Algorithms and Data Structures

Introduction

Morgan Ericsson

Today

- » Introduction
- » Introduction to algorithms

About the course

Welcome

Important

- We are back to normal after COVID-19
  - Campus classes, no streaming
  - Written exam
- Stay at home if you are ill or have symptomes
- We recommend that you get vaccinated if offered

Expectations

- We expect that you:
  - Stay up to date with material posted on the MyMoodle room (and resources linked from it) and the Slack channel
  - Do you best and ask for help if you get stuck
  - Treat teachers and other students with respect
- You can expect that we:
  - Do our best to support your learning

Course management

- Course reponsible and examiner:
  - Morgan Ericsson, `@morganericsson`, `morgan.ericsson@lnu.se`
- Teachers:
  - Charilaos Skandylas, `@Charilaos Skandylas`, `charilaos.skandylas@lnu.se`
- Administration:
  - Ewa Püschl, `eva.puschl@lnu.se`

Registration

- If you have not registered, please do
- If you cannot register:
  - Check that you passed the prerequisities
  - If you have, contact me
- The activity control after three weeks will be based on the first assignment
  - You need to have submitted something that “could” work
  - And passes a plagiarism check

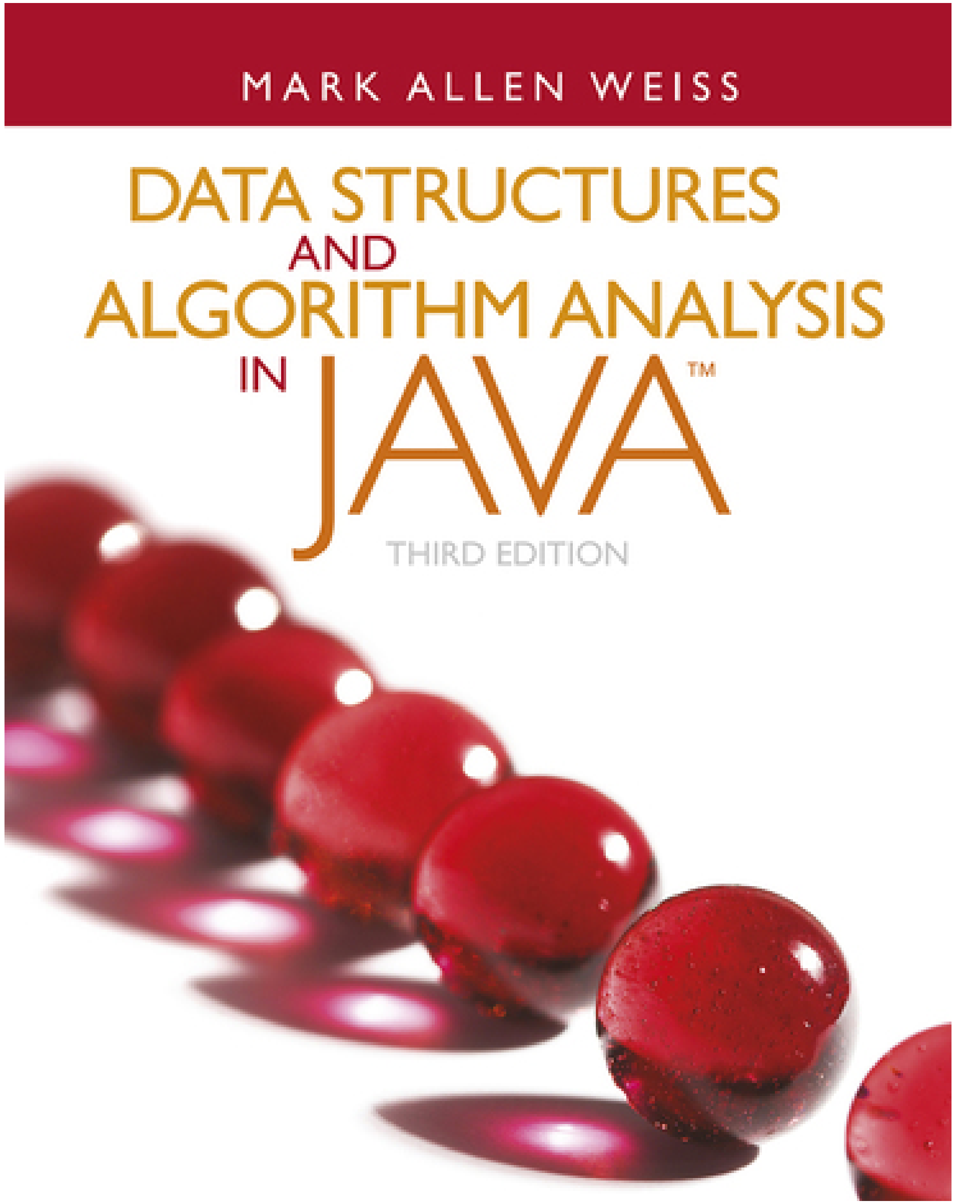
Communications

- All static information is available via MyMoodle
- Assignments are submitted via MyMoodle
- Questions and discussion on Slack
  - Use the course channel as much as possible
  - No DMs unless you need to ask or tell something private
- Try to avoid email. If you must, see policy for DMs
- No messages via MyMoodle!

Examination

- The examination is in two parts:
  - Written exam, 4.5 hp
  - Assignments, 3.0 hp
- You need to pass both to pass the course
- Final grade is 60% written exam and 40% assignments

Learning resources



Course evaluation

» There will be a course evaluation at the end of the course

» It is important that you give honest feedback …

» … and help improve future versions of the course

» Feedback during the course is also welcome

Practical details

Code examples

- Most code examples will be in Python
  - Might introduce new features
  - So, ask if you do not understand
- You will submit your assignments in Java
- All code from the slides is available on GitLab (link on MyMoodle)

Plagiarism

- All submissions will be automatically checked for plagiarism
  - If you are flagged, you will failed the assignment
  - And maybe reported to the disciplinary board
- You are here to learn, so do not copy code from the Internet
  - Instead, understand and adapt!

Lectures

- No streaming or recording
- Slides on MyMoodle, code on GitLab
- Questions in the lecture room or via Slack

Difficult topic

» Algorithms can be difficult!

» Read the book

» Check the slides

» Ask questions

» Start early!

Introduction to algorithms

What?

- An algorithm is a method for solving a problem
  - E.g., sorting or searching
- A data structure is a method to store information
  - E.g., a tree or a linked list

Why?

- Algorithms and data structures are everywhere!
  - Networking and web search
  - AI and machine learning
  - Physics simulations
  - Video compression
  - Security and encryption
  - ...

Pure problem solving

- The design of an algorithm or a data structure is about creating a solution to a problem
- Designing a good and efficient solution can be challenging
  - So, study to avoid repeating
- *“great algorithms are the poetry of computation”*

To be a good programmer

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important.

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

New concepts

- Old roots
  - Euclid’s algorithm (GCD) is an old example
  - Formalized by Church and Turing in the 1930s
- New concepts …
  - Scalability, computatbility, …
- … that make you a better programmer

What will you learn?

» List, stacks, and queues

» Trees

» Hashing

» Sorting

» Graphs

» Algorithm design, e.g., divide and conquer

» Algorithm analysis, e.g., Big-Oh

» N, P, and NP

But I already know that?

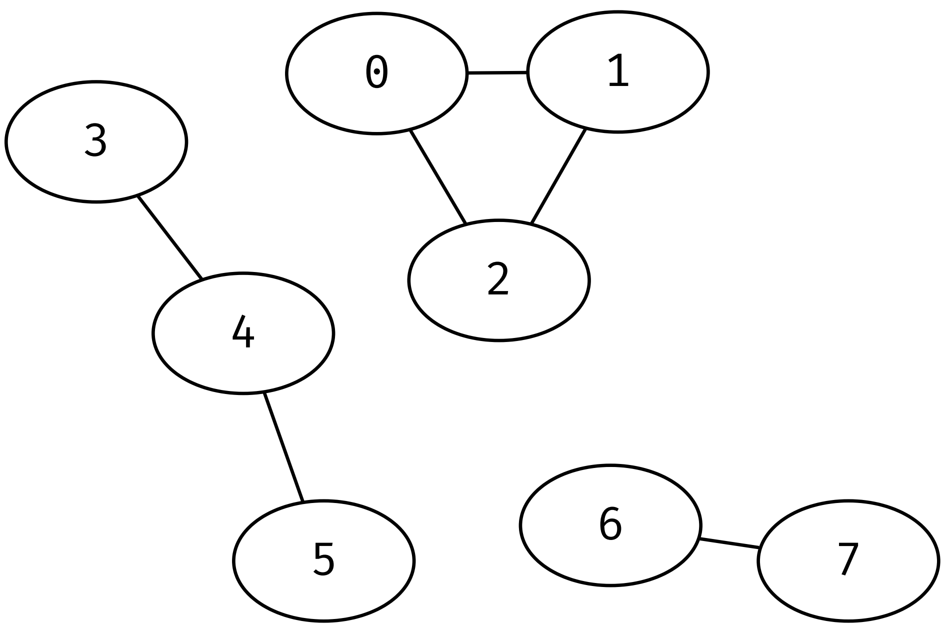
- We implemented most of this in, e.g., 1DV501!
  - No, there is much more to learn!

An example

Note

- This example will introduce some new ideas
- We will return to a lot of what we discuss
- So, try to keep up, but do not expect to understand everything
- Pay attention, you will implement it!

The problem



Given a set of objects, is there a path that connects two specific objects, e.g., 0 and 5.

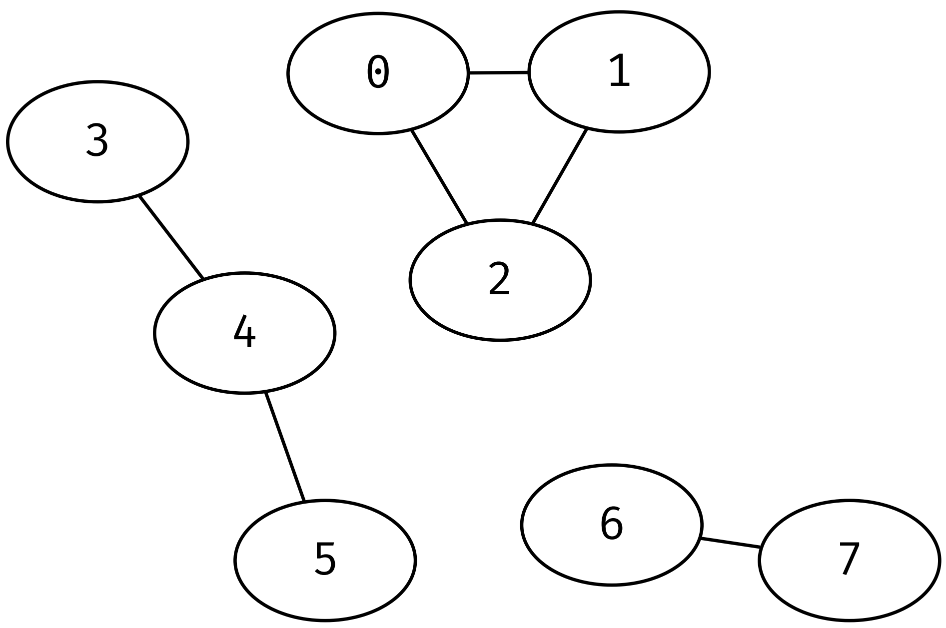
Solution / API

- Data structure with two operations:
  - Union connects two objects:
    - `union(a:int, b:int) -> None`
  - Connected determins whether two objects are connected:
    - `connected(a:int, b:int) -> bool`
- To simplify, we use an integer index to identify the objects

Connections

- We model the connection as an equivalence relation
  - Reflective, i.e., *a* is connected to *a*
  - Symmetric, i.e., if *a* is connected to *b*, then *b* is connected to *a*
  - Transitive, i.e., if *a* is connected to *b* and *b* is connected to *c* then *a* is connected to *c*.
- A Connected component is a maximal set of objects that are mutually connected

Example



There are three connected components: `{0,1,2}`, `{3,4,5}`, and `{6,7}`.

