Analysis of Algorithms, I

CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Thursday, April 2, 2015

Outline

Today

1. Recap: Kruskal's algorithm for MSTs
2. A union-find data structure for disjoint sets
3. Application: Clustering

Recap

- Minimum Spanning Trees (MST)
- The Cut Property and greedy algorithms for MSTs
  - Prim's algorithm
  - Kruskal's algorithm
  - Counting the #MSTs in K_n

Kruskal's algorithm: detailed description

Short description: at every step, add to A the lightest edge that does not create a cycle with the edges already in A

Alternative view of the algorithm: let $T(v)$ be the tree where vertex v belongs; initially, every vertex forms its own tree.

1. Initialize $A = \emptyset$
2. `Sort` the edges by increasing weight
3. For every edge $e = (u, v)$ in **increasing order**:
   - If u and v belong to the same tree, discard e
   - Else $A = A \cup \{e\}$; merge $T(u)$, $T(v)$ into a single tree

Implementing Kruskal's algorithm

Need a data structure that maintains a **collection of disjoint sets** and allows

1. to check if u, v belong to the same tree;
2. for updates to reflect the merging of two trees into one

Operations:

1. `MakeSet`(u): Given an element u, create a new tree containing only u. Target worst-case time: $O(1)$
2. `Find`(u): Given an element u, find which tree u belongs to. Target worst-case time: $O(\log n)$
3. `Union`(u, v): Merge the tree containing u and the tree containing v into a single tree. Target worst-case time: $O(\log n)$

Pseudocode

```
Kruskal(G = (V, E, w))
    A = ∅
    Sort(E) by w
    for u ∈ V do MakeSet(u)
    end for
    for (u, v) ∈ E by increasing w do
        if Find(u) ≠ Find(v) then
            A = A ∪ {(u, v)}
            Union(u, v)
        end if
    end for
```

Running time: $O((n+m)\log n)$

Today

1. Recap: Kruskal's algorithm for MSTs
2. A union-find data structure for disjoint sets
3. Application: Clustering

A tree data structure

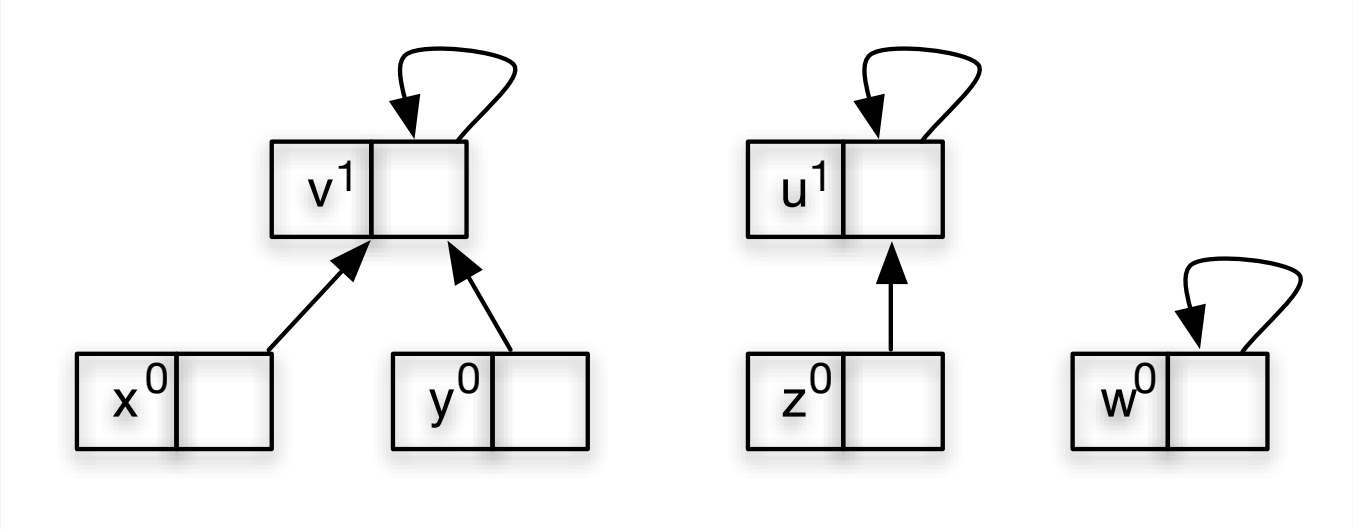
Store a set of elements as a **directed tree**.

