Go: Union-Find Datastructure

CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Logistics & Recap
2. Union-Find datastructure
3. Using Union-Find datastructures in Go environments

Logistics

- **TA Office hours:** Every **Thursday** from **1:00pm-2:00pm** in **UCOMM 3-136**
  - Drop in basis; just show up and ask questions
  - Starting this week (tomorrow)
- **Practice quiz questions:** Released this Friday (**Jan 16**)
  - Answers released Tuesday (Jan 20)
- **Quiz 1:** Friday, **Jan 23**
  - In-class, full 50 minutes
  - No need to email if you have to miss it; up to 3 replaced by final exam automatically
  - Questions will be very similar to practice questions
  - (at least 3 will be *suspiciously* similar!)

Recap: Go Implementation Issues

We looked at how the **go/go_helper.py** program implements a Go environment:

- **Board** represented as a 1-dimensional array of points
  - For an M-row, N-column board,
    Point at row r and column c is stored at index $rN + c$
  - Guarded representation adds one column and two rows of "guards"
    - Every point now has exactly 4 neighbours
- **Capture** is computed by searching the neighbours of a newly-placed stone
    - Follow all neighbours of same colour looking for an Empty point
    - if none found, block is captured
- **Scoring**: Search from each Empty point on the board to find Black or White stones
- **Superko** detected by storing all previous positions