A first attempt

- We use the connected components to formulate a solution
- Each connected component has an id that identifies it
  - When two objects are connected, they get the same id
  - If two objects have the same id, they are connected

A first attempt

```python
def init(N:int) -> list[int]:
    return list(range(N))

def connected(d:list[int], a:int, b:int) -> bool:
    return d[a] == d[b]
```

A first attempt

```python
def union(d:list[int], a:int, b:int) -> None:
    a_id = d[a]
    b_id = d[b]

    for ix, v in enumerate(d):
        if v == a_id:
            d[ix] = b_id
```

A first attempt

```
uf = init(8)
print(uf)
union(uf, 0, 1)
union(uf, 6, 7)
print(uf)
union(uf, 1, 2)
print(uf)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 1, 2, 3, 4, 5, 7, 7]
[2, 2, 2, 3, 4, 5, 7, 7]
```

What if?

```python
# uf = [2, 2, 2, 3, 4, 5, 7, 7]
union(uf, 0, 6)
print(uf)
```

```
[7, 7, 7, 3, 4, 5, 7, 7]
```

Using a class

```python
class UnionFind:
    def __init__(self, N:int) -> None:
        self.d = list(range(N))

    def connected(self, a:int, b:int) -> bool:
        return self.d[a] == self.d[b]
```

Using a class

```python
from fastcore.basics import patch

@patch
def union(self:UnionFind, a:int, b:int) -> None:
    a_id = self.d[a]
    b_id = self.d[b]

    for ix, v in enumerate(self.d):
        if v == a_id:
            self.d[ix] = b_id
```

Adding a nice print

```python
@patch
def __str__(self:UnionFind) -> str:
    tmpd = {}
    for ix, v in enumerate(self.d):
        if v not in tmpd:
            tmpd[v] = []
        tmpd[v].append(ix)

    s = ''
    for k, v in tmpd.items():
        s += f'{{{",".join(map(str, v))}}} '
    return s
```

Example

```
uf = UnionFind(8)
uf.union(0, 1)
uf.union(1, 2)
uf.union(3, 4)
uf.union(4, 5)
uf.union(6, 7)

assert uf.connected(1, 2) == True
assert uf.connected(0, 5) == False
print(uf)
```

```
{0,1,2} {3,4,5} {6,7}
```

Good enough?

- How do we evaluate?
  - Correct? Yes
  - Speed?
  - Memory use?

Good enough?

```
uf = UnionFind(1_000_000)
%timeit uf.union(0, 1)
```

```
41.4 ms ± 940 µs per loop (mean ± std. dev. of 7 runs, 10
loops each)
```

Good enough?

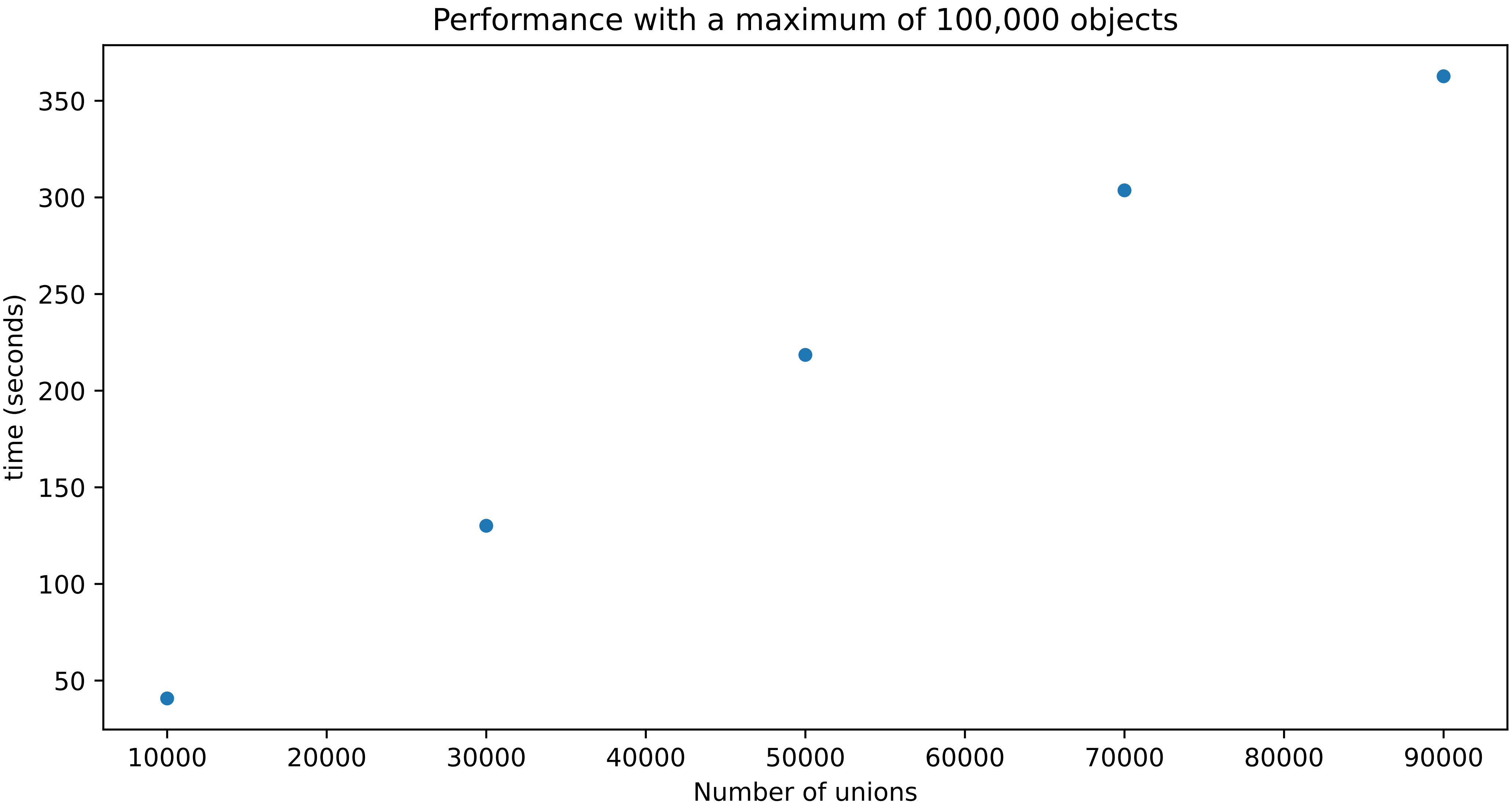
```
%timeit uf.connected(0,1)
```

```
115 ns ± 1.76 ns per loop (mean ± std. dev. of 7 runs,
10,000,000 loops each)
```

Why?

```python
# Fast
def connected(self, a:int, b:int) -> bool:
    return self.d[a] == self.d[b]

# Slower
def union(self:UnionFind, a:int, b:int) -> None:
    a_id = self.d[a]
    b_id = self.d[b]

    for ix, v in enumerate(self.d):
        if v == a_id:
            self.d[ix] = b_id
```

How bad?

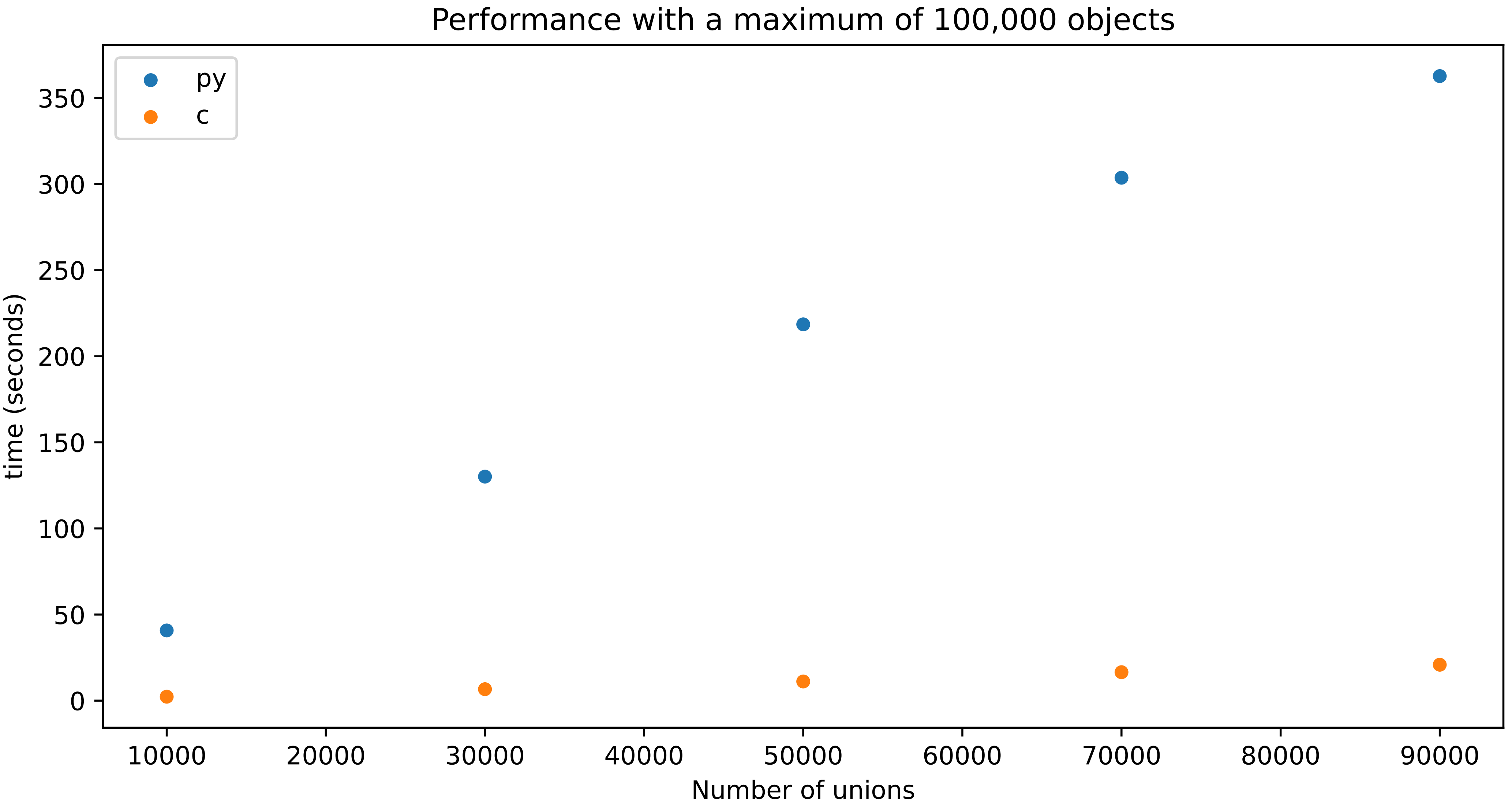


Performance with a maximum of 100,000 objects
time (seconds)
350
300
250
200
150
100
50
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of unions

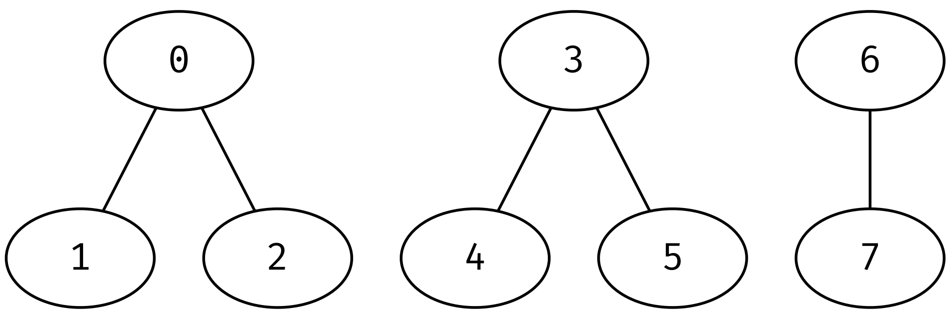
Python is slow?

```c
int *init(int n) {
    int *res = calloc(n, sizeof(int));
    for(int i=0; i<n; i++) res[i] = i;
    return res;
}

int connected(int *uf, int a, int b) {
    return (uf[a] == uf[b]);
}

void union_f(int *uf, int sz, int a, int b) {
    int id_a = uf[a];
    int id_b = uf[b];

    for(int i=0;i<sz;i++)
        if(uf[i] == id_a)
            uf[i] = id_b;
}
```

Python is slow?