- Nodes correspond to set elements (no particular order)
- Each node has a *parent* pointer
- If the root of the tree is element r, then the set is assigned the *name* r.
  - the root's *parent* pointer is a self-loop
- Every node has a *rank*

$$rank(u) = \text{height of } u\text{'s subtree}$$

A set represented as a forest of directed trees

A set of 6 elements maintained by 3 disjoint sets. For example, elements $\{x, y\}$ belong to tree v, while z belongs to tree u. The superscript next to each node's name is its rank.



Operations `MakeSet`(u), `Find`(u)

`MakeSet`(u)

$\pi(u) = u$

$rank(u) = 0$

`Find`(u) //returns the *name* of the set where u belongs

while $\pi(u) \neq u$ **do**

$u = \pi(u)$

end while

`return` u

Running time?

Operation `Union`(u, v) constructs the tree

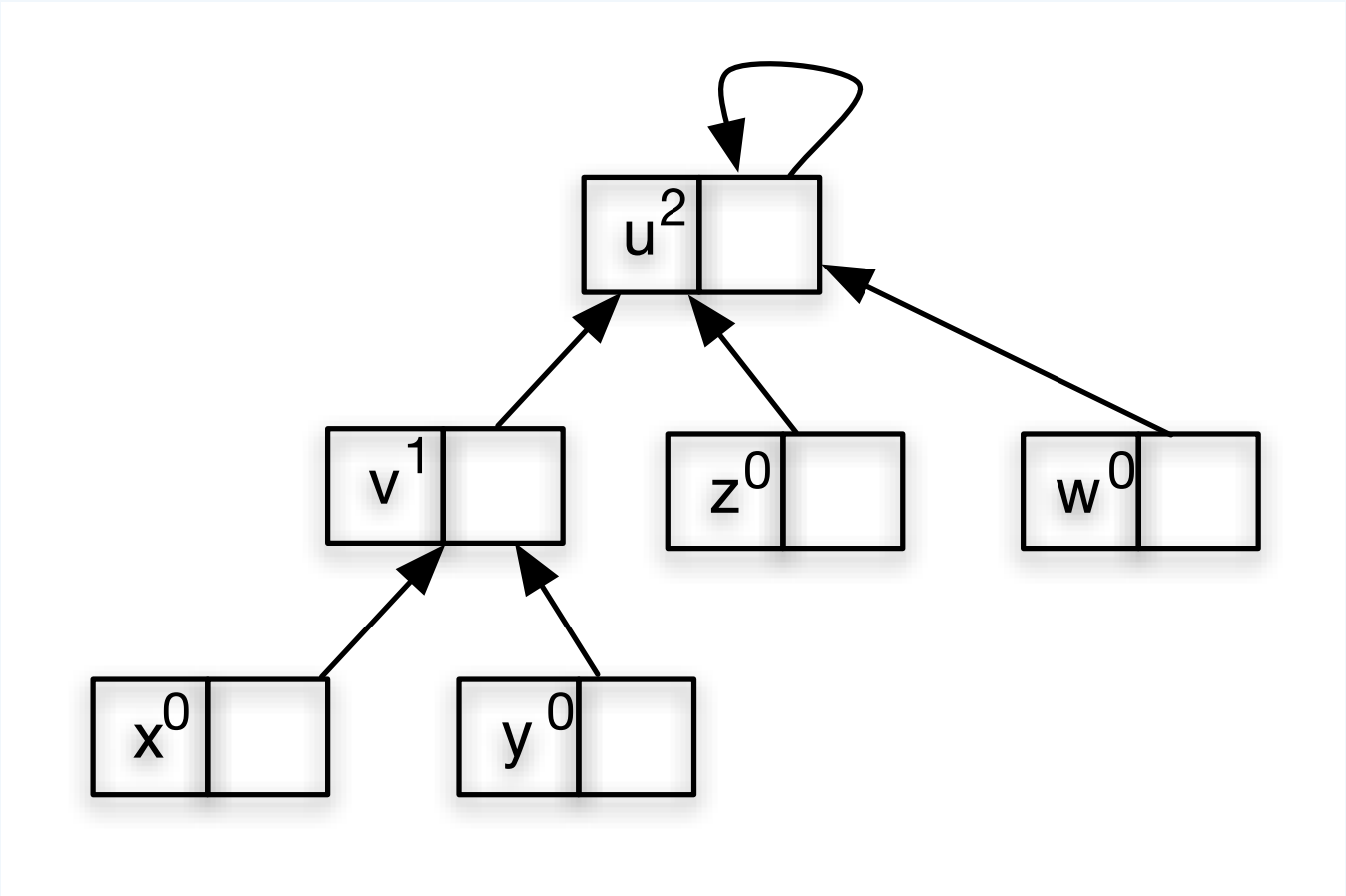
```
Union(u, v)                    //merges the trees where u, v belong
    r_u = Find(u)              //find the root of u's tree
    r_v = Find(v)
    if r_u == r_v then         //if u, v in the same tree, do nothing
        return
    end if
    if rank(r_u) > rank(r_v) then
        π(r_v) = r_u           //make the shorter tree point to the taller
    else
        π(r_u) = r_v
        //if trees equally tall, increase height of resulting tree
        if rank(r_u) == rank(r_v) then
            rank(r_v) = rank(r_v) + 1
        end if
    end if
```