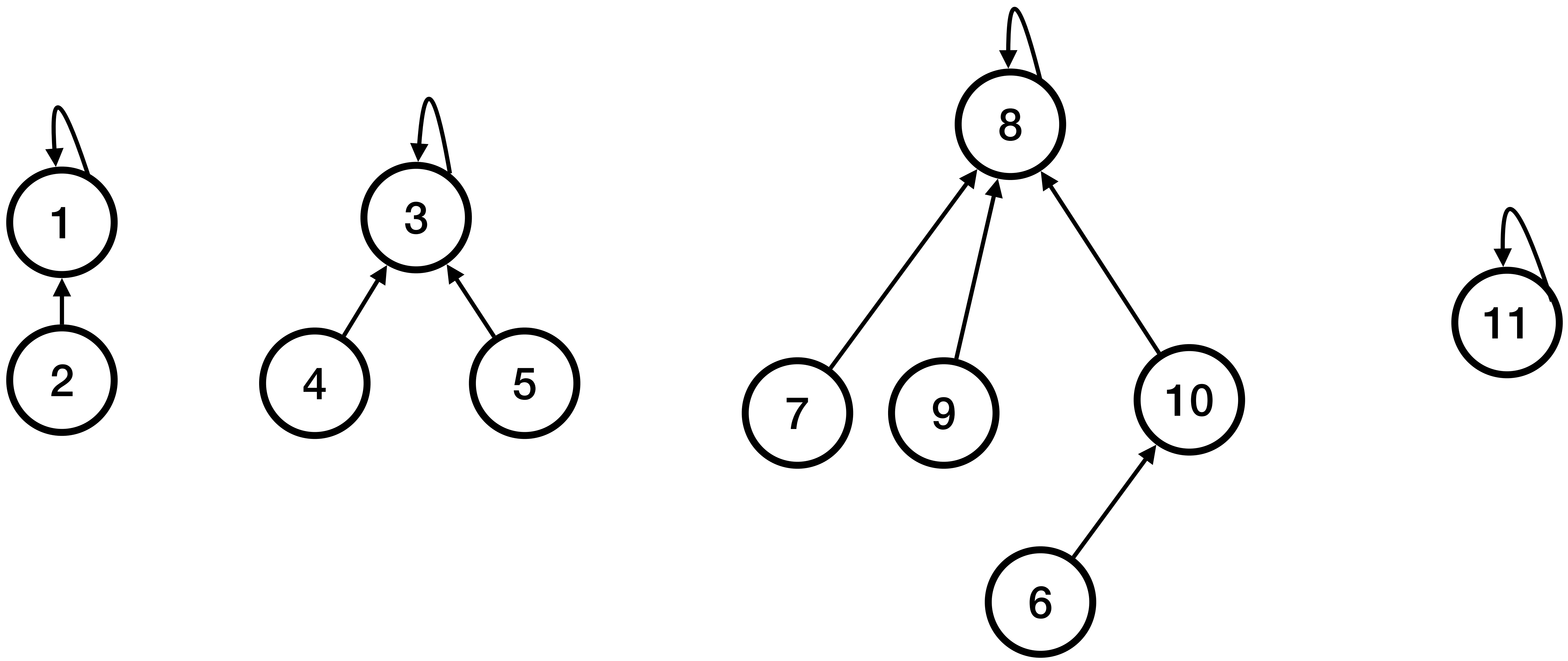
Tracking Blocks

- Searching for liberties after every stone means recomputing blocks & liberties
- What if we instead tracked:
  - Which block a point belongs to
  - Which liberties the block has
- Then, after placing a stone:
  1. Check if we've removed the last liberty of any adjacent block
  2. Add the new stone to the appropriate block(s)
- This is exactly the approach taken in example code **hexgo/stone_board.py**
- Key component: **Union-Find** datastructure for tracking blocks

Union-Find Datastructure

- A **Union-Find datastructure** tracks a partition of a set of **items**
  - I.e., each item belongs to **exactly one group**
  - For Go (and Hex): Items are the **points**, and groups are the **blocks**
- We represent each group as a **tree** of items
  - Each item has a single **parent**
  - The "name" of the group is the root of its tree
- Initially, each item is in its own singleton group
  - **Question:** What should the parent be set to for each item initially?
- Find(item) operation: returns the group that item belongs to
- Union(group1, group2) operation: Merges group1 and group2 into a single group

Example: Groups representation



Questions

1. What group does node 2 belong to?
2. What about node 5?
3. Node 3?
4. Node 6?
5. Node 11?
6. What procedure should we follow to answer these questions?

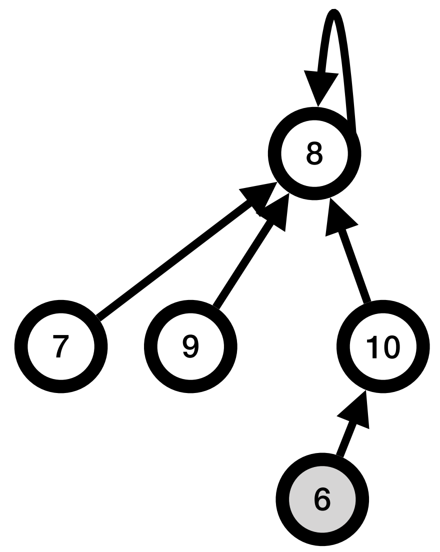
Union-Find Implementation

- In **hexgo/stone_board.py**, track one parent for each point:

```
self.parents   = {}          # point -> parents in block
```

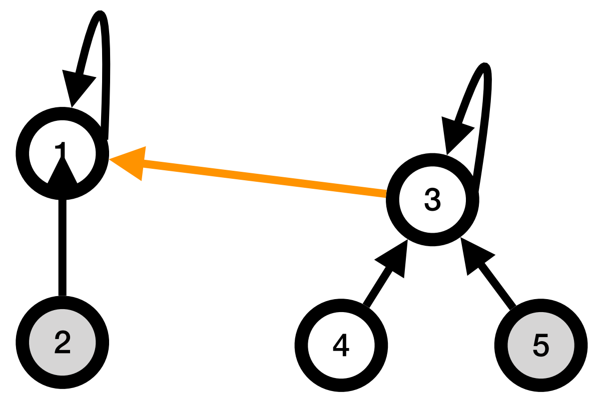
- To find the block that a point belongs to, follow parents until you find an item that is its own parent:

```
class UF:
    def find(parents, x):
      while x != parents[x]:
        x = parents[x]
      return x
```



- To combine the groups that items **x** and **y** belong to, make one of the roots point to the other one:

```
    def union(parents, x, y):
      x = UF.find(parents, x)
      y = UF.find(parents, y)
      parents[y] = x          # x is root of merged trees
      return x, y
```