Another approach (quick union)

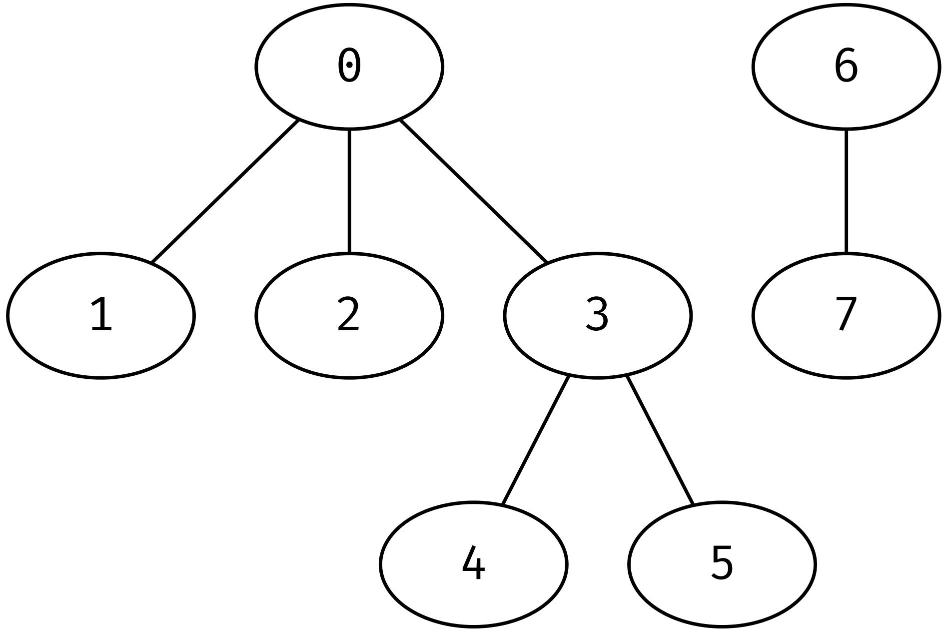


» Each component is represented as a tree

» Same root means same component

» When adding an object to a component, its root is connected to the root of the component

Another approach



» After `union(0, 3)`

Quick Union

```python
def init(N:int) -> list[int]:
    return list(range(N))

def connected(d:list[int], a:int, b:int) -> bool:
    return root(d, a) == root(d, b)

def union(d:list[int], a:int, b:int) -> None:
    ra = root(d, a)
    rb = root(d, b)
    d[ra] = rb
```

Quick Union

```python
def root(d:list[int], a:int) -> int:
    while a != d[a]:
        a = d[a]
    return a
```

Quick Union

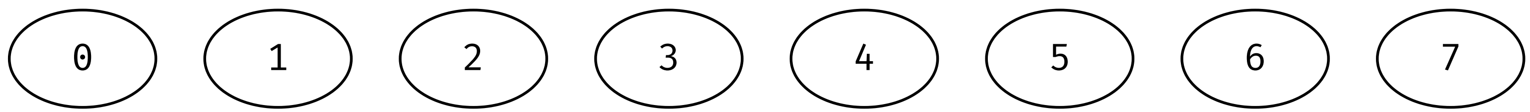
```python
uf = init(8)
union(uf, 0, 1)
union(uf, 1, 2)
union(uf, 3, 4)
union(uf, 4, 5)
union(uf, 6, 7)

assert connected(uf, 1, 2) == True
assert connected(uf, 0, 5) == False
```

How does it work?

```
uf = init(8)
print(uf)
union(uf, 0, 1)
union(uf, 1, 2)
print(uf)
union(uf, 3, 4)
print(uf)
```

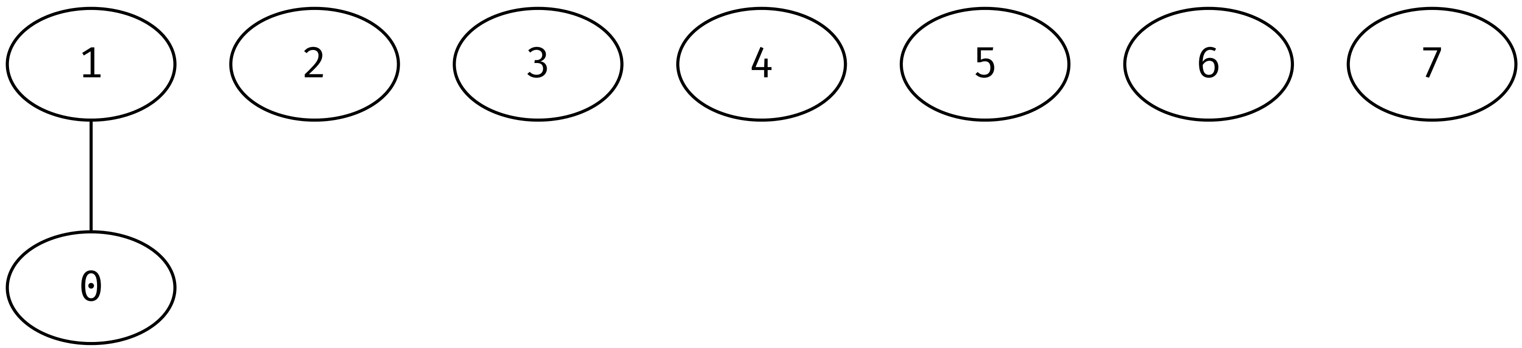
```
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 2, 2, 3, 4, 5, 6, 7]
[1, 2, 2, 4, 4, 5, 6, 7]
```

How does it work?



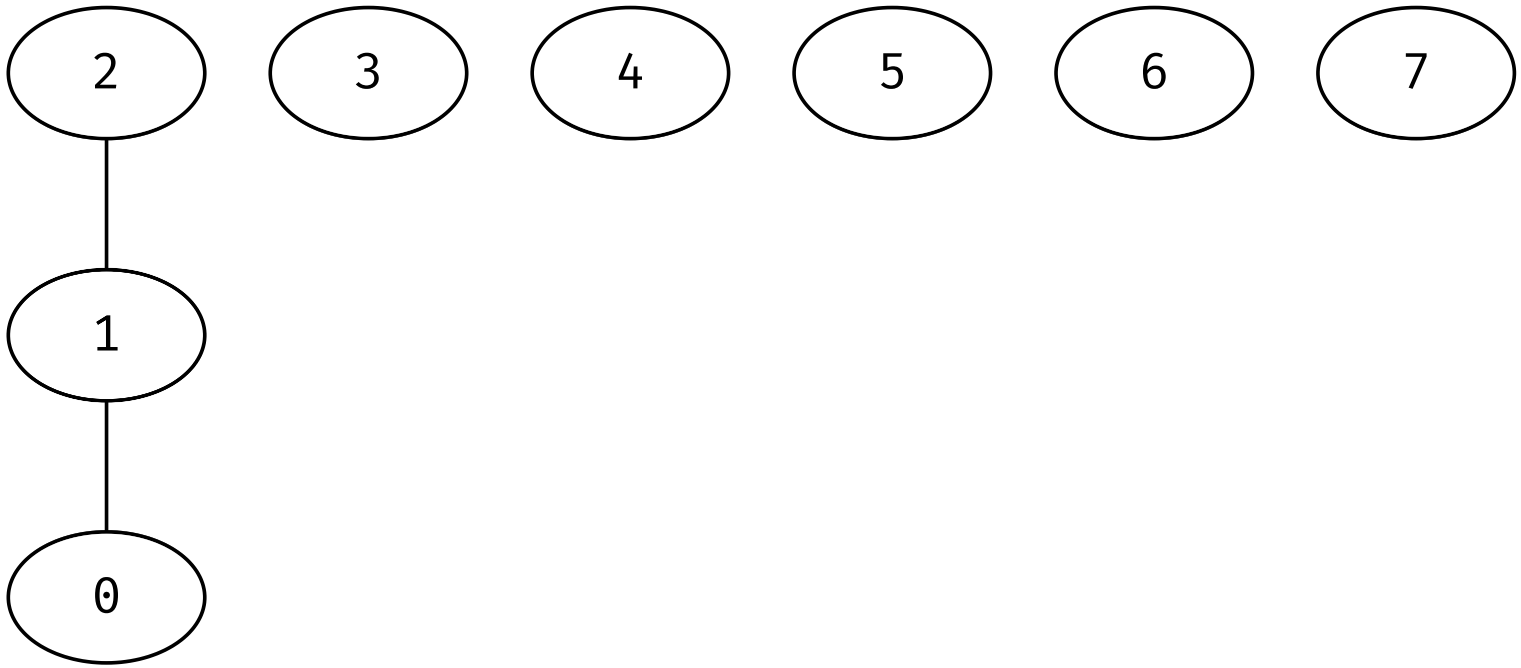
`init(8)`

How does it work?



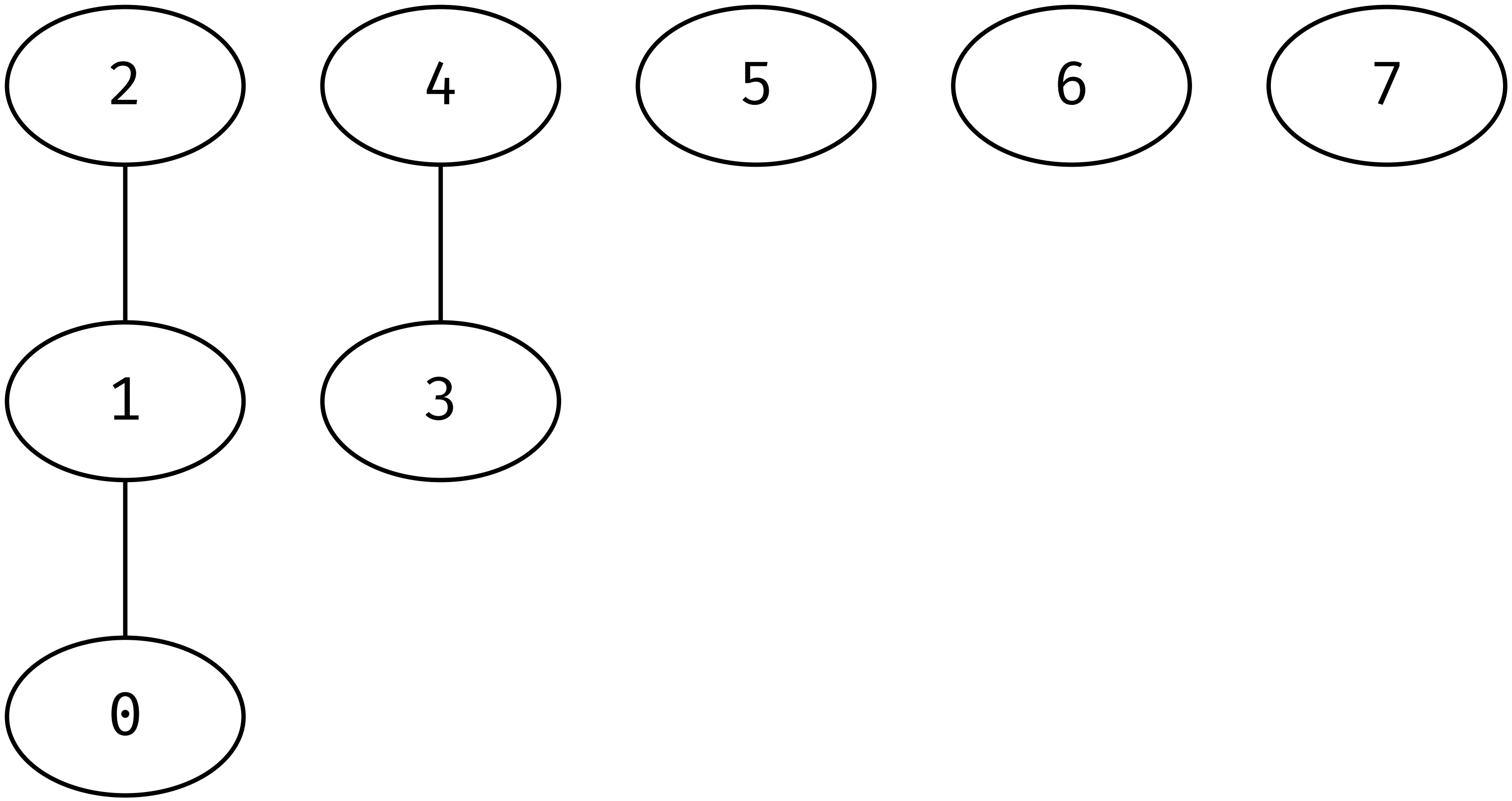
`init(8),union(0, 1)`

How does it work?



