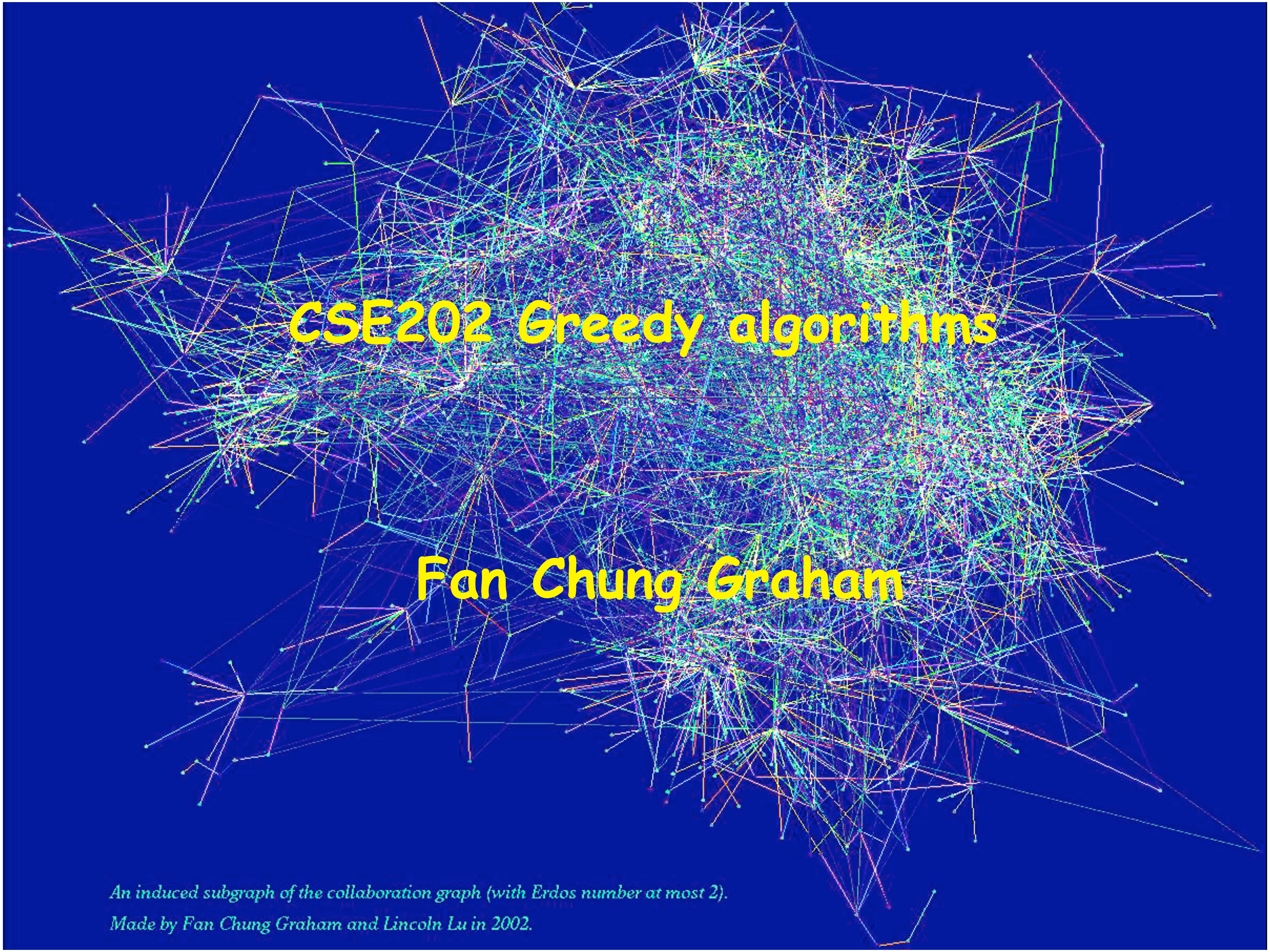
CSE202 Greedy algorithms

Fan Chung Graham



An induced subgraph of the collaboration graph (with Erdos number at most 2).
Made by Fan Chung Graham and Lincoln Lu in 2002.

Announcement

- Reminder: Homework #1 has been posted, due April 15.
- This lecture includes material in Chapter 5 of *Algorithms,* Dasgupta, Papadimitriou and Vazirani, http://www.cs.berkeley.edu/~vazirani/algorithms/chap5.pdf