The outcome of a sequence of `Union` operations

Starting from an empty data structure with 6 elements, perform
`Union`(x, v), `Union`(x, y), `Union`(z, u), `Union`(y, u), `Union`(x, w)
(Break ties by making the alphabetically smaller root the new root.

The outcome of a sequence of `Union` operations

Starting from an empty data structure with 6 elements, perform
`Union`(x, v), `Union`(x, y), `Union`(z, u), `Union`(y, u), `Union`(x, w)
(Break ties by making the alphabetically smaller root the new root.



Properties of *rank*

1. $rank(u) < rank(\pi(u))$
   $\Rightarrow$ every element has at most one ancestor of rank k
2. # nodes in the tree of any root node of rank k: $\geq 2^k$
   (by induction)
3. # nodes in the subtree of any node of rank k: $\geq 2^k$
4. subtrees of different rank k nodes are disjoint (by 1.$\Rightarrow$)

$\Rightarrow$ If x nodes of rank k, then $\geq x \cdot 2^k$ nodes in the x subtrees.

Max tree height and worst-case running time

Therefore, if we have n elements in total,

- $x \cdot 2^k \leq n \Rightarrow$ at most $\frac{n}{2^k}$ nodes of rank k
- the maximum rank is $\log_2 n$

Thus *max tree height* $= \log_2 n$

Hence worst-case running time for `Find`, `Union` $= O(\log_2 n)$, and Kruskal's algorithm takes $O((n+m)\log_2 n)$ time.

What if edges are already sorted?

What if edge weights are already sorted?

Or, they are small enough, e.g., $w(e) < m$ for all $e \in E$, so they can be sorted in linear time (e.g., using Bucketsort)?

Then the data structure is the *bottleneck* for the performance of Kruskal's algorithm.

Goal: design a data structure that allows for linear (or almost linear) running time