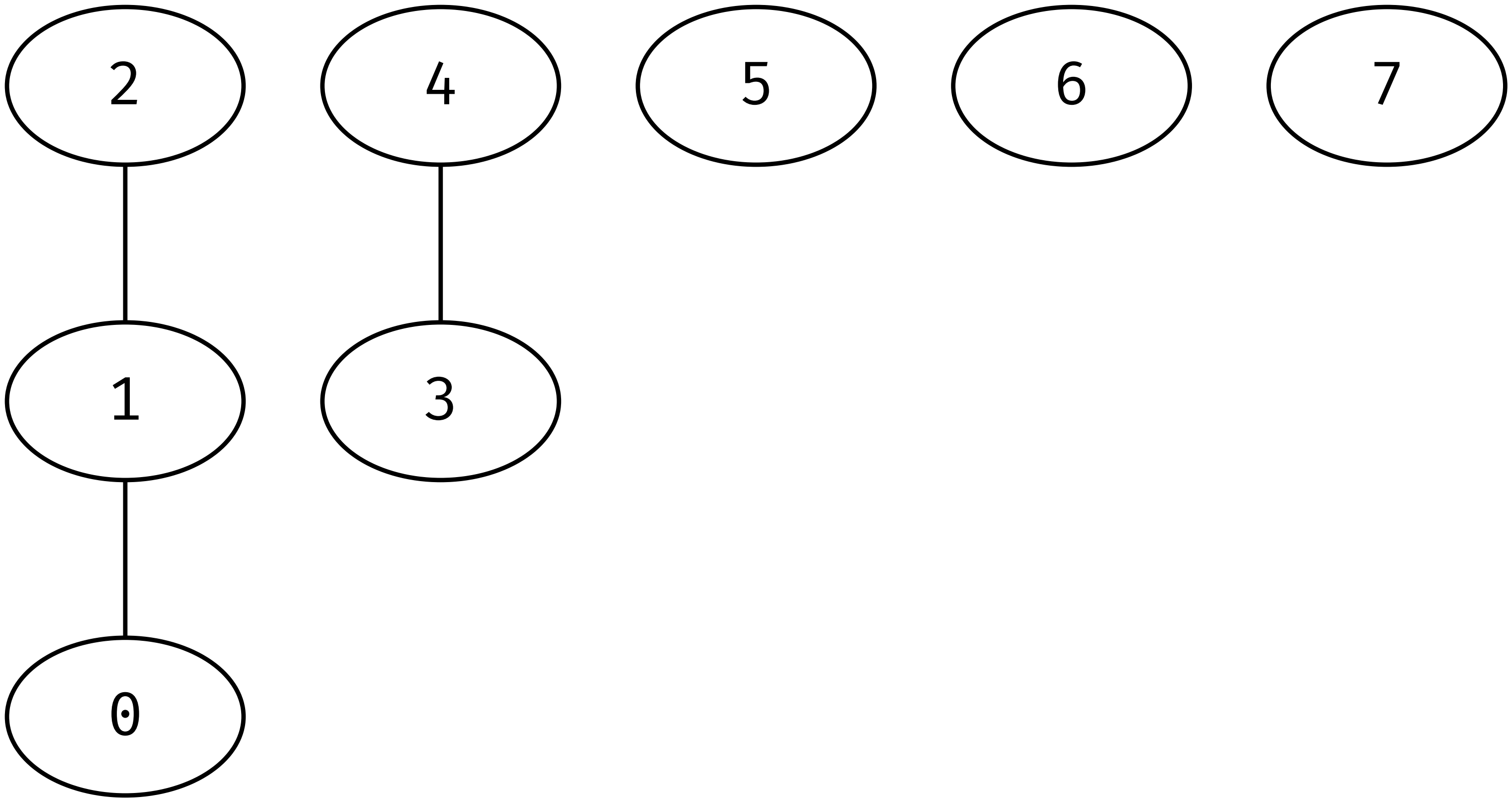
`init(8)`, `union(0, 1)`, `union(1, 2)`

How does it work?



`init(8),union(0, 1),union(1, 2),union(3, 4)`

How does it work?



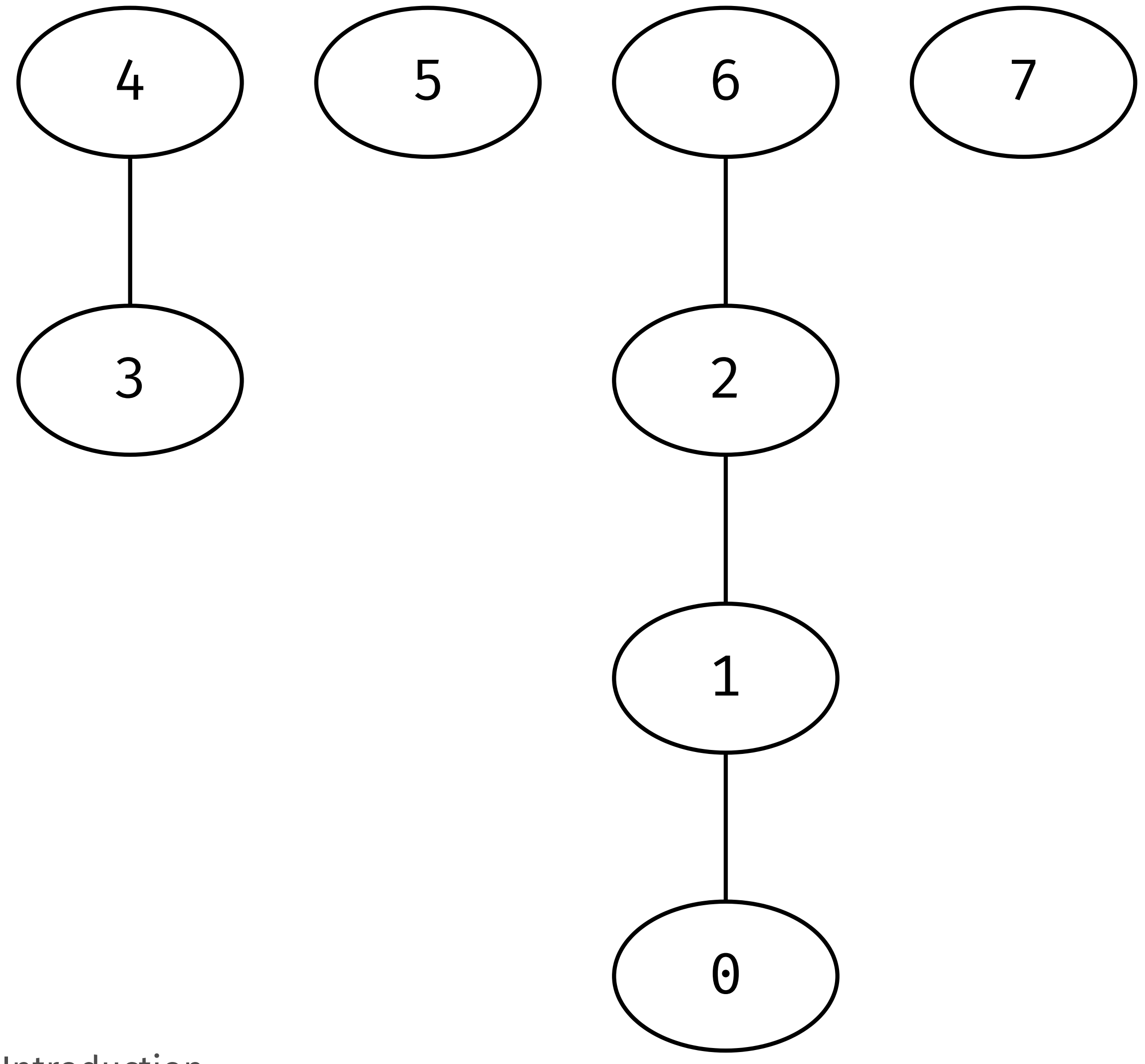
`connected(1, 3)`? No.

How does it work?

```python
# uf = [1, 2, 2, 4, 4, 5, 6, 7]
union(uf, 0, 6)
print(uf)
```

```
[1, 2, 6, 4, 4, 5, 6, 7]
```

How does it work?



Moving to a class

```python
class QUnionFind:
    def __init__(self, N:int) -> None:
        self.d = list(range(N))

    def connected(self, a:int, b:int) -> bool:
        return self.root(a) == self.root(b)

    def union(self, a:int, b:int) -> None:
        ra = self.root(a)
        rb = self.root(b)
        self.d[ra] = rb
```

Moving to a class

```python
@patch
def root(self:QUnionFind, a:int) -> int:
    while a != self.d[a]:
        a = self.d[a]
    return a
```

Better?

```
quf = QUnionFind(1_000_000)
%timeit quf.union(0, 1)
```

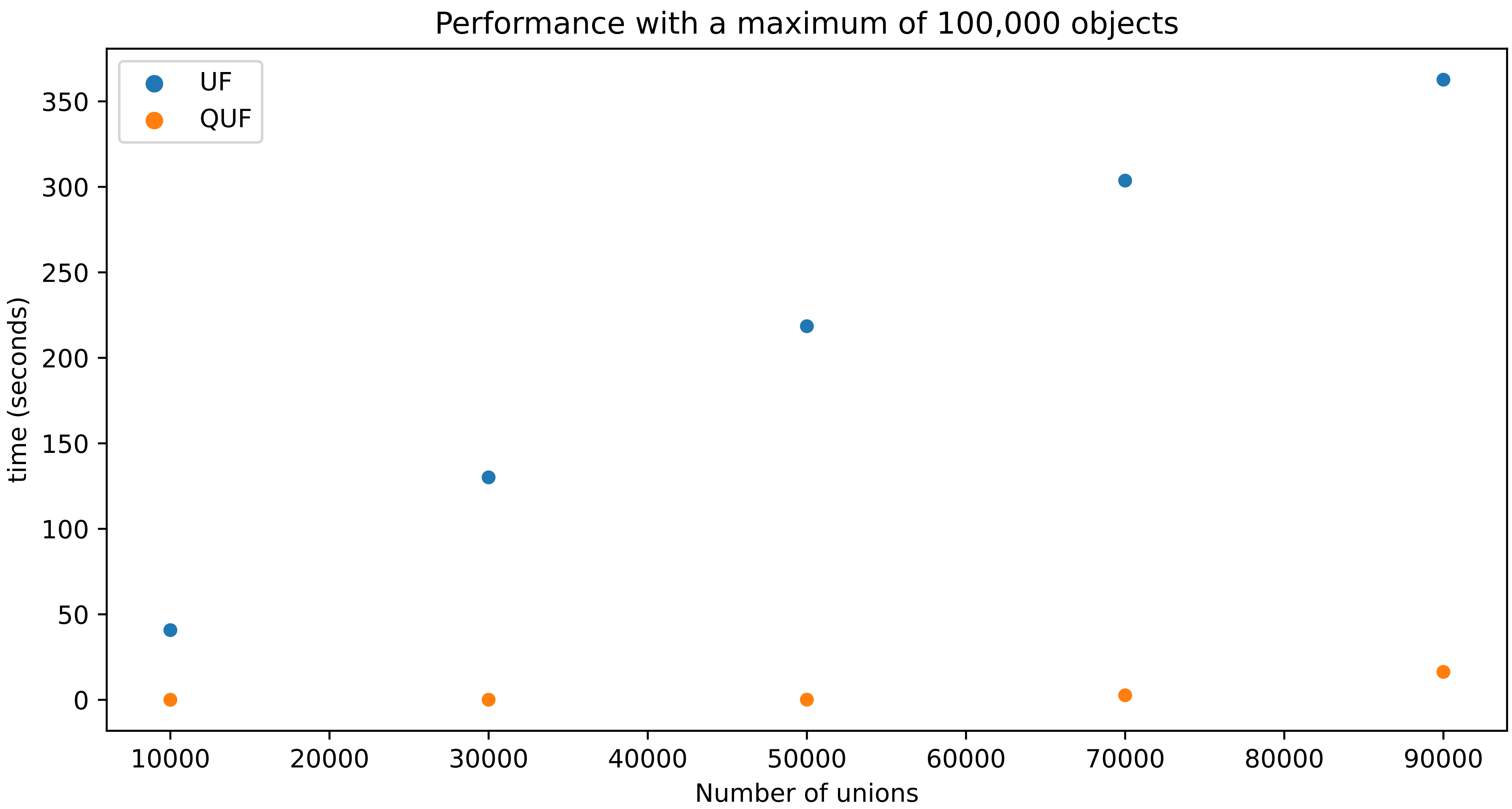
```
345 ns ± 7.13 ns per loop (mean ± std. dev. of 7 runs,
1,000,000 loops each)
```

Better?