FIFO

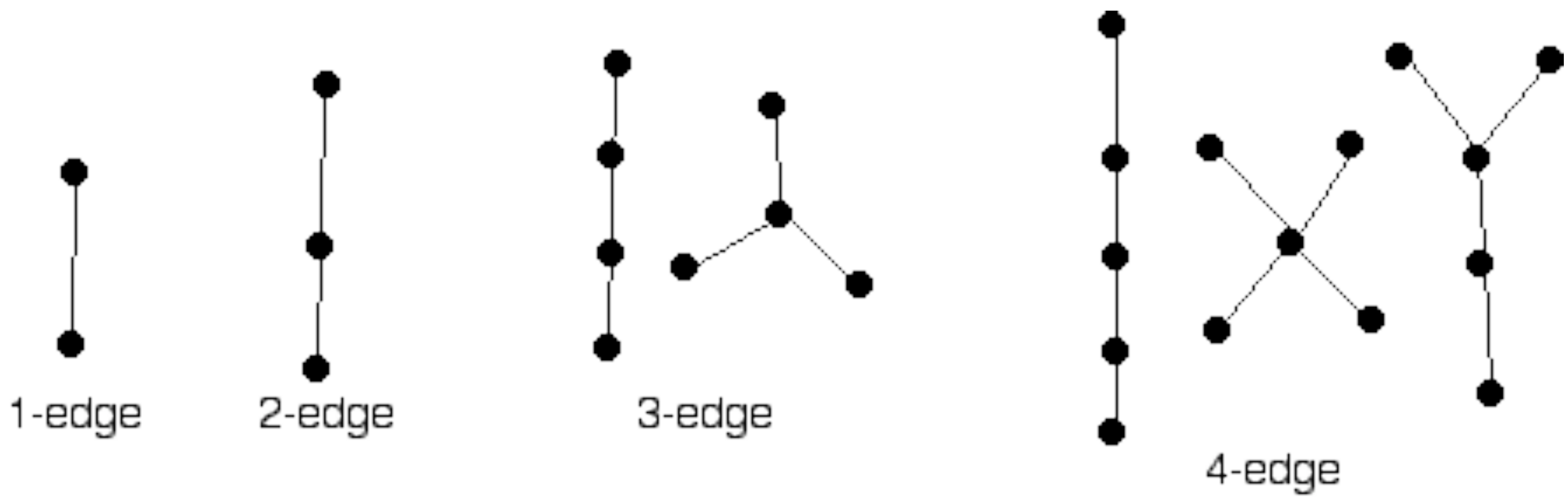
LIFO

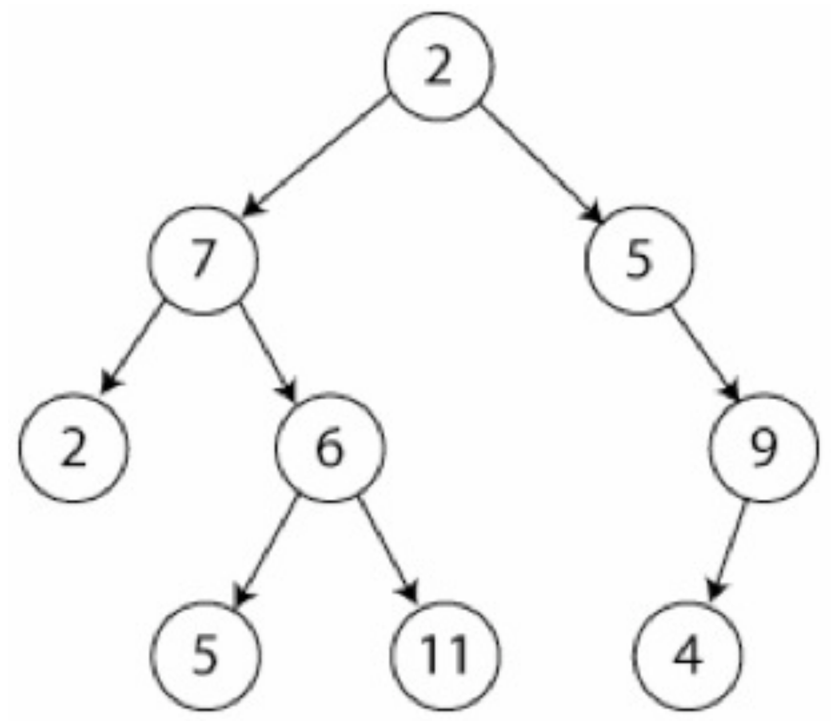
Queue

Stack

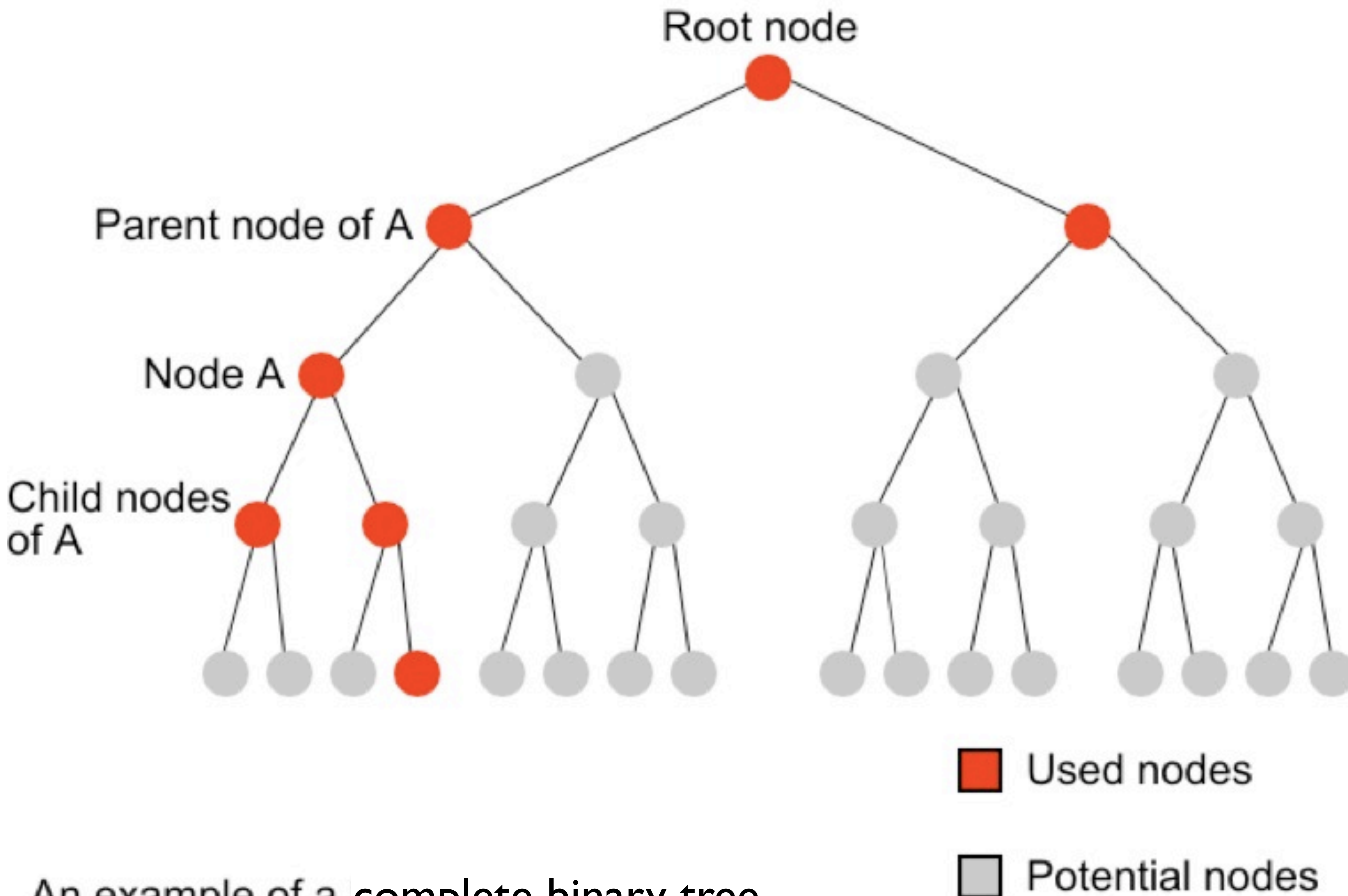
tree

Trees with at most 4 edges

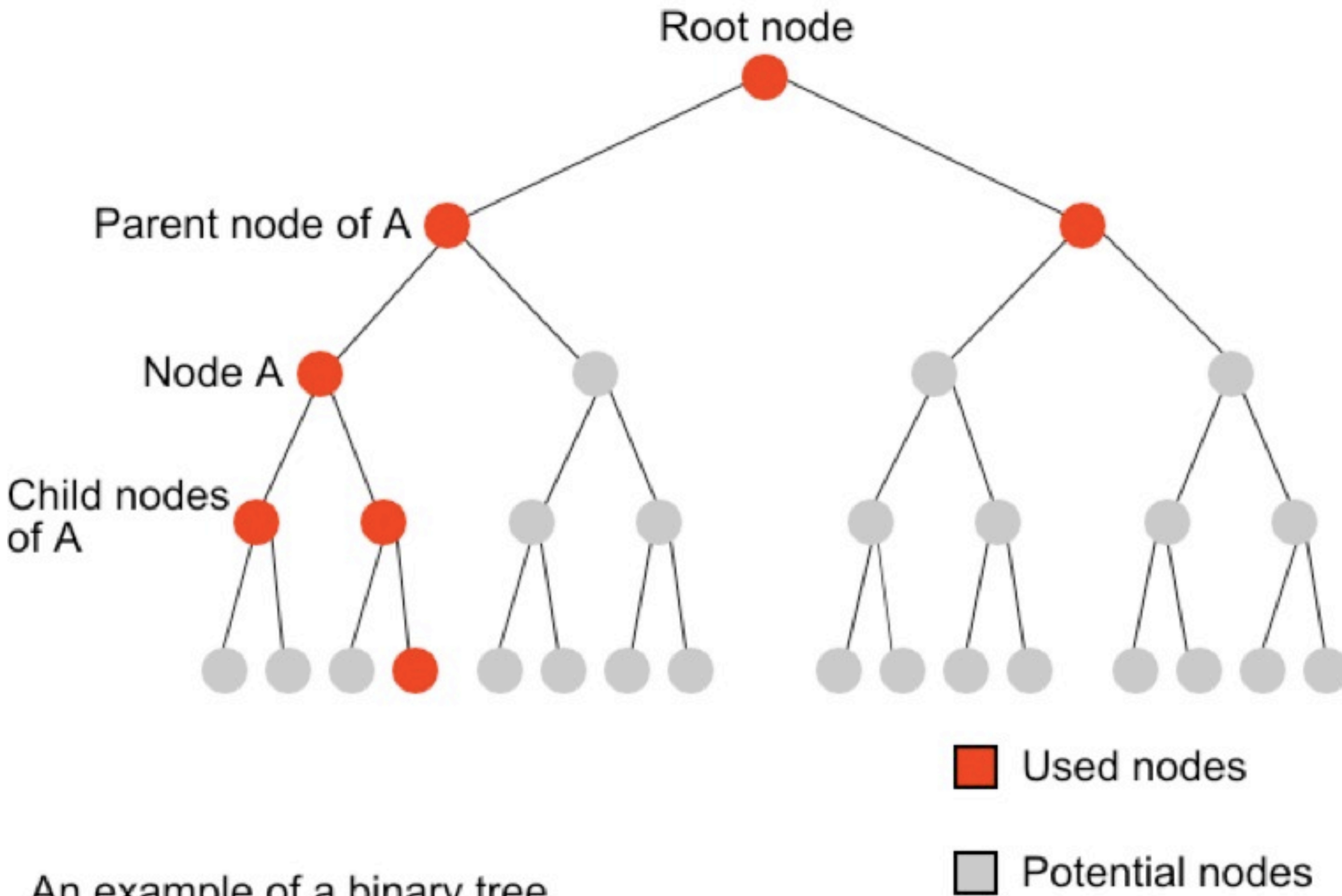




A binary tree



An example of a complete binary tree



An example of a binary tree

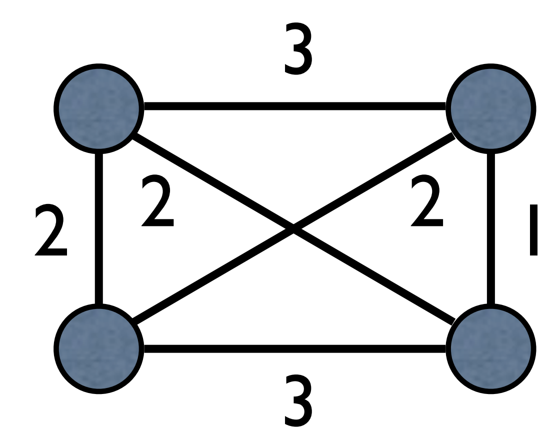
A k-level complete binary tree has ?? vertices.

Greedy Algorithms

- Minimum Spanning Trees
- The Union/Find Data Structure