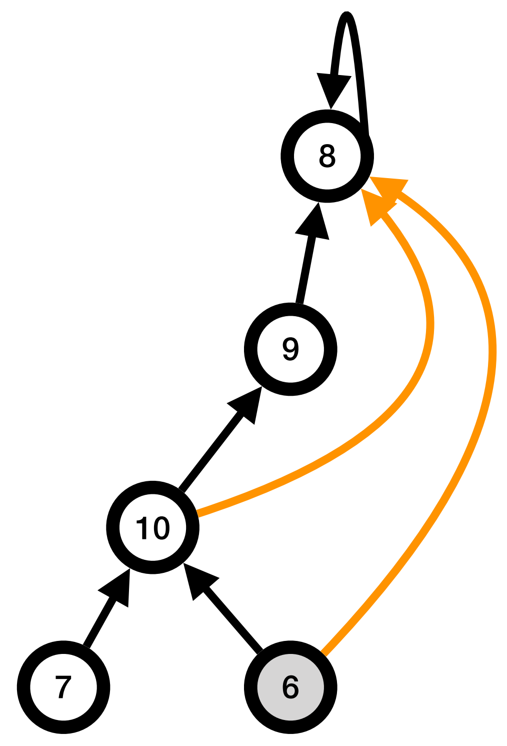


Questions

1. What is the **worst-case time complexity** of **find**?
2. What happens if we call **union** on two items in the **same group**?
3. What is the worst-case **time complexity** of **union**?
4. Could we do better?

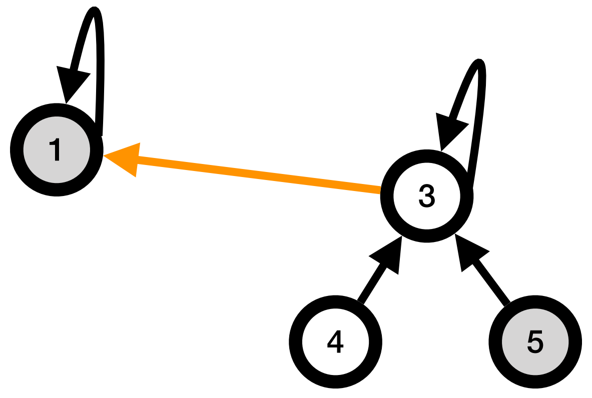
Union-Find Optimizations

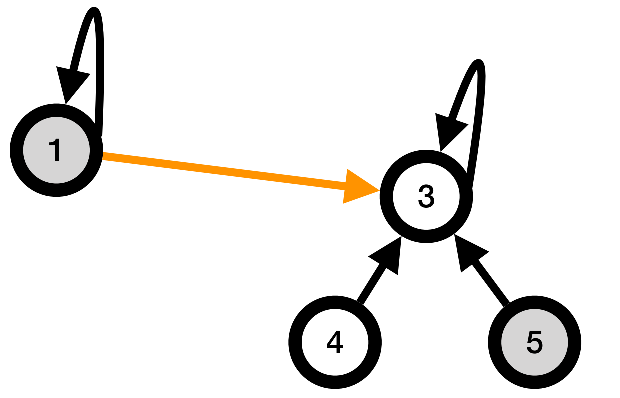
1. Make trees shallow during find:
    - Update each parent along search path to point to root




```
def find(parents, x):
    if x == parents[x]:
        return x
    parents[x] = find(parents, parents[x])
    return parents[x]
```

2. Union by rank:
    - Store an upper bound on a tree's height as rank
    - Larger rank tree becomes parent during union
    - If both have same rank, choose arbitrarily and increment new root's rank






```
def union(parents, ranks, x, y):
    x = find(parents, x)
    y = find(parents, y)

    if ranks[x] == ranks[y]:
        parents[x] = y
        ranks[y] = ranks[y] + 1
    elif ranks[x] < ranks[y]:
        parents[x] = y
    else:
        parents[y] = x
```

hexgo/stone_board.py

Track groups and liberties instead of searching after each move:

```
    self.stones    = [set(), set()]  # start with empty board
    self.nbrs      = {}   # point -> neighbors
    self.liberties = {}   # point -> liberties
    self.parents   = {}   # point -> parents in block
```

Use find and union to update tracking after each move:

```
def add_stone(self, color, point):
    self.stones[color].add(point)
    self.blocks[point].add(point)

    for n in self.nbrs[point]:
        if n in self.stones[color]: # same-color nbr
            self.merge_blocks(n, point)
        if n in self.stones[Cell.opponent(color)]: # opponent nbr
            self.remove_liberties(n, point)
```