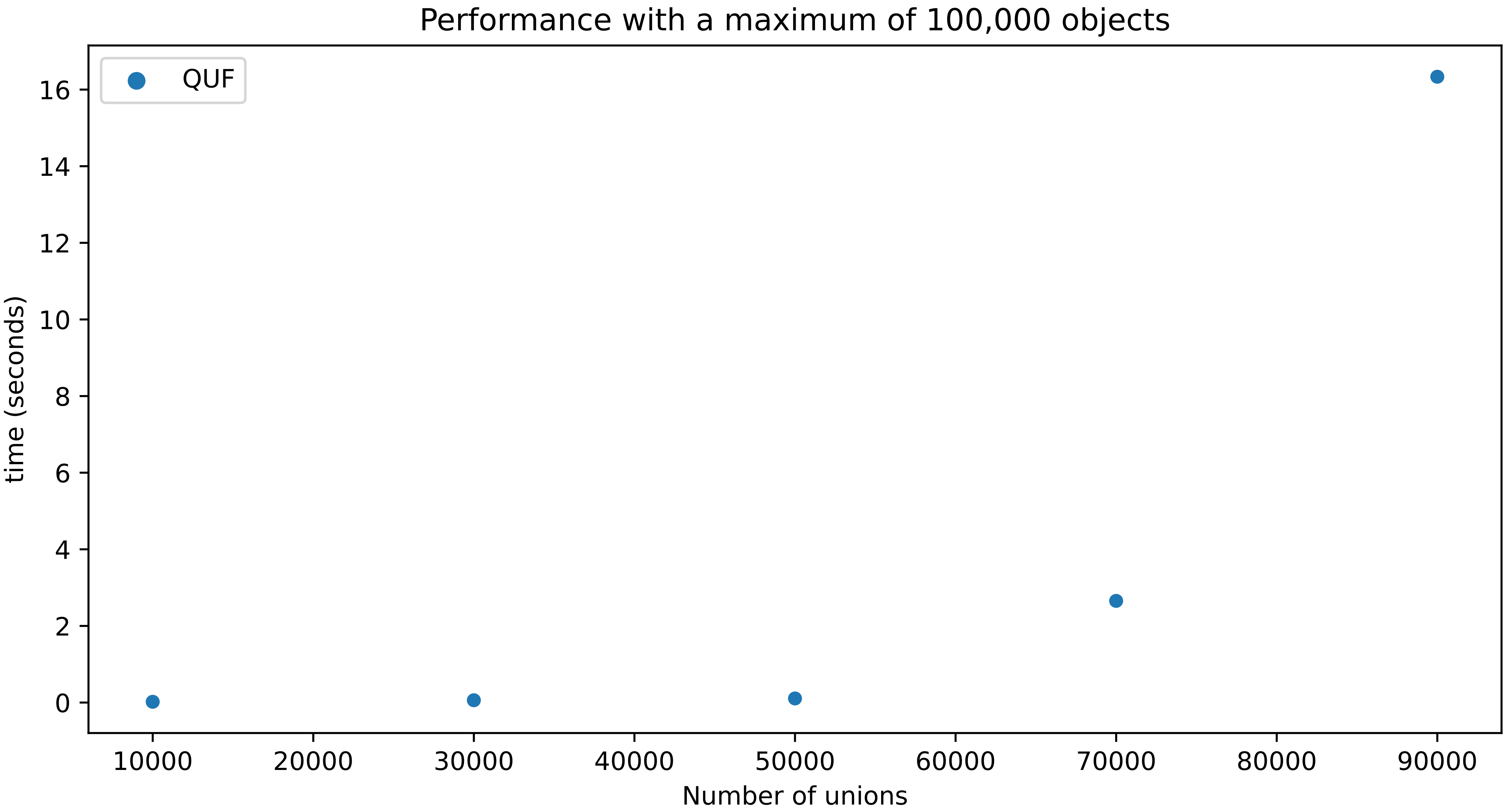
```
%timeit quf.connected(0,1)
```

```
328 ns ± 4.93 ns per loop (mean ± std. dev. of 7 runs,
1,000,000 loops each)
```

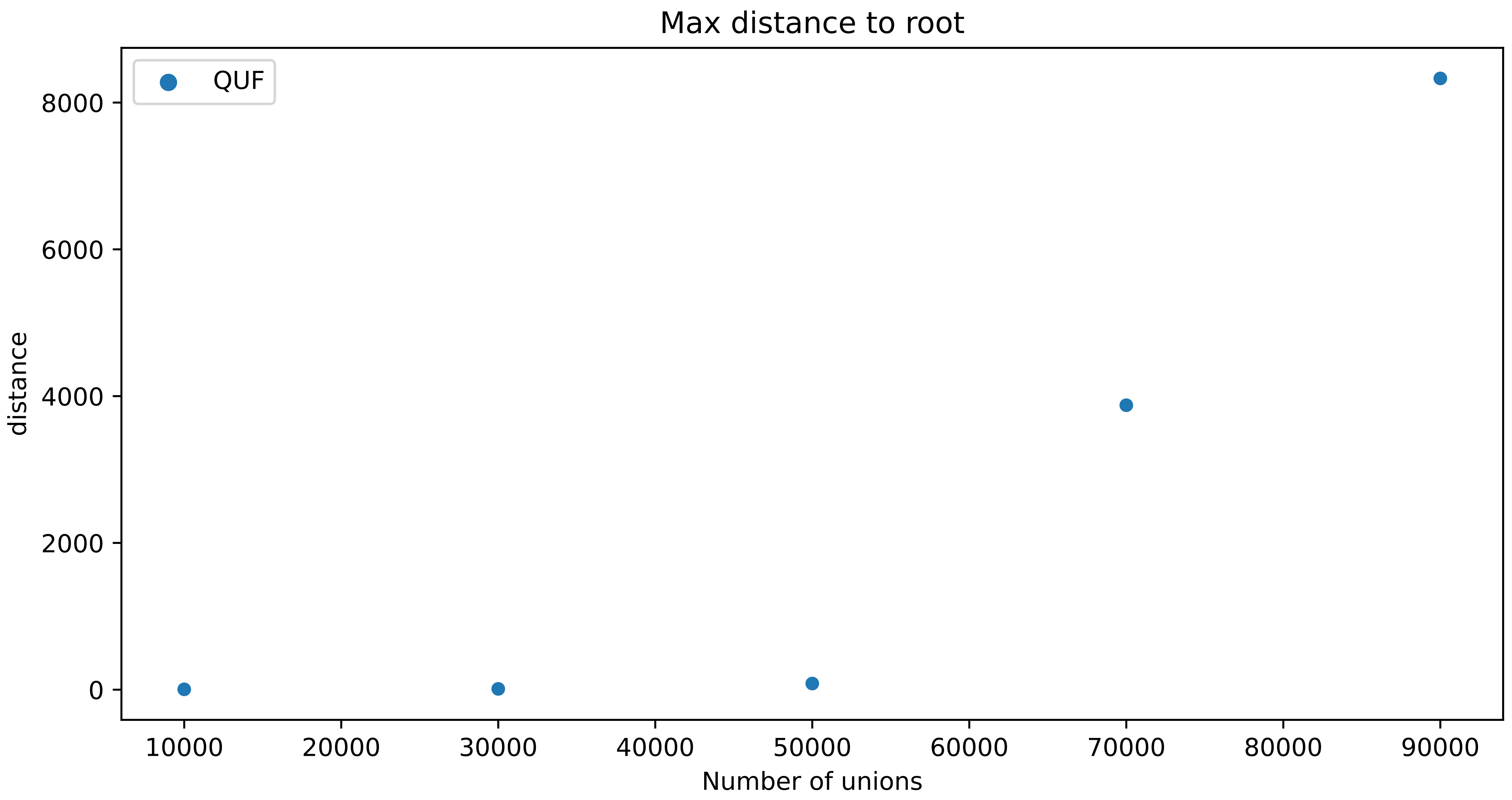
Compared to the first attempt



Looks good?


Performance with a maximum of 100,000 objects
QUF
time (seconds)
0
2
4
6
8
10
12
14
16
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of unions

Why the quick increase in time?



Can we do better?

```python
class WQUnionFind:
    def __init__(self, N:int) -> None:
        self.d = list(range(N))
        self.sz = [1]*N

    def connected(self, a:int, b:int) -> bool:
        return self.root(a) == self.root(b)

    def root(self, a:int) -> int:
        while a != self.d[a]:
            a = self.d[a]
        return a
```

Can we do better?

```python
@patch
def union(self:WQUnionFind, a:int, b:int) -> None:
    ra = self.root(a)
    rb = self.root(b)

    if self.sz[ra] < self.sz[rb]:
        self.d[ra] = rb
        self.sz[rb] += self.sz[ra]
    else:
        self.d[rb] = ra
        self.sz[ra] += self.sz[rb]
```

Can we do better?

```
wquf = WQUnionFind(8)
print(wquf.sz, wquf.d)
wquf.union(0, 1)
print(wquf.sz, wquf.d)
wquf.union(1, 2)
print(wquf.sz, wquf.d)
wquf.union(3, 4)
print(wquf.sz, wquf.d)
```

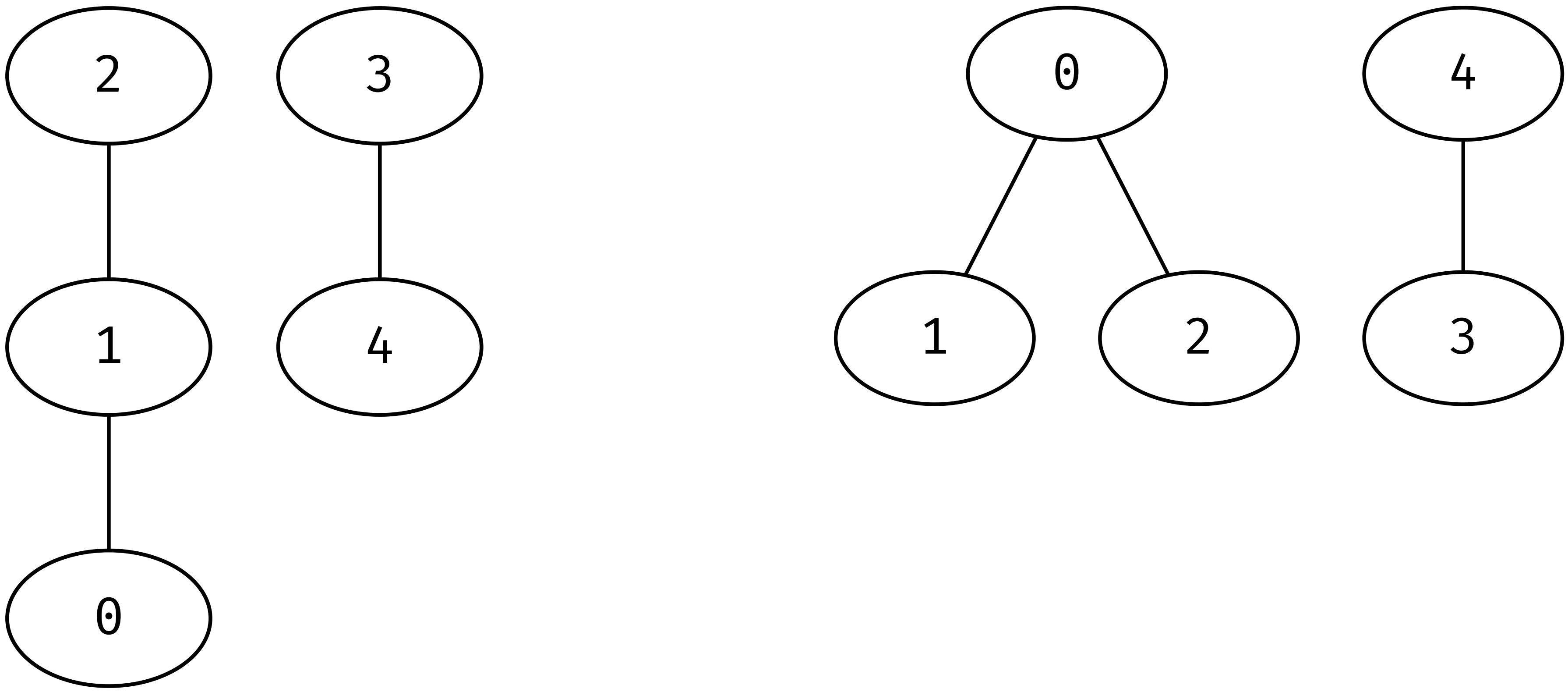
```
[1, 1, 1, 1, 1, 1, 1, 1] [0, 1, 2, 3, 4, 5, 6, 7]
[2, 1, 1, 1, 1, 1, 1, 1] [0, 0, 2, 3, 4, 5, 6, 7]
[3, 1, 1, 1, 1, 1, 1, 1] [0, 0, 0, 3, 4, 5, 6, 7]
[3, 1, 1, 2, 1, 1, 1, 1] [0, 0, 0, 3, 3, 5, 6, 7]
```