A Network Design Problem

Problem: Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.



A Network Design Problem

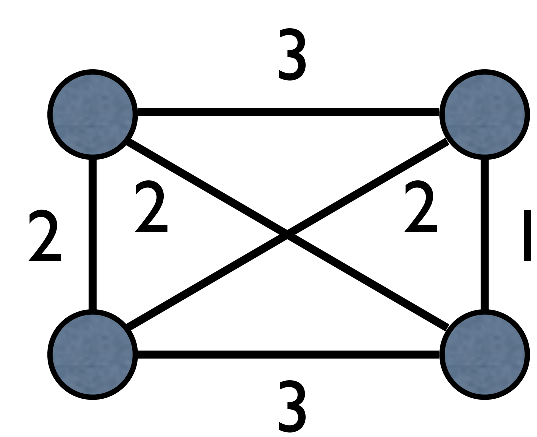
Problem: Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

Graph-Theoretic Formulation:

Node = Computer

Edge = Pair of computers

Edge Cost(u,v) = Distance(u,v)



A Network Design Problem

Problem: Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

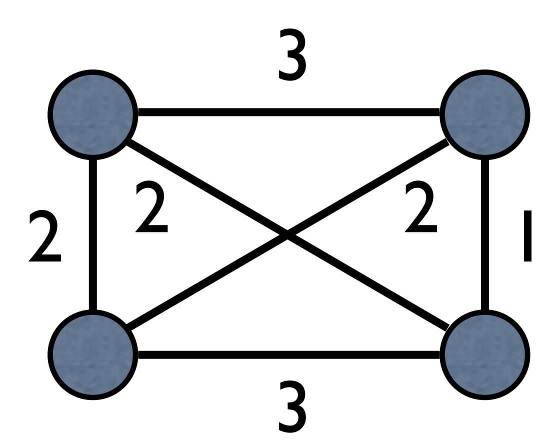
Graph-Theoretic Formulation:

Node = Computer

Edge = Pair of computers

Edge Cost(u,v) = Distance(u,v)

Find a subset of edges T such that the cost of T is minimum and all nodes are connected in (V,T)



A Network Design Problem

Problem: Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

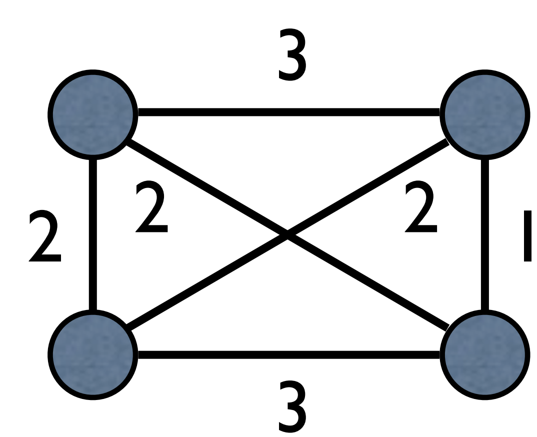
Graph-Theoretic Formulation:

Node = Computer

Edge = Pair of computers

Edge Cost(u,v) = Distance(u,v)

Find a subset of edges T such that the cost of T is minimum and all nodes are connected in (V,T)



Can U contain a cycle?

A Network Design Problem

Problem: Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

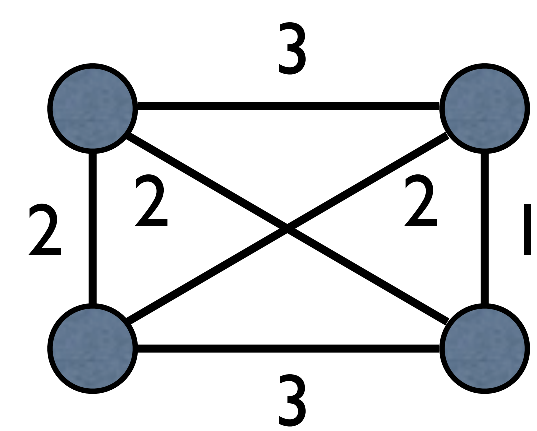
Graph-Theoretic Formulation:

Node = Computer

Edge = Pair of computers

Edge Cost(u,v) = Distance(u,v)

Find a subset of edges T such that the cost of T is minimum and all nodes are connected in (V,T).

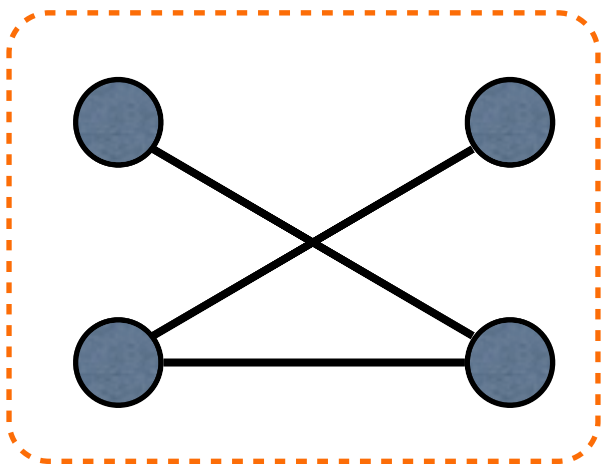
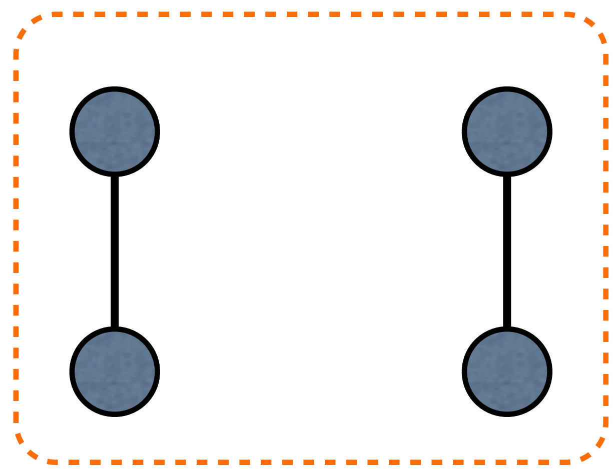
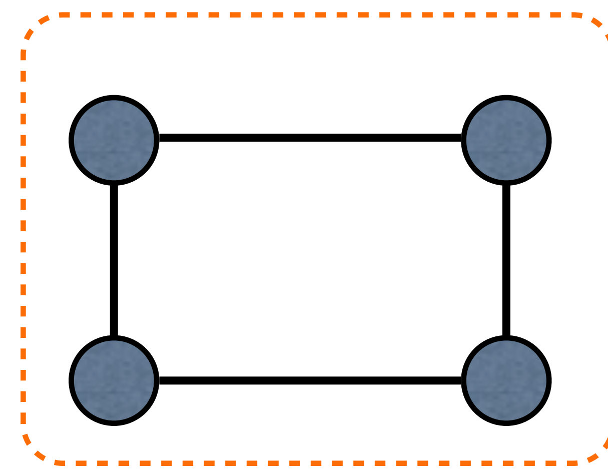


Can U contain a cycle?

Solution is connected and acyclic, so a **tree**.

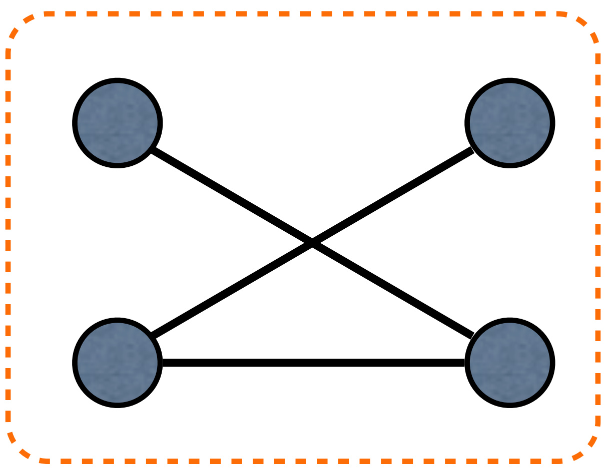
Trees

A connected, undirected and acyclic graph is called a **tree**.

