant of the Ackermann function
  - In practise is $\alpha(m,n) \leq 3$.
- Nevertheless: there cannot be a union-find structure that guarantees linear running time.
  (under cell-probe model, only accounting for memory access)

## Comparison to Exploration-based Approach

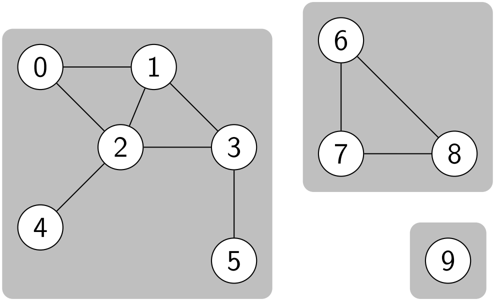
- Chapter C2: Algorithm `ConnectedComponents`, based on graph exploration.
- After the precomputation, queries only require constant time.
- In practise, disjoint-set forests are often faster, because for many applications, we do not have to build up the full tree.
- If the graph has already been built up, graph exploration can be better.
- Another advantage of union find:
  - Online approach
  - We can easily introduce further edges.

## C3.2 Connected Components and Equivalence Classes

## Reminder: Connected Components

Undirected graph

- Two vertices $u$ and $v$ are in the same connected component if there is a path between $u$ and $v$ (= vertices $u$ and $v$ are connected).



## Connected Components: Properties

- The connected components define a partition of the vertices:
  - Every vertex is in a connected component.
  - No vertex is in more than one connected component.
- „is connected with“ is an equivalence relation.
  - reflexive: Every vertex is connected with itself.
  - symmetric: If $u$ is connected with $v$, then $v$ is connected with $u$.
  - transitive: If $u$ is connected with $v$, and $v$ with $w$, then $u$ is connected with $w$.

## Partition in General

**Definition (Partition)**

A partition of a finite set $M$ is a set $P$ of non-empty subsets of $M$, such that

- every element of $M$ is in some set in $P$: $\bigcup_{S \in P} S = M$, and
- that sets in $P$ are pairwise disjoint: $S \cap S' = \emptyset$ for $S, S' \in P$ with $S \neq S'$.

The sets in $P$ are called blocks.

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

- $P_1 = \{\{e_1, e_4\}, \{e_3\}, \{e_2, e_5\}\}$ is a partition of $M$.
- $P_2 = \{\{e_1, e_4, e_5\}, \{e_3\}\}$ is not a partition of $M$.
- $P_3 = \{\{e_1, e_4, e_5\}, \{e_3\}, \{e_2, e_5\}\}$ is not a partition of $M$.
- $P_4 = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_5\}\}$ is a partition of $M$.

## Equivalence Relations in General

**Definition (Equivalence Relation)**

An equivalence relation over set $M$ is a symmetric, transitive and reflexive relation $R \subseteq M \times M$.
We write $a \sim b$ for $(a, b) \in R$ and say that $a$ is equivalent to $b$.

- symmetric: $a \sim b$ implies $b \sim a$
- transitive: $a \sim b$ and $b \sim c$ implies $a \sim c$
- reflexive: for all $e \in M$: $e \sim e$

## Equivalence Classes

**Definition (Equivalence Classes)**

Let $R$ be an equivalence relation over $M$.
The equivalence class of $a \in M$ is the set

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

- The set of all equivalence classes is a partition of $M$.
- Vice versa:
  For partition $P$ define $R = \{(x, y) \mid \exists B \in P : x, y \in B\}$
  (i.e. $x \sim y$ if and only if $x$ and $y$ are in the same block).
  Then $R$ is an equivalence relation.
- We can consider blocks in partitions as equivalence classes and vice versa.

## Union-Find and Equivalence

- Given: finite set $M$, sequence $s$ of equivalences $a \sim b$ over $M$
- Consider equivalences as edges in a graph with set $M$ of vertices.
- The connected components correspond to the equivalence classes of the finest equivalence relation that considers all equivalences from $s$.
  - no “unnecessary” equivalences.

Can use union-find data structures to determine equivalence classes.

## C3.3 Summary

## Summary

- A union-find data structure maintains a collection of disjoint sets.
  - union: merge two sets.
  - find: identify the set containing an object and return its representative.
- Good implementation: Disjoint-set forest with improvements to keep the height of the trees low:
  - Union adjoins the shorter tree to the taller tree.
  - Find reconnects traversed nodes to the root (path compression).
- Applications:
  - Connected components
  - Finest equivalence relation