Spot the difference

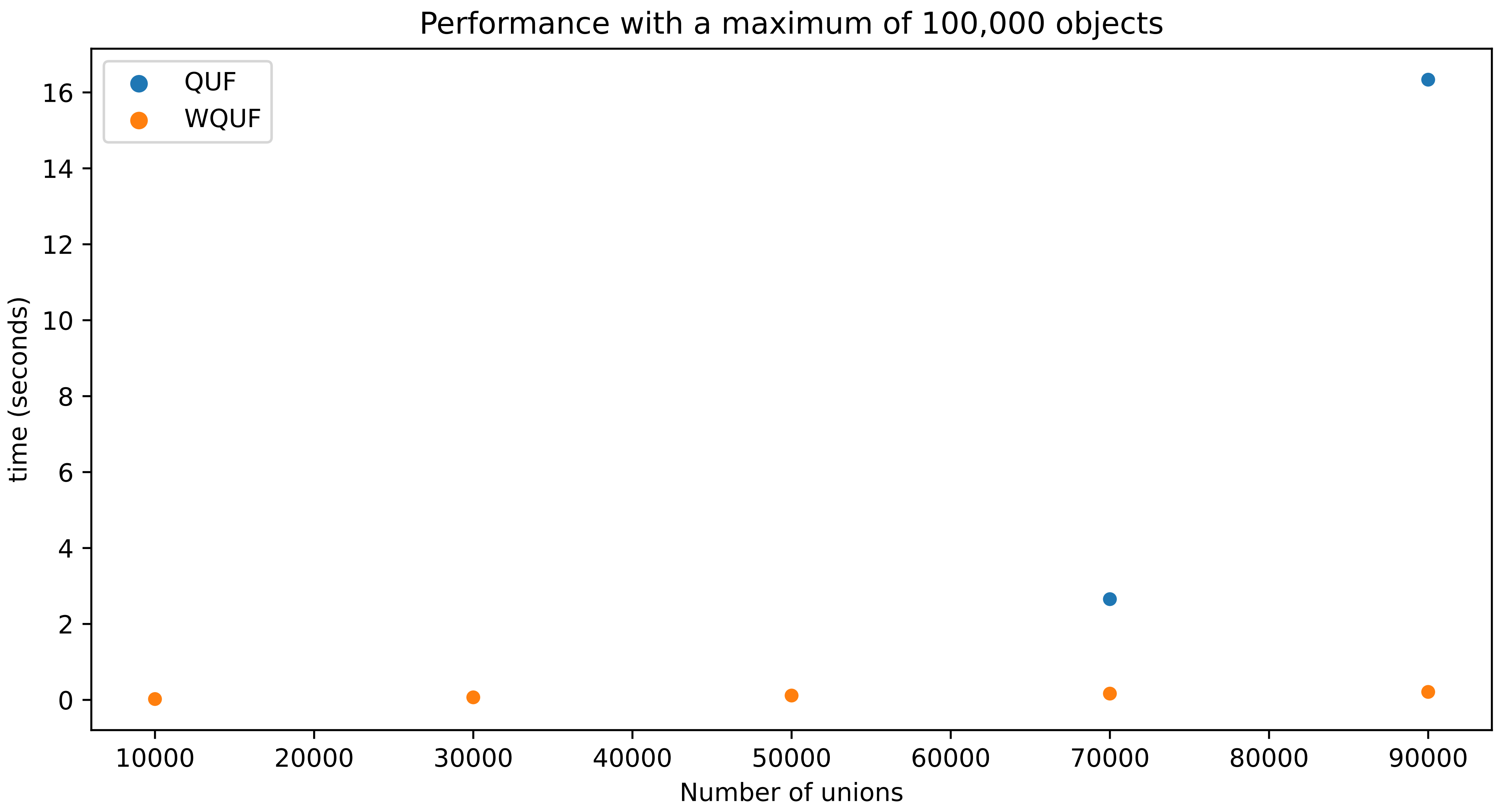
```
quf = QUnionFind(8)
quf.union(0, 1)
quf.union(1, 2)
quf.union(3, 4)

wquf = WQUnionFind(8)
wquf.union(0, 1)
wquf.union(1, 2)
wquf.union(3, 4)

print(quf.d)
print(wquf.d)
```

```
[1, 2, 2, 4, 4, 5, 6, 7]
[0, 0, 0, 3, 3, 5, 6, 7]
```