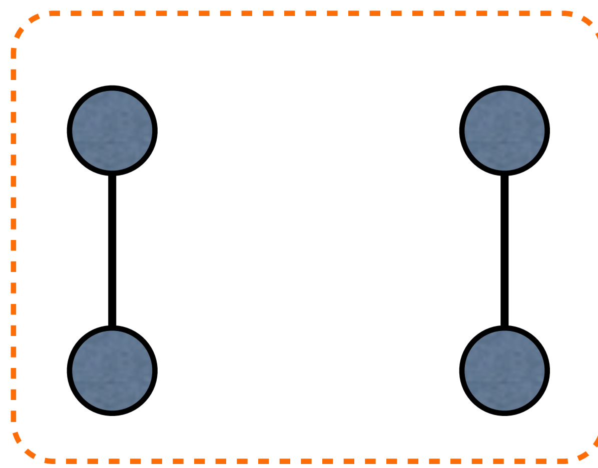




Trees

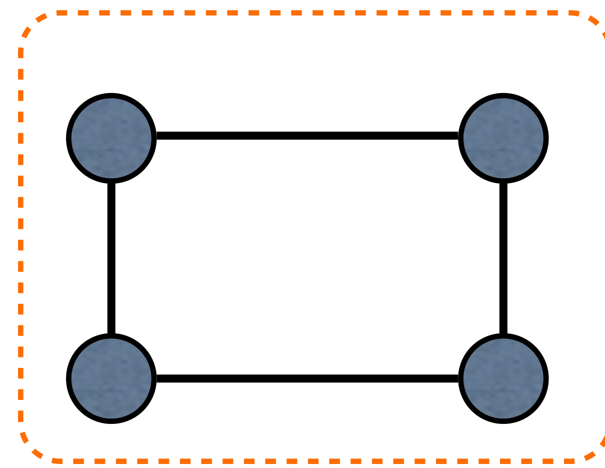
A connected, undirected and acyclic graph is called a **tree**.



Tree



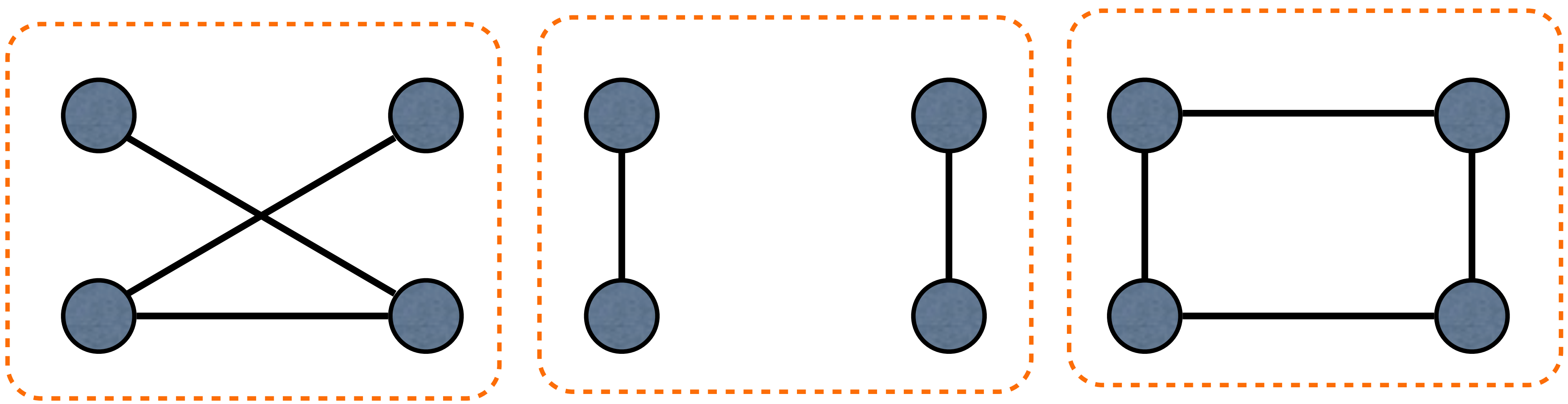
Not Tree



Not Tree

Trees

A connected, undirected and acyclic graph is called a **tree.**



Tree | Not Tree | Not Tree

Property 1. A tree on n nodes has exactly n - 1 edges.

Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

Proof. By induction.

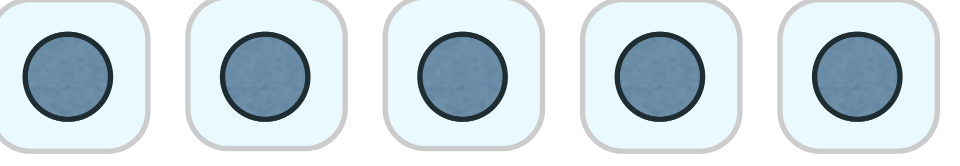
Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

Proof. By induction.

Base Case:
n nodes, no edges,
n connected components.



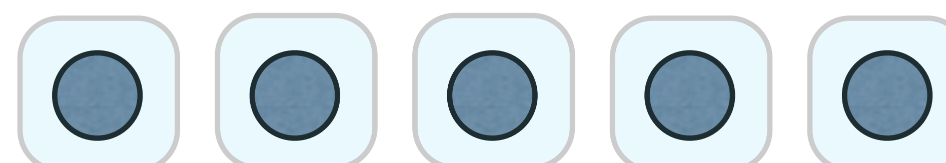
Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

Proof. By induction.

Base Case:
n nodes, no edges,
n connected components.



Inductive Case:
Add edge between two
connected components.
No cycle created.
#components decreases by 1

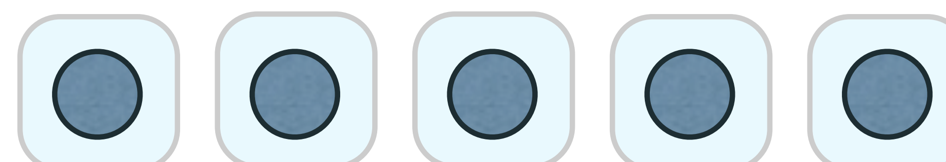
Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

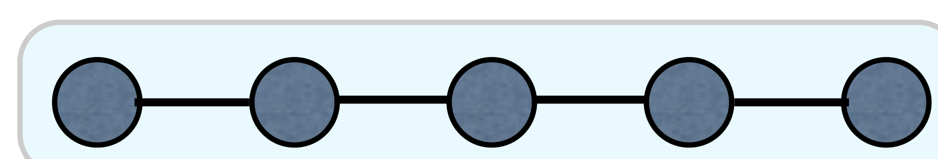
Proof. By induction.

Base Case:
n nodes, no edges,
n connected components.