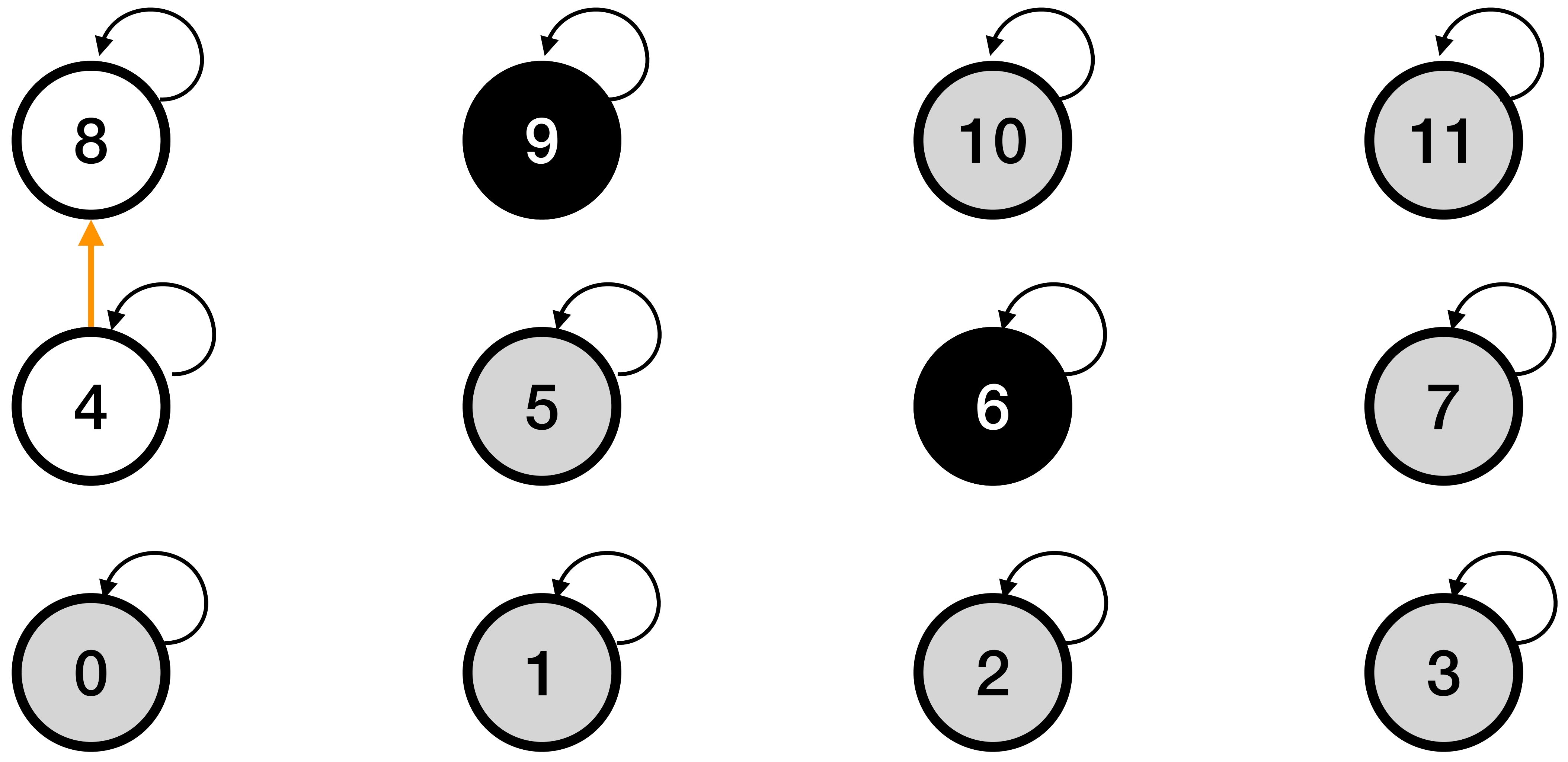
Union to update block membership:

```
    def merge_blocks(self, p, q):
        proot, qroot = UF.union(self.parents, p, q)
        self.liberties[proot].update(self.liberties[qroot])
        self.liberties[proot] -= {q}
```

Find to update block liberties:

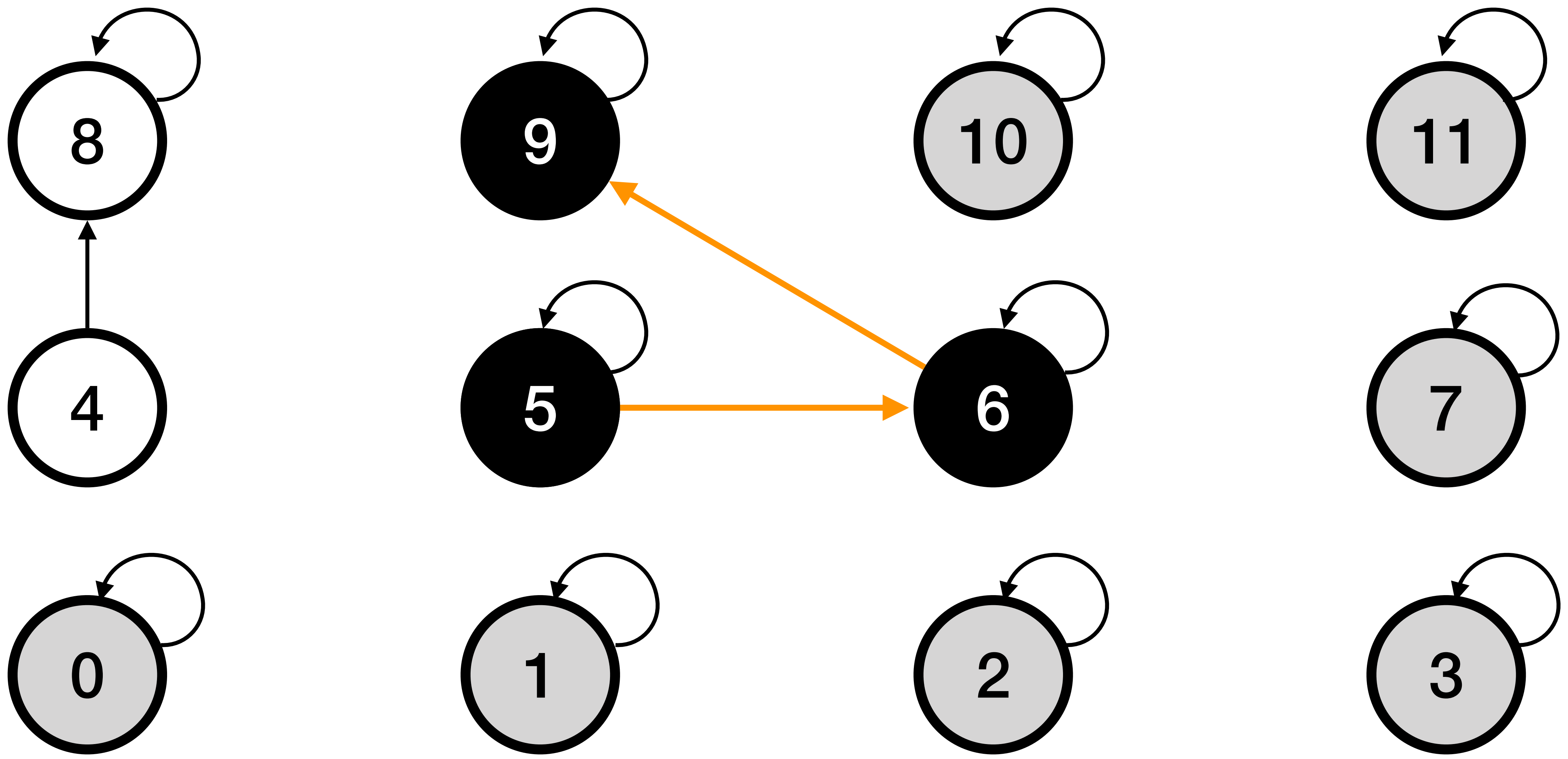
```
    def remove_liberties(self, p, q):
        proot = UF.find(self.parents, p)
        qroot = UF.find(self.parents, q)
        self.liberties[proot] -= {q}
        self.liberties[qroot] -= {q}
```