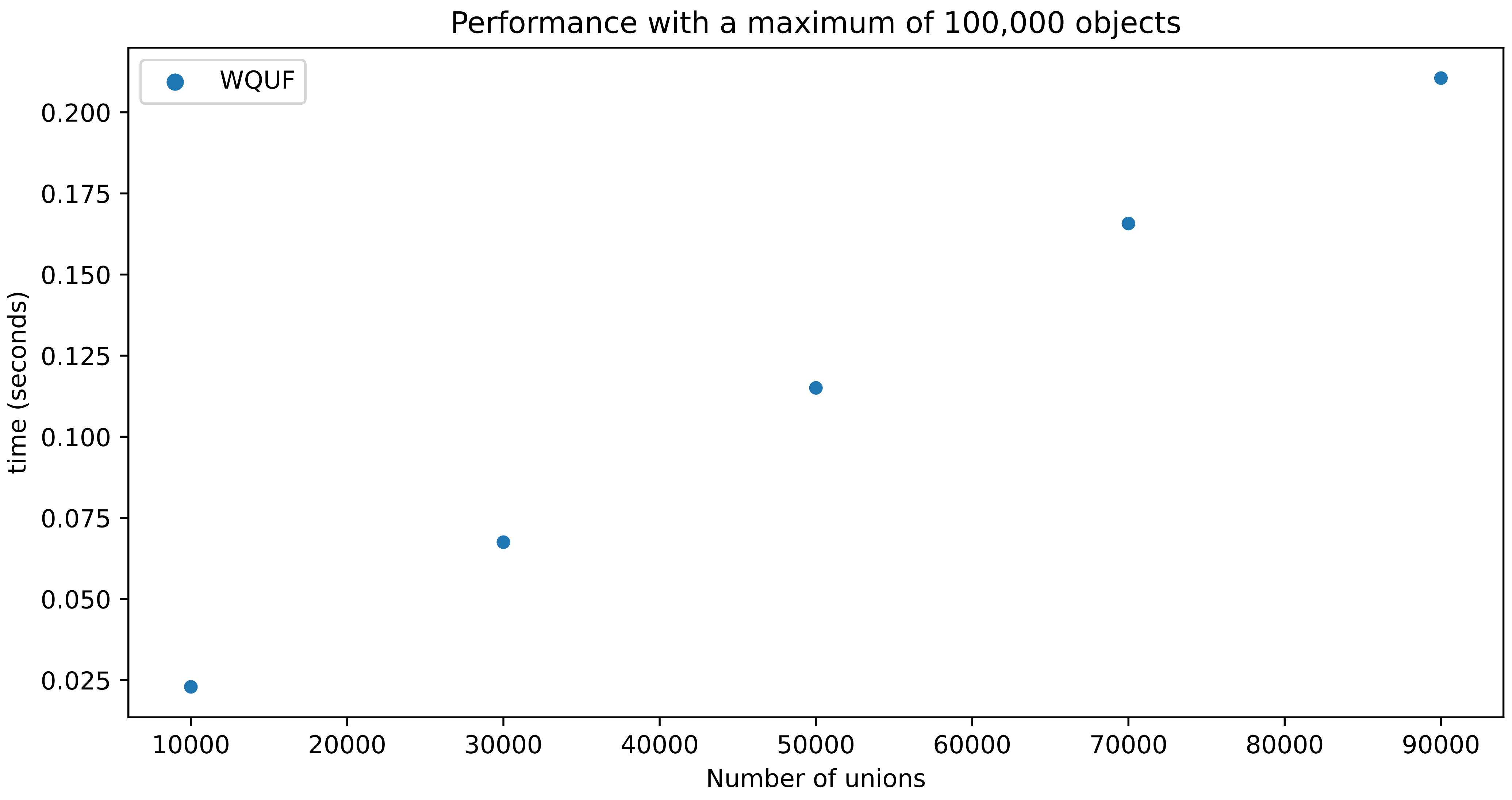
Spot the difference



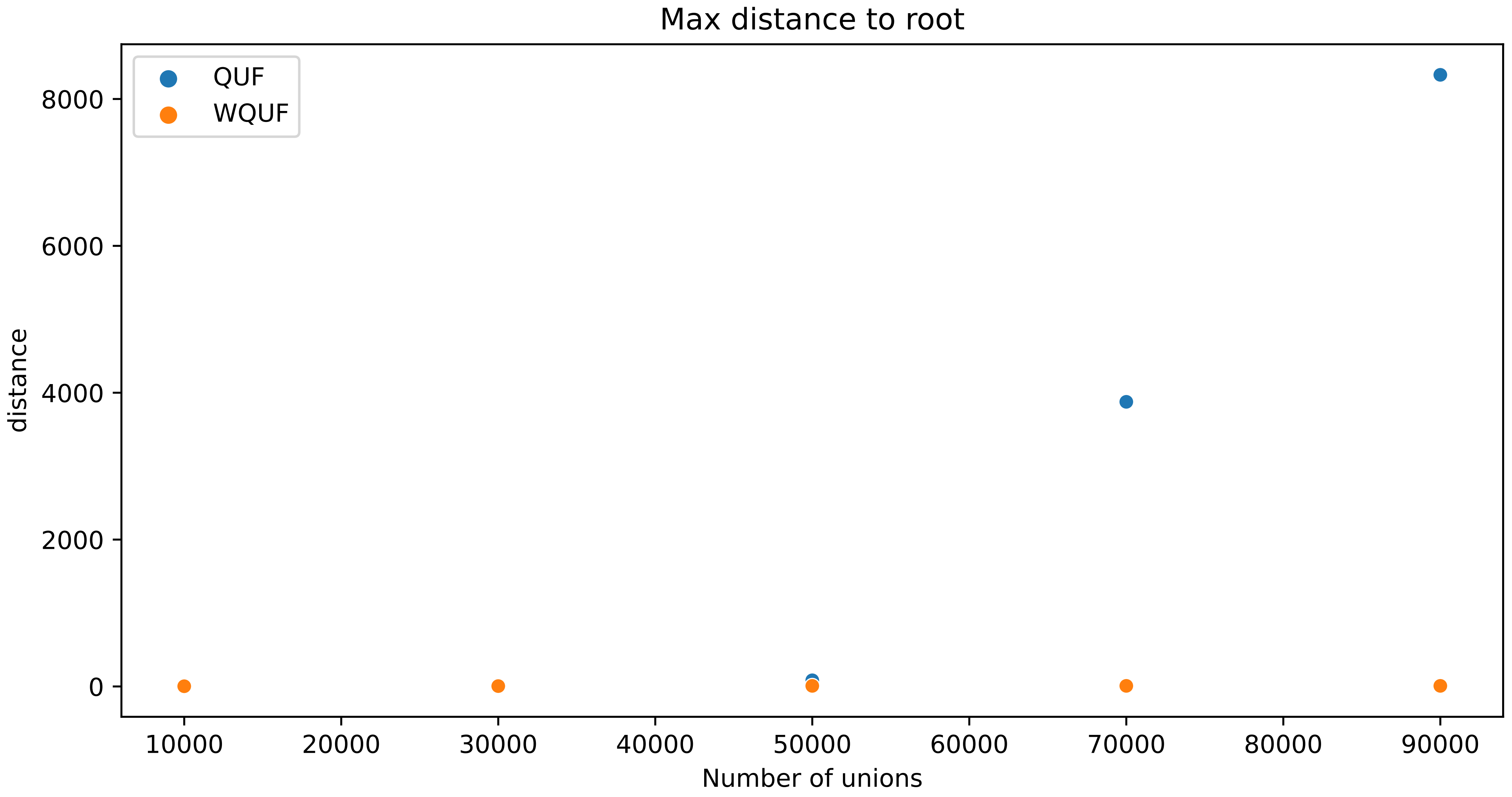
Performance compared to previous



Nicer growth

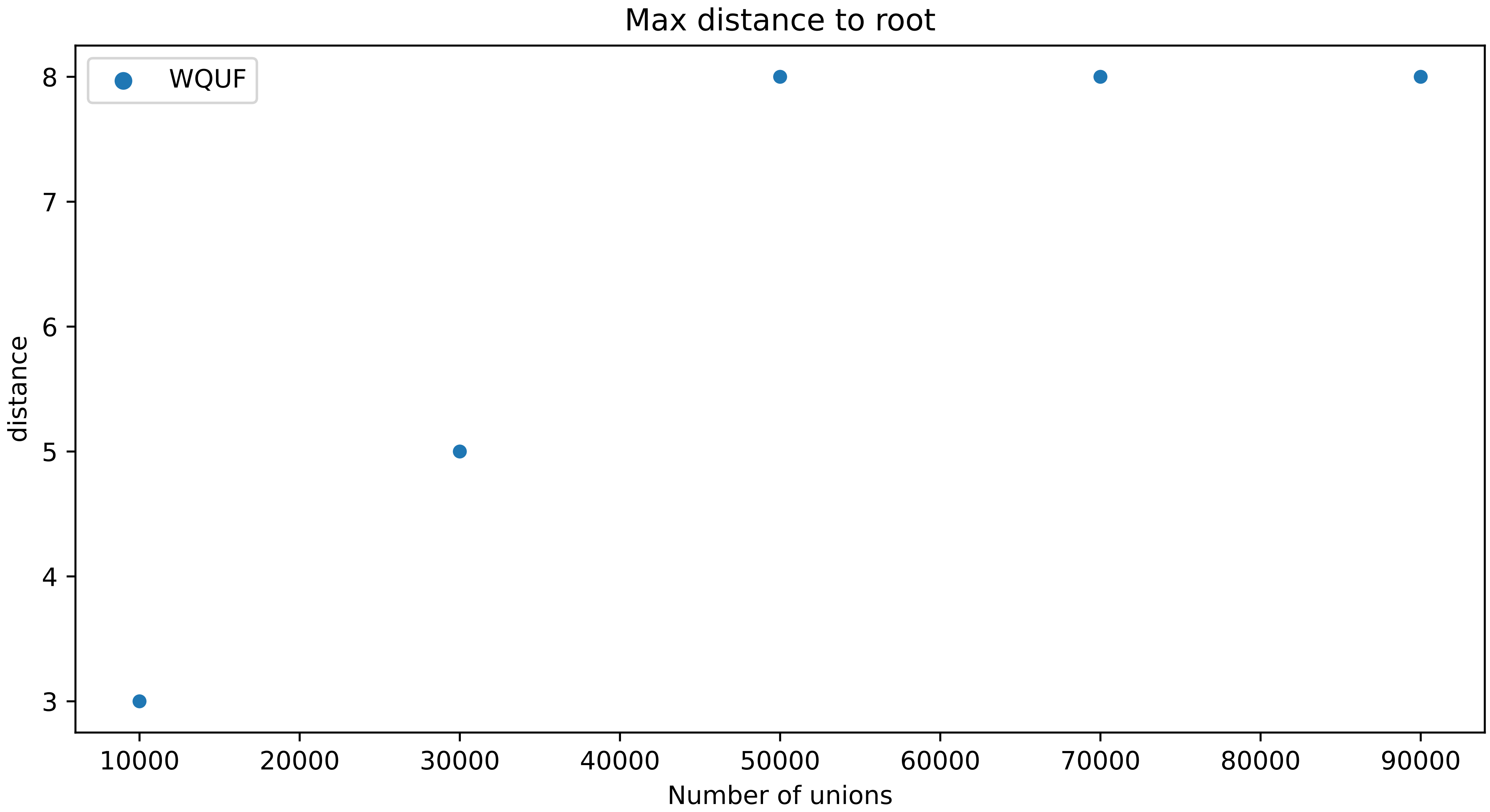


Tree depth

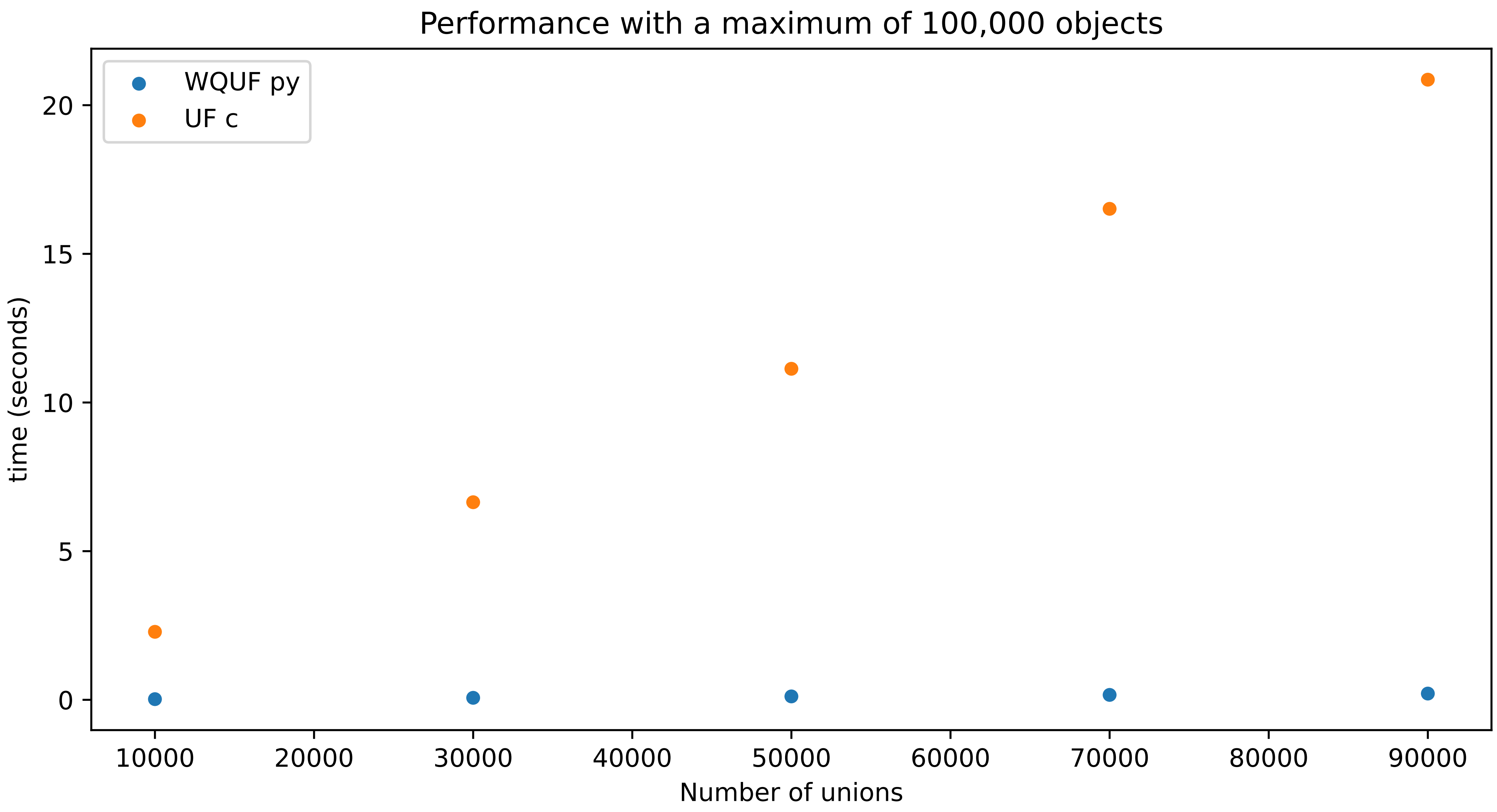


Max distance to root
QUF
WQUF
8000
6000
4000
2000
0
distance
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of unions

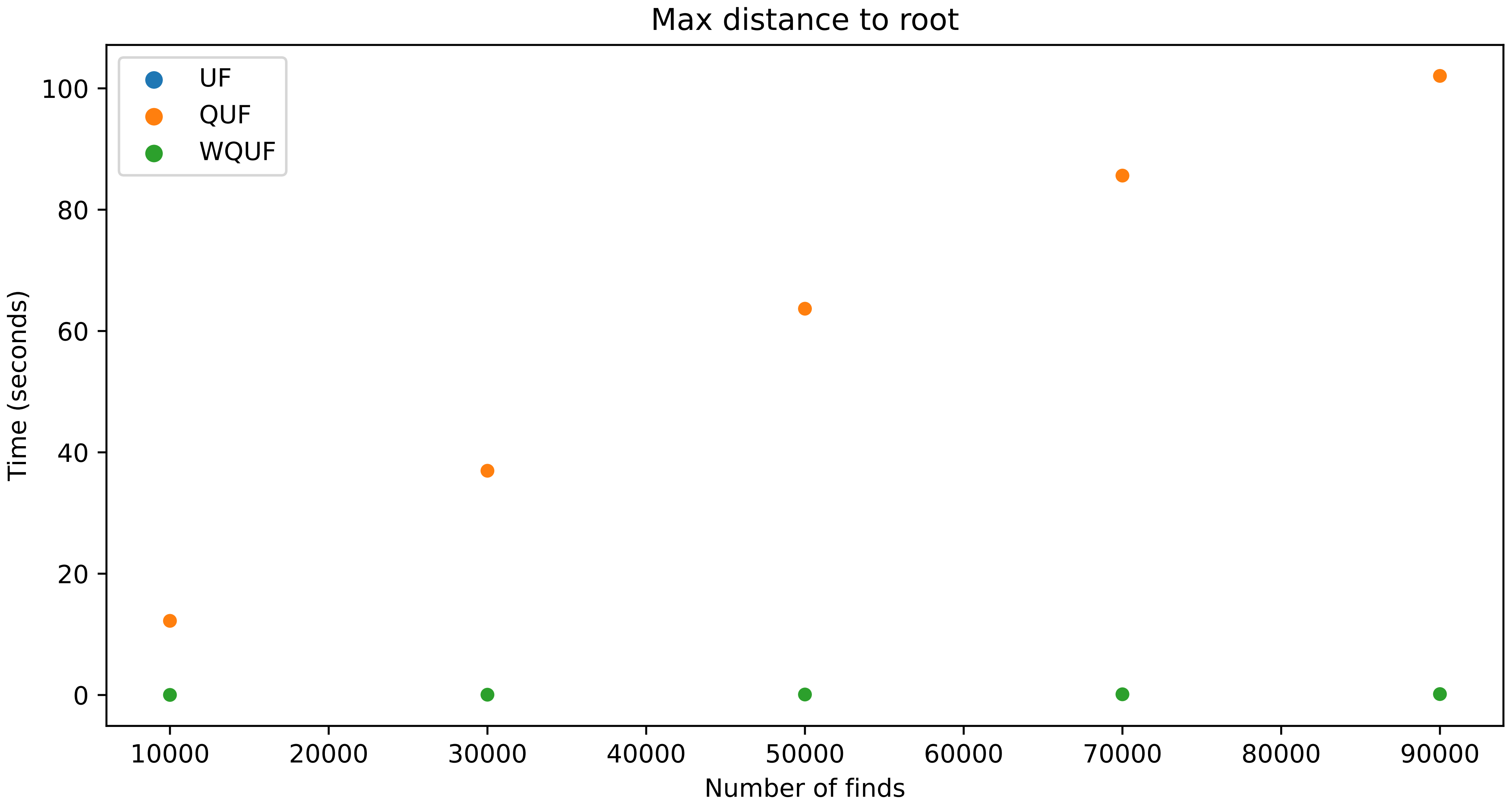
Much shorter distances



Python is slow?



Drawbacks?