Inductive Case:
Add edge between two
connected components.
No cycle created.
#components decreases by 1.

At the end: 1 component.



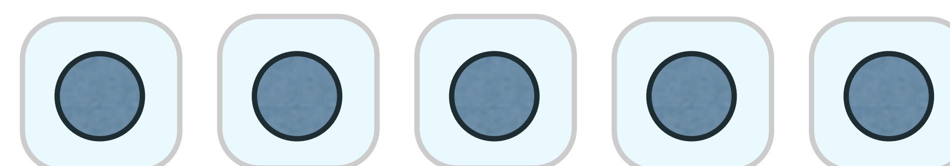
Trees

A connected, undirected and acyclic graph is called a **tree**

Property 1. A tree on n nodes has exactly n - 1 edges

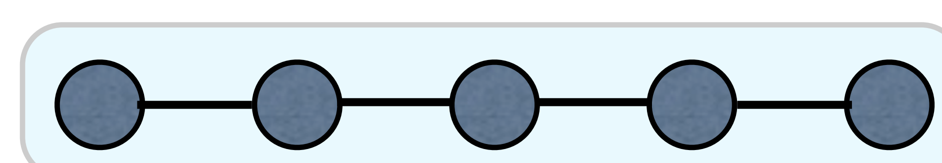
Proof. By induction.

Base Case:
n nodes, no edges,
n connected components



Inductive Case:
Add edge between two
connected components
No cycle created
#components decreases by 1

At the end: 1 component



How many edges were added?

Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

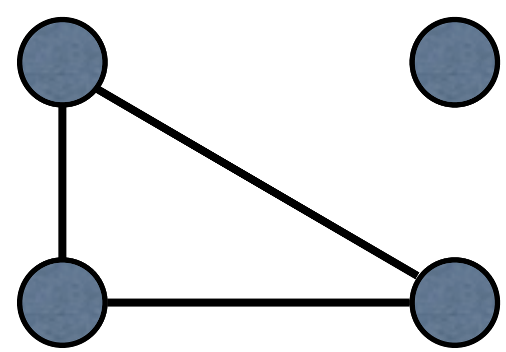
Is any graph on n nodes and n - 1 edges a tree?

Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

Is any graph on n nodes and n - 1 edges a tree?

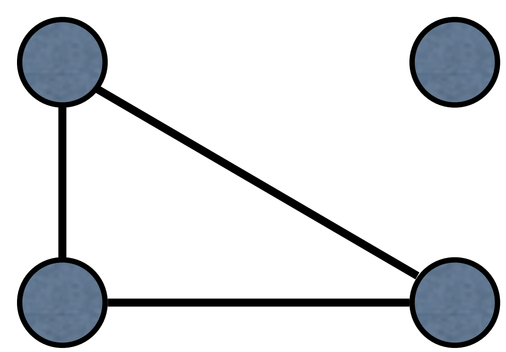


Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

Is any graph on n nodes and n - 1 edges a tree?



Property 2. Any **connected**, undirected graph on n nodes and n - 1 edges is a tree.

Trees

A connected, undirected and acyclic graph is called a **tree**.

Property 1. A tree on n nodes has exactly n - 1 edges.

Property 2. Any **connected**, undirected graph on n nodes and n - 1 edges is a tree.

Proof: Suppose G is connected, undirected, has some cycles.

Trees

A connected, undirected and acyclic graph is called a **tree**

Property 1. A tree on n nodes has exactly n - 1 edges

Property 2. Any **connected**, undirected graph on n nodes and n - 1 edges is a tree

Proof: Suppose G is connected, undirected, has some cycles. While G has a cycle, remove an edge from this cycle.

Trees

A connected, undirected and acyclic graph is called a **tree**

Property 1. A tree on n nodes has exactly n - 1 edges

Property 2. Any **connected**, undirected graph on n nodes and n - 1 edges is a tree

Proof: Suppose G is connected, undirected, has some cycles.
While G has a cycle, remove an edge from this cycle.
Result: G' = (V, E') where E' is a tree. So |E'| = n - 1

Trees

A connected, undirected and acyclic graph is called a **tree**

Property 1. A tree on n nodes has exactly n - 1 edges

Property 2. Any **connected**, undirected graph on n nodes and n - 1 edges is a tree

Proof: Suppose G is connected, undirected, has some cycles.
While G has a cycle, remove an edge from this cycle.
Result: G' = (V, E') where E' is a tree. So |E'| = n - 1.
Thus, E = E', and G is a tree.

Minimum Spanning Trees (MST)

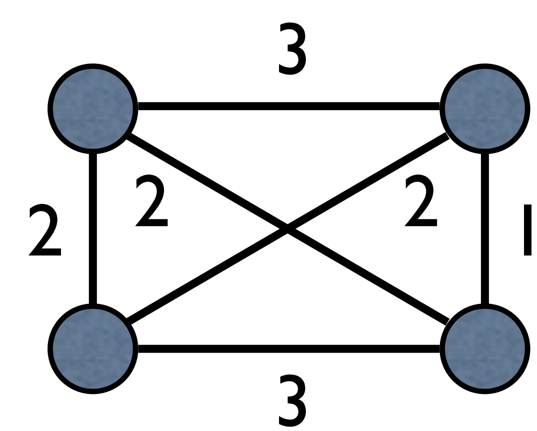
Problem: Given distances between a set of computers, find the cheapest set of pairwise connections so that they are all connected.

Graph-Theoretic Formulation:

Node = Computer

Edge = Pair of computers

Edge Cost(u,v) = Distance(u,v)



Find a subset of edges T such that the cost of T is minimum and all nodes are connected in (V,T).

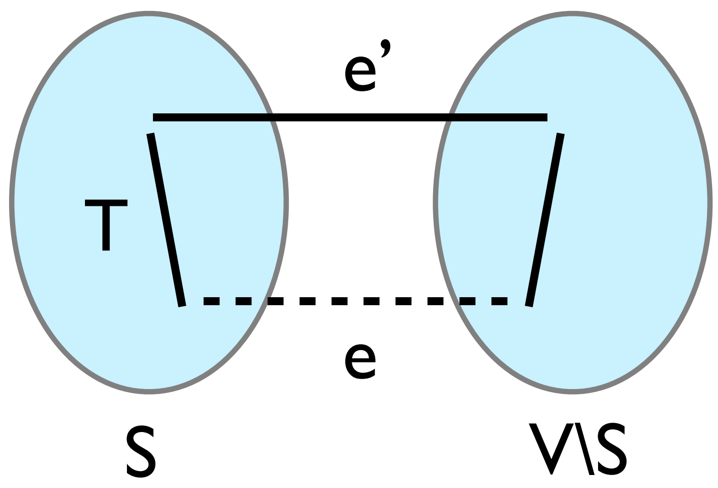
Goal: Find a spanning tree T of the graph G with minimum total cost

We'll see a greedy algorithm to construct T.