Maintenance operations that pay off in the long run

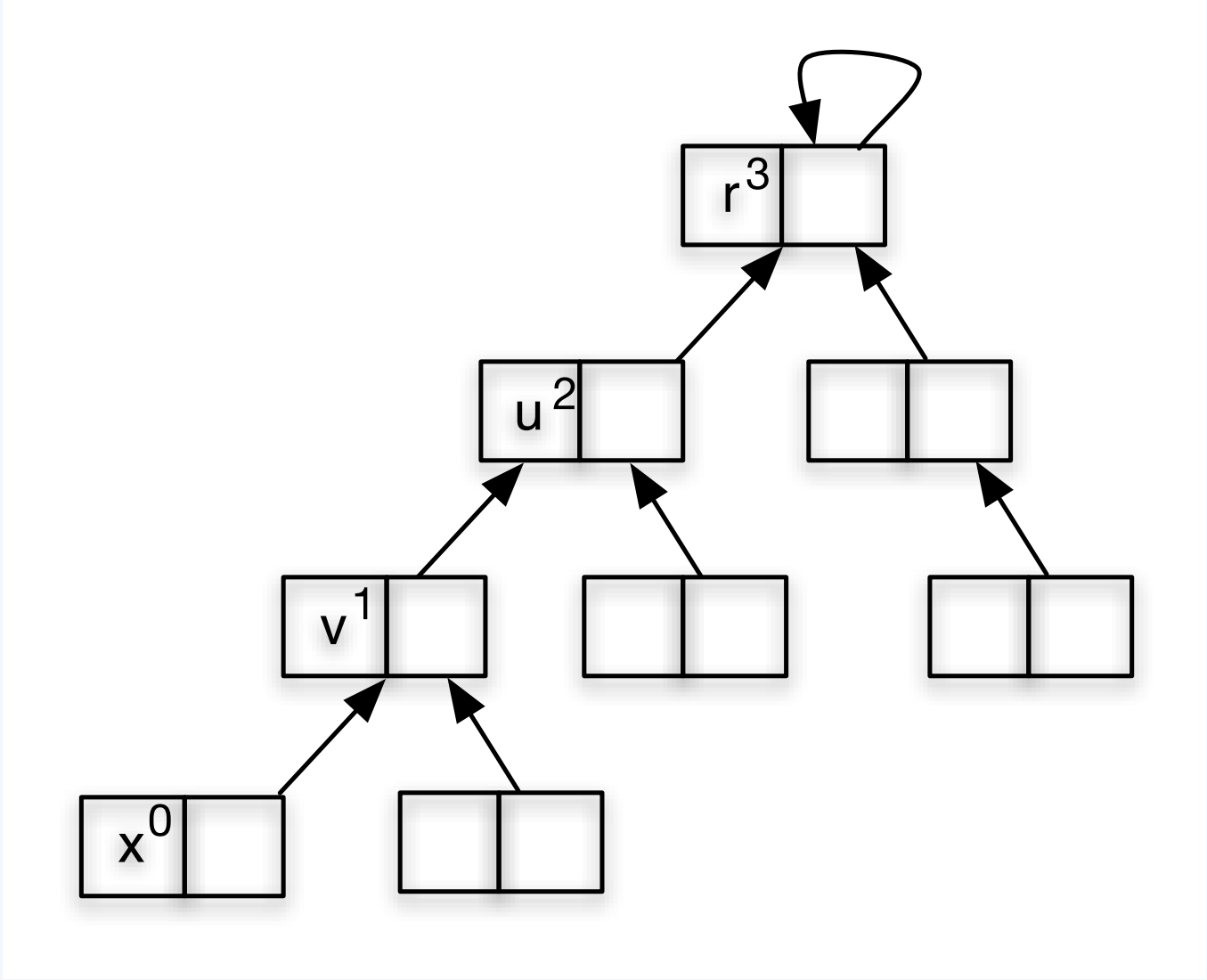
Goal: maintain *short* trees since the time for `Find`(u) corresponds to u's depth in the tree

Heuristic idea: when performing `Find`(u), update the parent pointers of **every** node x on the u-r path to point to r.

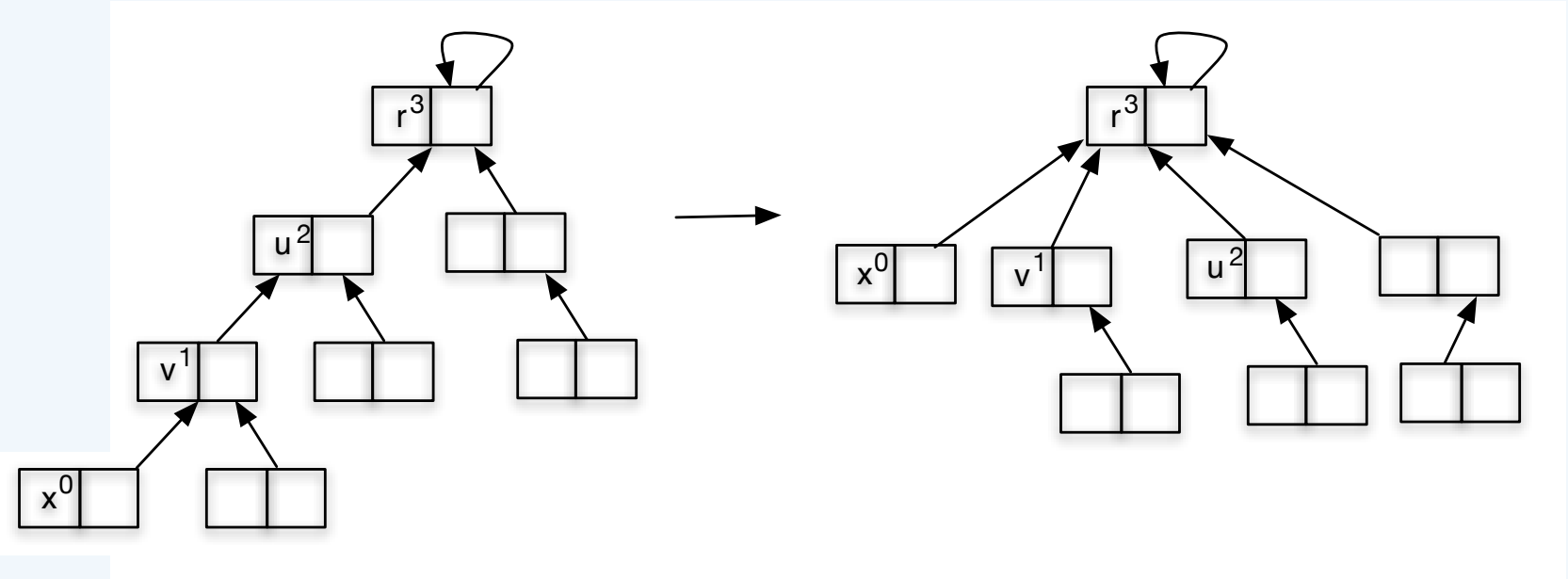
Why? All future `Find`(x) will start from much closer to the root (although x might not point to the root anymore $-$*why?*).