Example: Union-Find Operations in Go



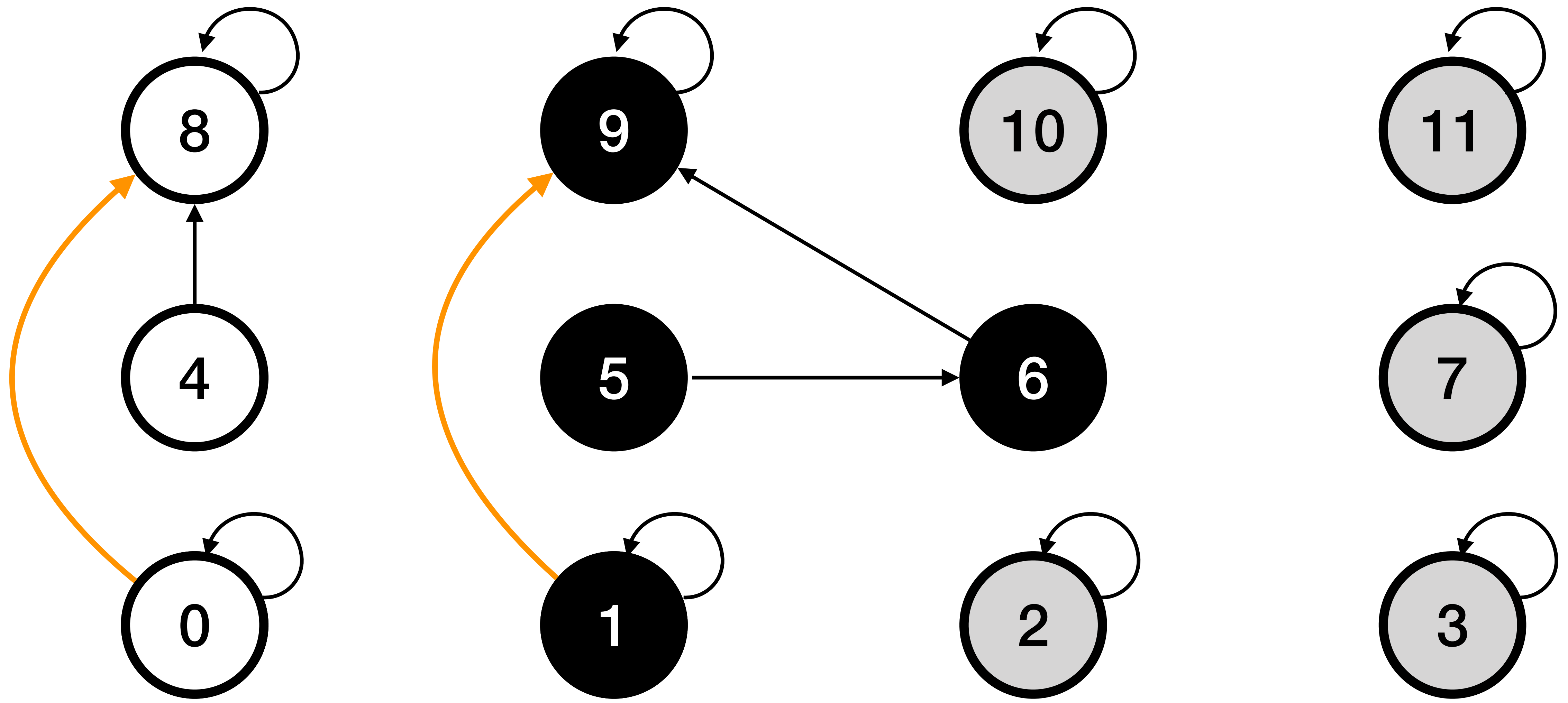
- Black stone on 6
- White stone on 8
- Black stone on 9
- White stone on 4

Example: Union-Find Operations in Go



- Black stone on 6
- White stone on 8
- Black stone on 9
- White stone on 4
- Black stone on 5

Example: Union-Find Operations in Go



- Black stone on 6
- White stone on 8
- Black stone on 9
- White stone on 4
- Black stone on 5
- White stone on 0
- Black stone on 1
    - 0's block's liberties become empty

Summary

- Union-Find datastructure is an efficient way to track block membership over time
- Naive implementation can have poor worst-case performance
  - Optimization: update pointers to be shallow during **find**
  - Optimization: keep trees as short as possible during **union**
- **hexgo/Stone_board.py** implementation:
  - Track blocks in union-find datastructure
  - Track each block's liberties in a set
  - List of each block's neighbours in a lookup table