Properties of MSTs

For a cut (S, V\S), the lightest edge in the cut is the minimum cost edge that has one end in S and the other in V\S.
Assume all edge costs are distinct.

Property 1. A lightest edge in any cut always belongs to an MST



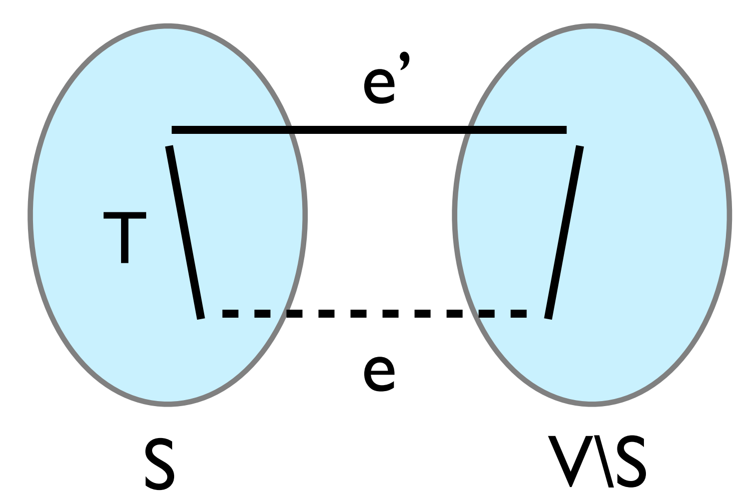
Properties of MSTs

For a cut (S,V\S), the lightest edge in the cut is the minimum cost edge that has one end in S and the other in V\S.
Assume all edge costs are distinct.

Property 1. A lightest edge in any cut always belongs to an MST

Proof. Suppose not.

Let e = lightest edge in (S,V\S), T = MST, e is not in T



Properties of MSTs

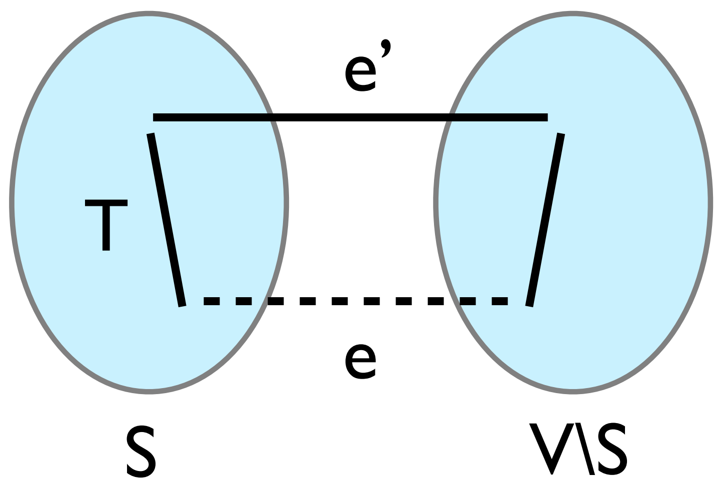
For a cut (S,V\S), the lightest edge in the cut is the minimum cost edge that has one end in S and the other in V\S.
Assume all edge costs are distinct.

Property 1. A lightest edge in any cut always belongs to an MST.

Proof. Suppose not.

Let e = lightest edge in (S,V\S), T = MST, e is not in T.

T U {e} has a cycle with edge e' across (S,V\S).



Properties of MSTs

For a cut (S,V\S), the lightest edge in the cut is the minimum cost edge that has one end in S and the other in V\S.
Assume all edge costs are distinct.

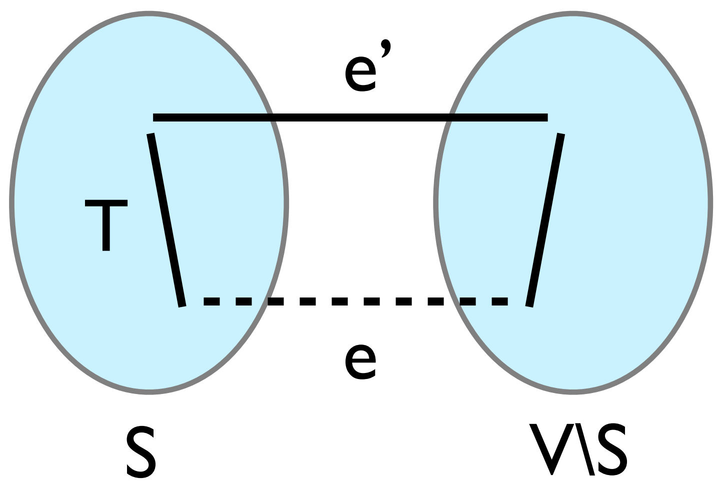
Property 1. A lightest edge in any cut always belongs to an MST.

Proof. Suppose not.

Let e = lightest edge in (S,V\S), T = MST, e is not in T.

T U {e} has a cycle with edge e' across (S,V\S).

Let T' = T \ {e'} U {e}.



Properties of MSTs

For a cut (S,V\S), the lightest edge in the cut is the minimum cost edge that has one end in S and the other in V\S.
Assume all edge costs are distinct.

Property 1. A lightest edge in any cut always belongs to an MST.

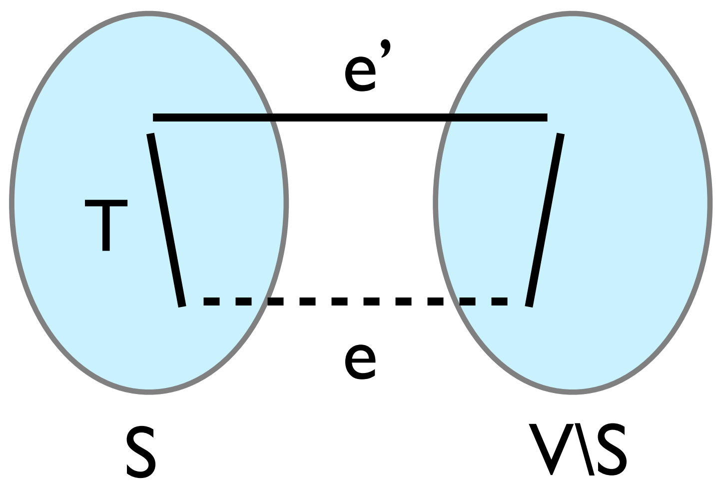
Proof. Suppose not.