This motivates a different kind of analysis: consider sequences of `Find` and `Union` operations, and look at the **average** time spent per operation (amortized analysis).

Find(x)



`Find`(x) with path compression



Find with **path compression**

//returns the *name* of the set where u belongs
//sets every node on the u-r path to point to r
Find(u)
  while $\pi(u) \neq u$ **do**
    $\pi(u) =$ Find(u)
  end while
  return $\pi(u)$

Remark 1.

Path compression *does not change the ranks of the nodes. However, the rank of a node no longer corresponds to the height of its subtree.*

Roadmap

- A `Find` may still take $O(\log n)$ time (exercise).
- Instead of bounding the max time spent on *individual* `Find` operations, bound the time spent on a *sequence* of them.
- If we perform a total of m `Find` operations, we want to spend linear or almost linear time for all of them.

Roadmap cont'd

1. Partition the nodes in a small number of carefully designed intervals, depending on the nodes' ranks.

2. Think of `Find`(u) operations as traversing two different types of pointers (the type will be determined by the intervals where u and $\pi(u)$ belong).

   - Directly bound the time spent by a single `Find` on pointers of one type.
   - Carefully bound the total time required for traversing all pointers of the second type by all m `Find`'s.

Partitioning nodes into groups

If there are n elements, their ranks range from 0 to $\log n$.

Divide the nonzero ranks into groups as follows:

1. Group 0: $[1]$ $[0+1, 2^0]$
2. Group 1: $[2]$ $[1+1, 2^1]$
3. Group 2: $[3, 4]$ $[2+1, 2^2]$
4. Group 3: $[5, 16]$ $[4+1, 2^4]$
5. Group 4: $[17, 2^{16}]$ $[16+1, 2^{16}]$
6. Group 5: $[65537, 2^{65537}]$ $[65536+1, 2^{65537}]$
7. ...

#nodes in group $[k+1, 2^k]$

$\log^* n$ = #iterations of the $\log_2$ function on n until we get a number less than or equal to 1
Examples: $\log^* 4 = 2, \log^* 16 = 3$

- Group i is of the form $(2^{i-1}, 2^{2^{i-1}}]$ (except for group 0).
- For simplicity, denote groups by $[k+1, 2^k]$.
- Total # groups: $\leq \log^* n$ (*why?*)
- For all practical purposes, $\log^* n \leq 5$ (else, $n \geq 2^{65537}$).

Fact 1.

There are at most $\frac{n}{2^k}$ *elements in group* $[k+1, 2^k]$.

Creative accounting

Idea: assign 2^k dollars (corresponding to units of time) to every node in group $[k+1, 2^k]$.

By Fact 1, we are spending at most extra $n \log^* n$ dollars for all nodes (this amount is "linear" in n).

We will spend these dollars to pay for the work required by `Find` to follow pointers such that the ranks of the end nodes belong to the same group.

Types of pointers in a `Find` operation

Let $v = \pi(u)$. Recall that `Find`(u) follows a sequence of pointers. We distinguish between two types of pointers.

1. Type 1: a pointer is of Type 1 if u and v belong to different groups, or if v is the root.
2. Type 2: a pointer is of Type 2 if u and v belong to the same group.

We *account* for the two Types of pointers in two different ways:

1. Type 1 pointers are charged directly to the `Find` operation
2. Type 2 pointers are charged to u, who pays using its pocket money

Counting the work spent on `Find`(u) operations

Suppose u belongs to group $[k+1, 2^k]$. Let $v = \pi(u)$.