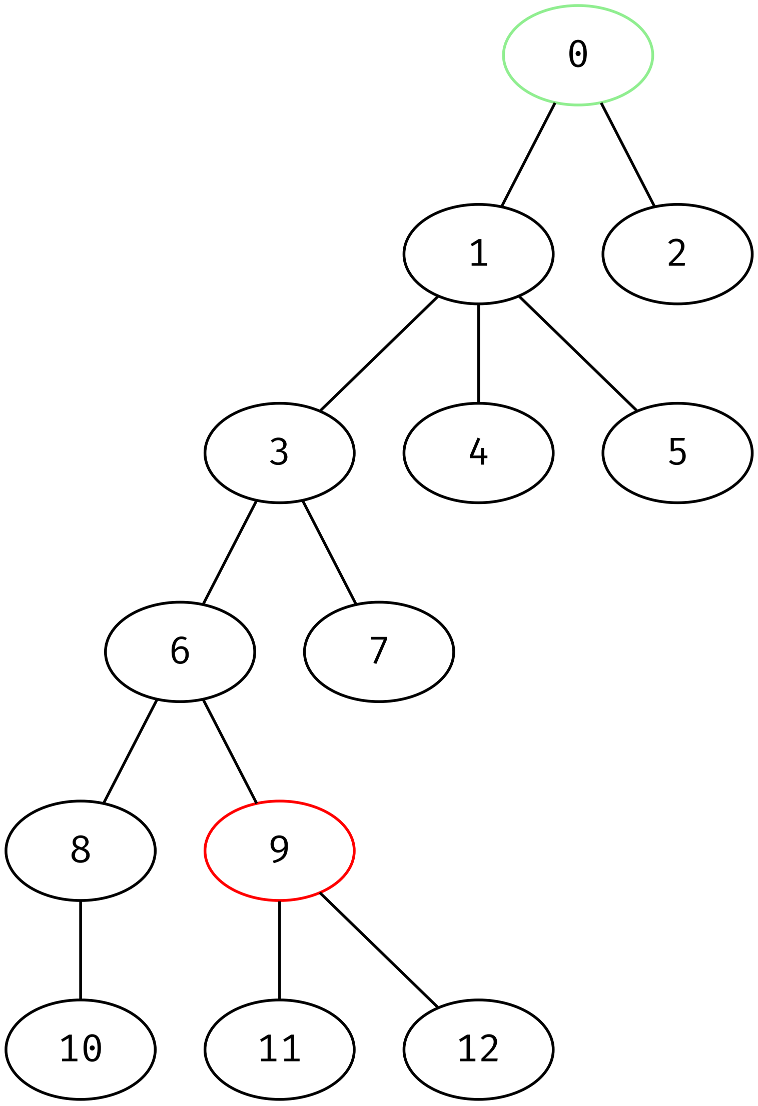


Max distance to root
UF
QUF
WQUF
100
80
60
40
20
0
Time (seconds)
10000
20000
30000
40000
50000
60000
70000
80000
90000
Number of finds

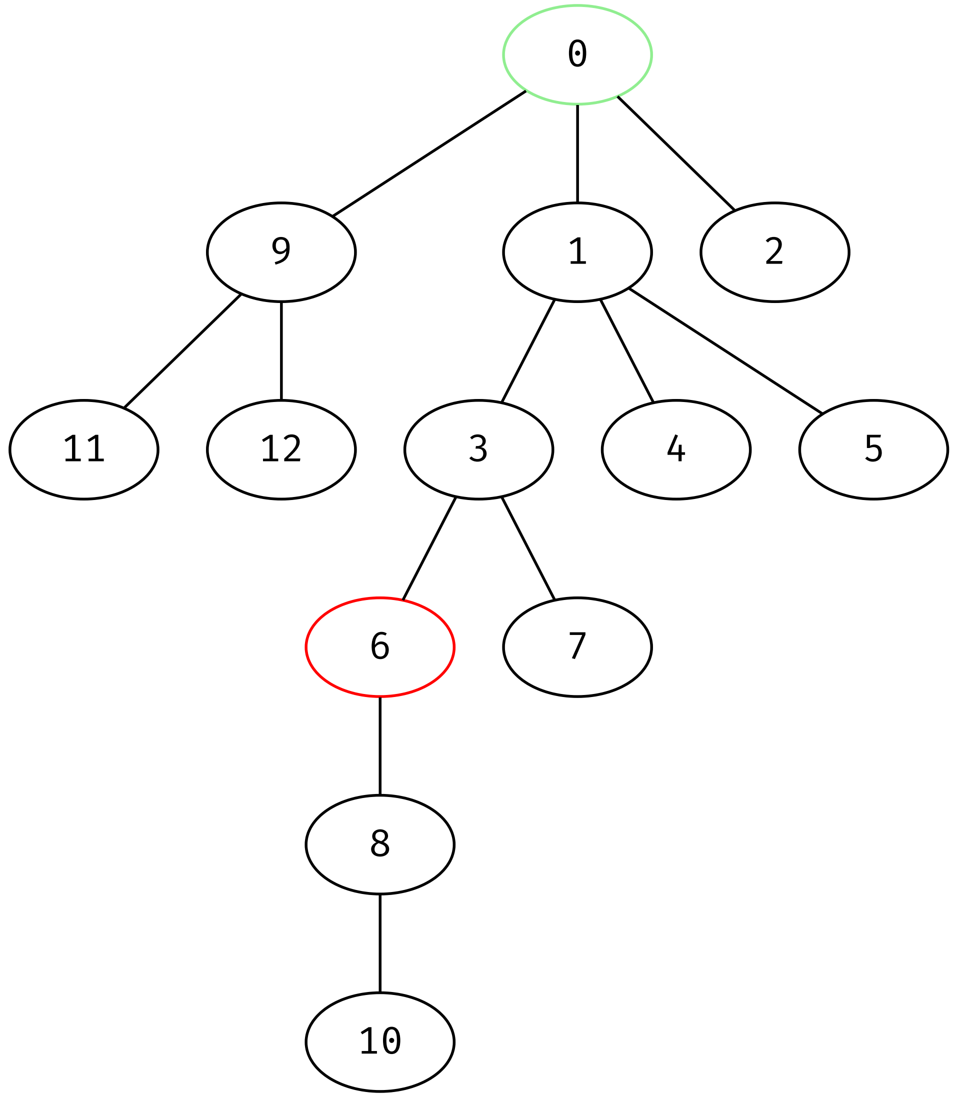
Path compression

- The time to find the root depends on the height of the tree
- Merging to smaller helped
- What if we can make the trees even “flatter” (but wider)
- Idea: move subtrees when we are looking for the root?

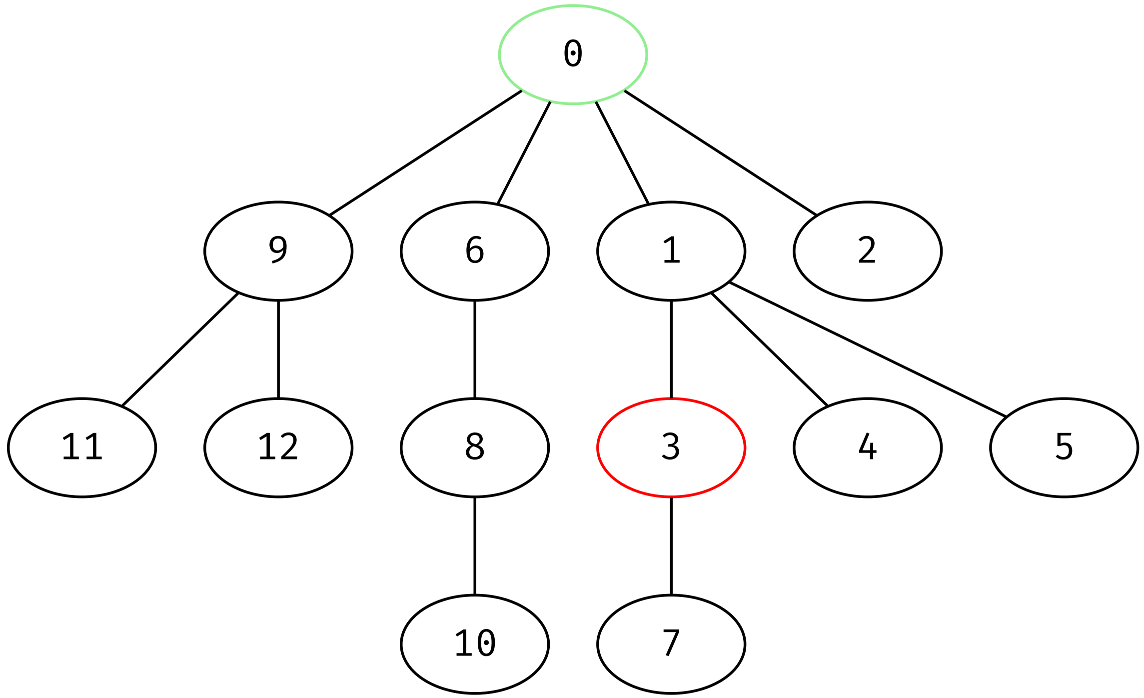
Path compression



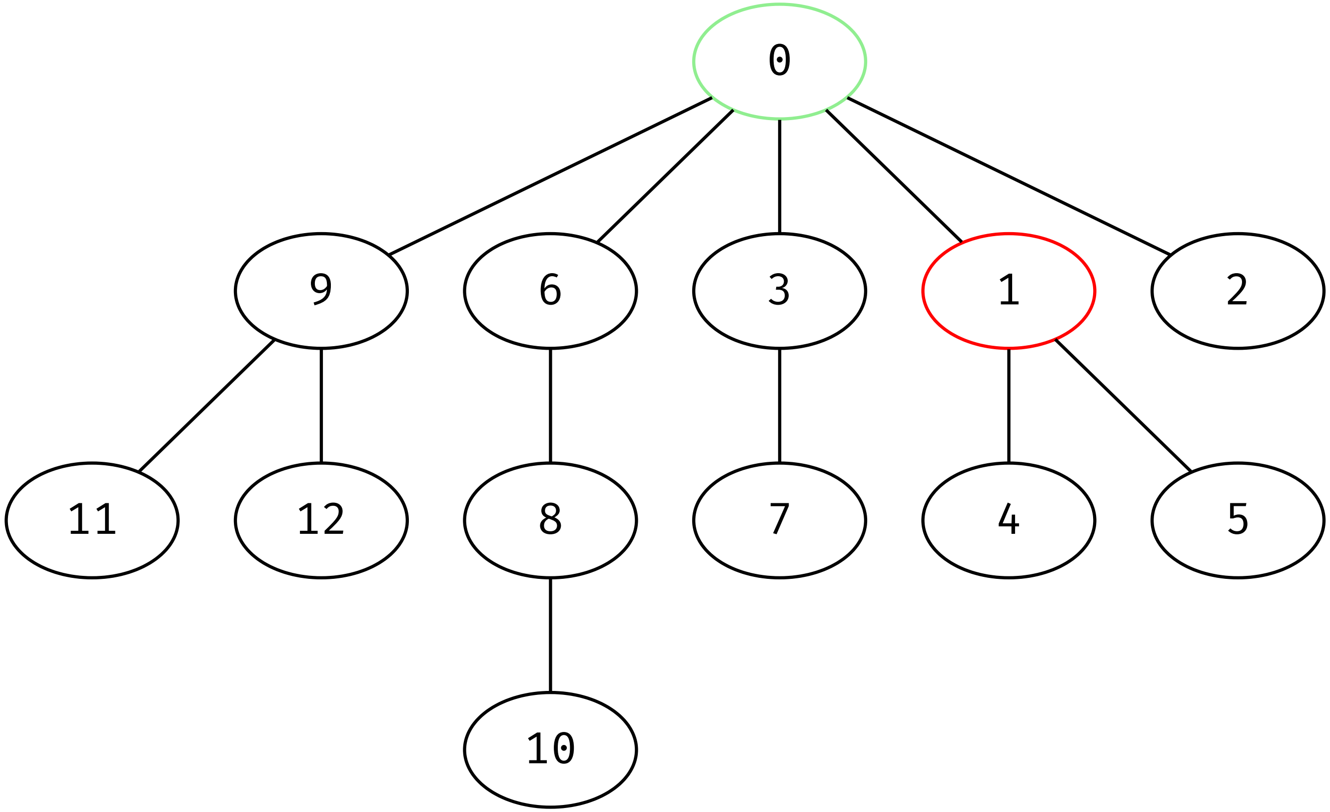
Path compression



Path compression



Path compression



Simple change

```python
@patch
def root(self:WQUnionFind, a:int) -> int:
    while a != self.d[a]:
        self.d[a] = self.d[self.d[a]]
        a = self.d[a]
    return a
```

Reading instructions

Reading instructions

» Ch. 1 (you should know most of this already)

» Ch. 8.1 - 8.5, 8.7 (we will cover 8.6 later)