1. Type 1 pointers: charged directly to the `Find` operation
   At most $\log^* n$ pointers of Type 1 in **each** `Find` operation.

2. Type 2 pointers: recall that every node with rank in group $[k+1, 2^k]$ is given 2^k dollars (units of time).
   u pays a dollar for each of them using its pocket money.

Does u have enough money to pay for the Type 2 pointers in all m `Find` *operations?*

u's allowance suffices for all m operations

Recall that

- both u, v are in group $[k+1, 2^k]$;
- each `Find`(u) causes u to pay a dollar.

Key observation: each `Find`(u) causes $\pi(u)$ to point to the root of u's tree; so $rank(v)$ increases by at least 1.

How many times can v's rank increase before u and v are in different groups?

Fewer than 2^k.

Summary

Suppose we perform $2m$ `Find` operations.

- Each `Find` is charged at most $\log^* n$ dollars.
  Hence all $2m$ `Find` require at most $O(m \log^* n)$ time.
- We spend at most extra $n \log^* n$ dollars.

$\Rightarrow$ The total amount of time spent for a sequence of $2m$ `Find` and $n - 1$ `Union` operations is

$$O((n + m) \log^* n).$$

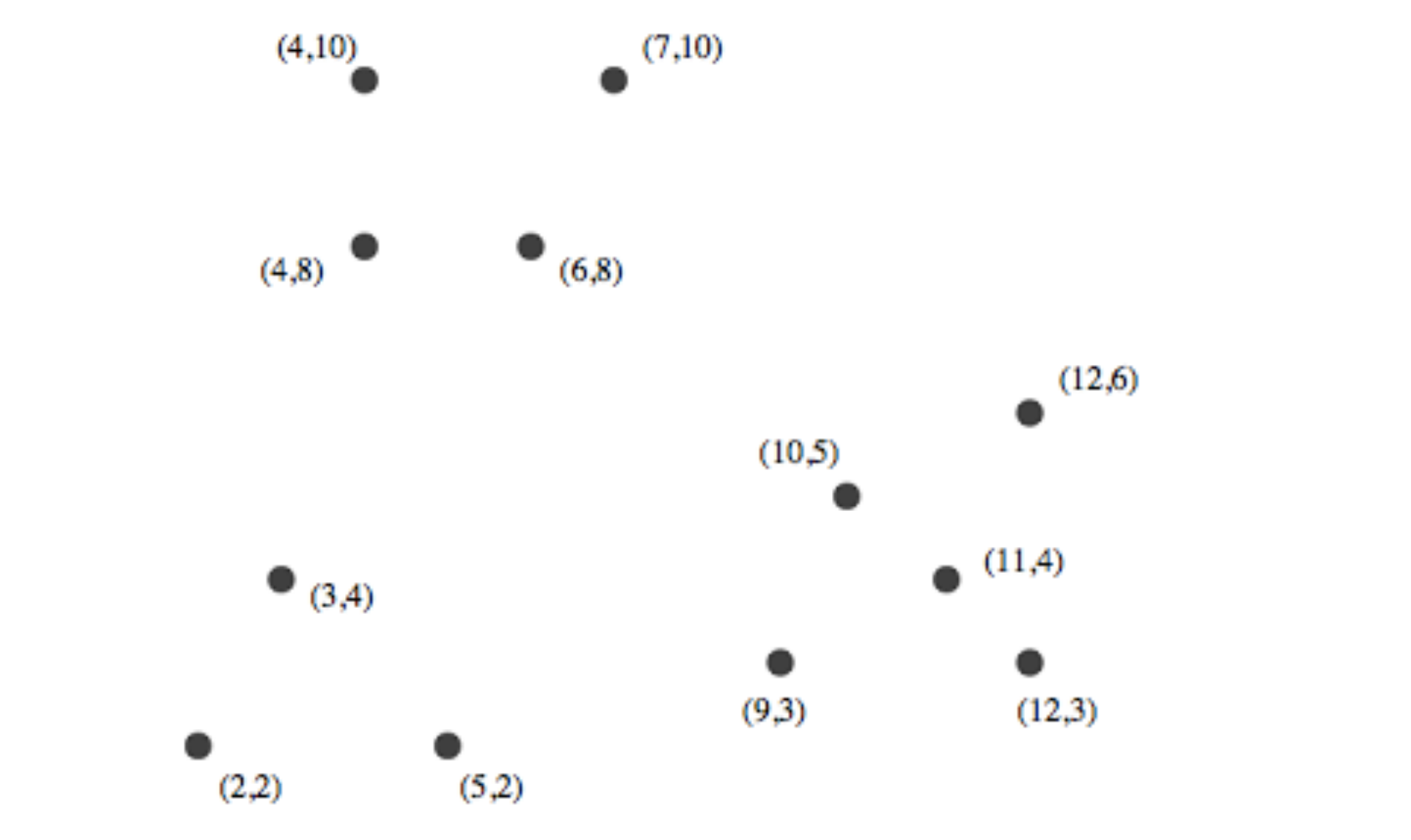
$\Rightarrow$ **On average**, every `Find` operation takes $\log^* n$ time.

Today

1. Recap: Kruskal's algorithm for MSTs
2. A union-find data structure for disjoint sets
3. Application: Clustering

Clustering

Input: a set of n data points on the plane $p_i = (x_i, y_i)$
Output: k clusters containing at least one element each, according to some clustering criterion



Hierarchical agglomerative clustering

- Initially, every point is a cluster (thus there are n clusters).
- Two clusters will be merged at every step according to some criterion.
- Merging will stop once k clusters have been formed.

Generic hierarchical clustering algorithm

while more than k clusters exist **do**
    Find the two best clusters according to clustering criterion
    Merge these two clusters into one cluster
end while

Criterion: maximize distance between clusters

The distance between two clusters is the minimum distance between any two points p, q such that p belongs to one cluster and q to the other.

Goal: Output k clusters so that the distances between them are maximal.

Using Kruskal's algorithm for clustering

Idea: use the following merging rule; always merge the two clusters that are at minimum distance (nearest neighbors)

1. For all $1 \leq i < j \leq n$ compute $dist(p_i, p_j) = ||p_i - p_j||$
2. Let G be the complete graph with $V = \{p_1, \ldots, p_n\}$; every edge has an associated weight $weight(p_i, p_j) = dist(p_i, p_j)$.
3. Run Kruskal's algorithm on G until $n - k$ edges have been included in A; the graph $H = (V, A)$ consists of k trees.
4. Output the k trees of H as the k clusters.

Kruskal's algorithm returns the optimal clustering

Equivalently,

1. run Kruskal's algorithm and obtain a full MST T
2. remove the $k-1$ most expensive edges from T
3. output the resulting k connected components as k clusters of maximum spacing

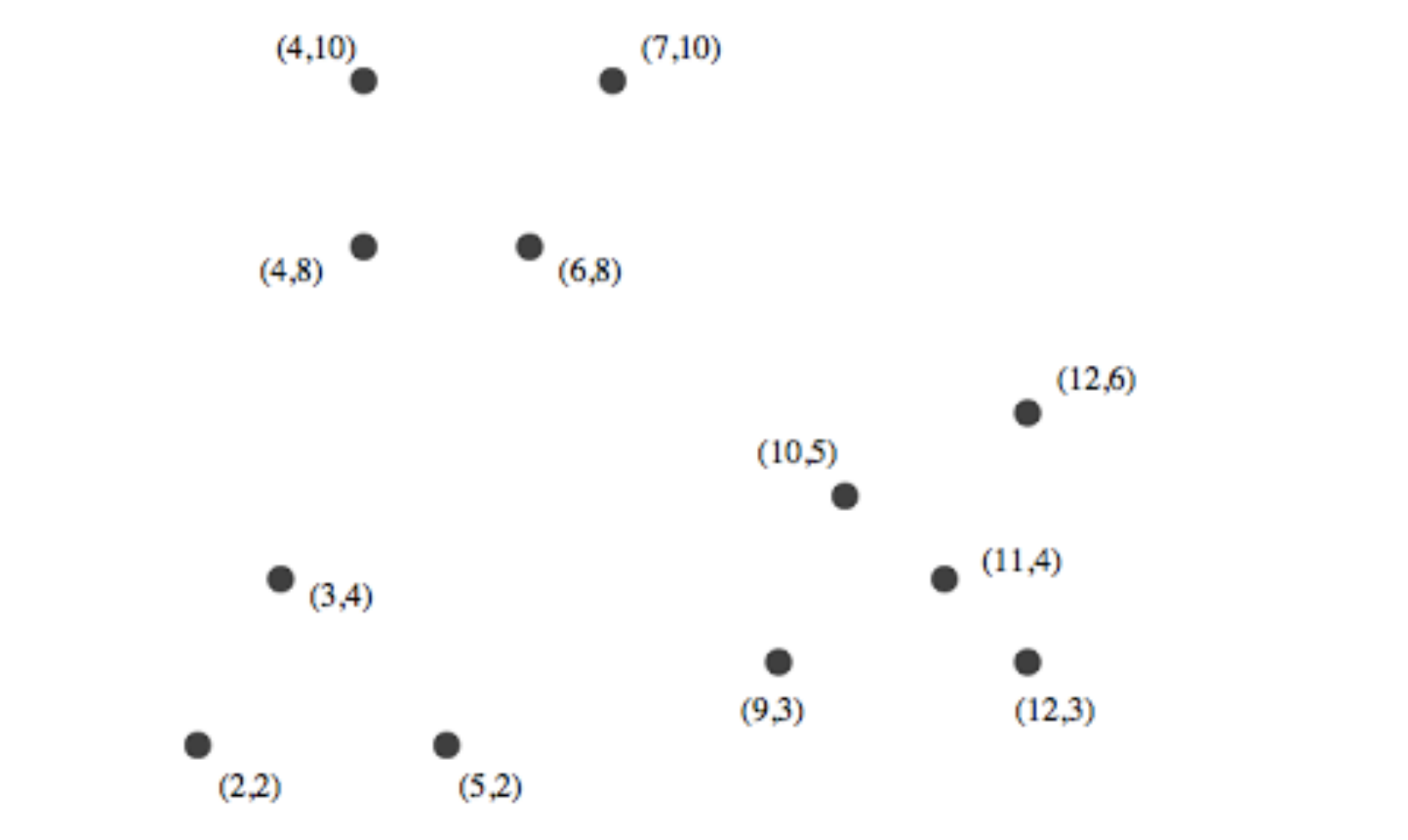
Can show that the algorithm above is optimal for this clustering criterion (**exercise**).

Example data set

Input:

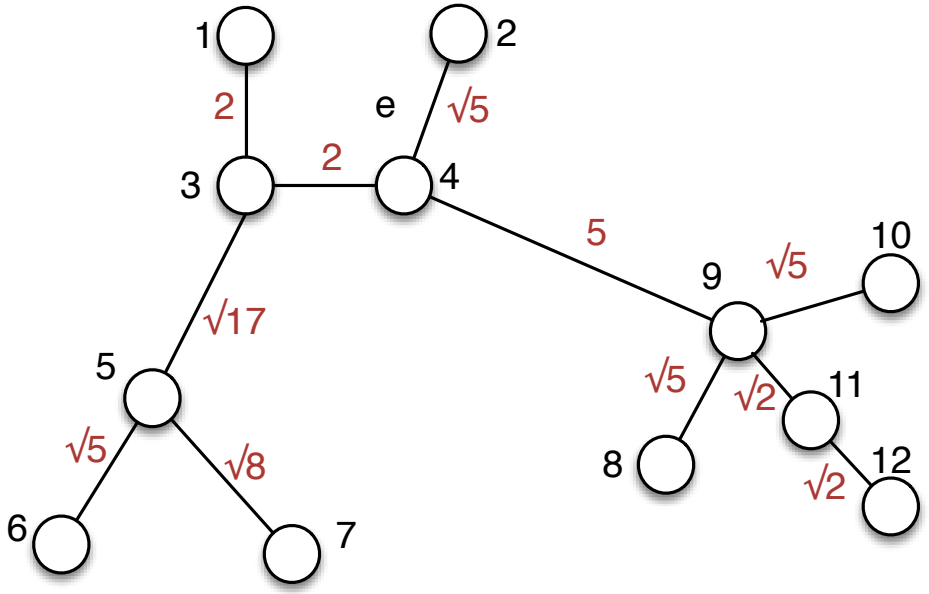
a set of $n = 12$ data points $p_i = (x_i, y_i)$

Output: $k = 3$ clusters containing at least one element each



Kruskal's MST for example data set

- Compute all pairwise distances between the $n = 12$ points
- Form a weighted K_{12}: for every $e = (u, v)$, $w(e) = dist(u, v)$
- Run Kruskal's algorithm on the weighted K_{12}
- Output: an MST of the weighted K_{12}



Clusters for example data set

- Remove the $k - 1 = 2$ heaviest edges from the MST; the graph now consists of $k = 3$ connected components (disjoint trees)
- Output the disjoint trees as the k clusters