Let e = lightest edge in (S,V\S), T = MST, e is not in T.

T U {e} has a cycle with edge e' across (S,V\S).

Let T' = T \ {e'} U {e}.

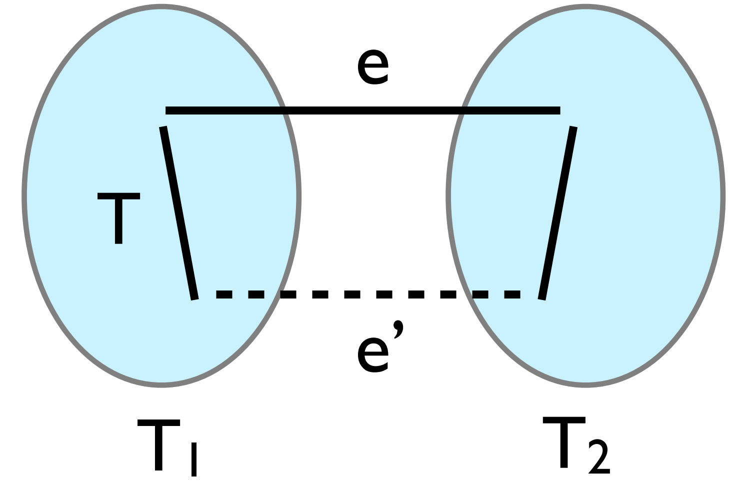
cost(T') = cost(T) + cost(e) - cost(e') < cost(T)



Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle.

Property 2. The heaviest edge in a cycle never belongs to an MST unless all edges in the cycle has the same cost.

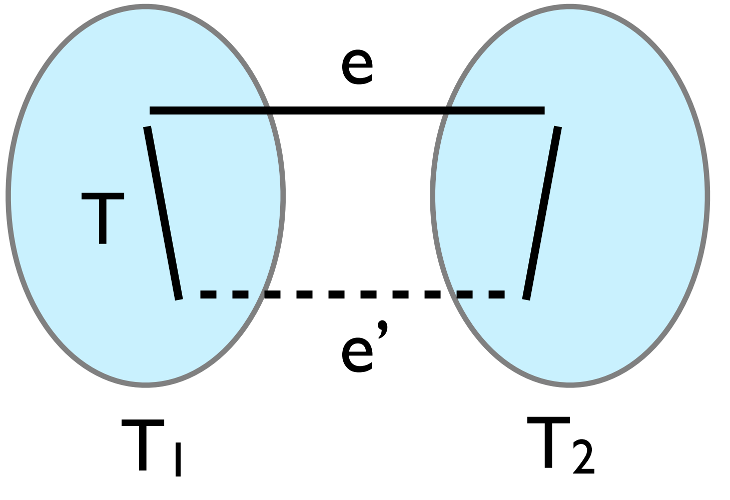


Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle.

Property 2. The heaviest edge in a cycle never belongs to an MST.

Proof. Suppose not. Let T = MST, e = heaviest edge in some cycle, e in T
Suppose that cost(e) is greater than the cost of other edges in the cycle.

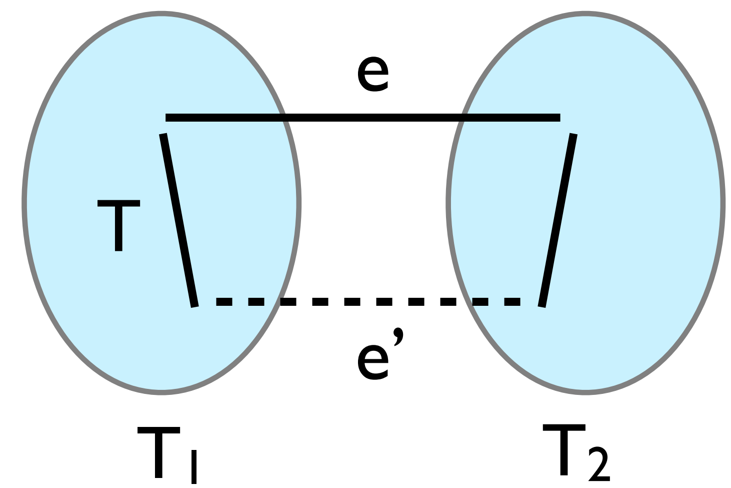


Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle.

Property 2. The heaviest edge in a cycle never belongs to an MST.

Proof. Suppose not. Let T = MST, e = heaviest edge in some cycle, e in T.
Suppose that cost(e) is greater than the cost of other edges in the cycle.
Delete e from T to get subtrees T_1 and T_2.



Properties of MSTs

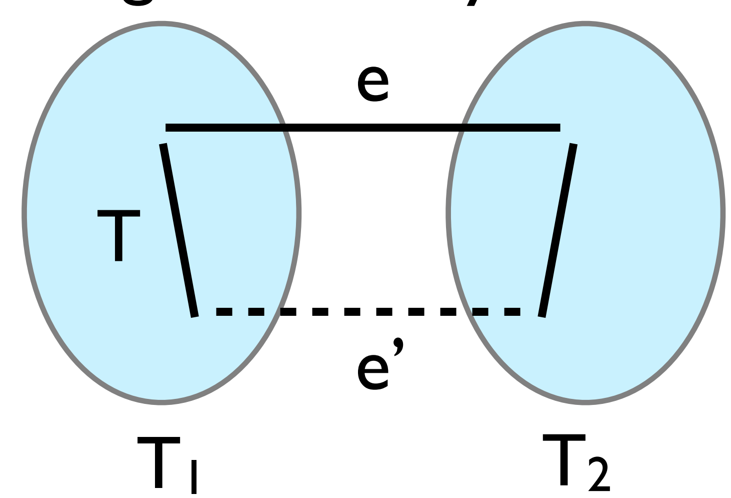
The heaviest edge in a cycle is the maximum cost edge in the cycle.

Property 2. The heaviest edge in a cycle never belongs to an MST.

Proof. Suppose not. Let T = MST, e = heaviest edge in some cycle, e in T.
Suppose that cost(e) is greater than the cost of other edges in the cycle.
Delete e from T to get subtrees T_1 and T_2.

Let e' = lightest edge in the cut (T_1, V \ T_1).

Then, cost(e') < cost(e).



Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle.

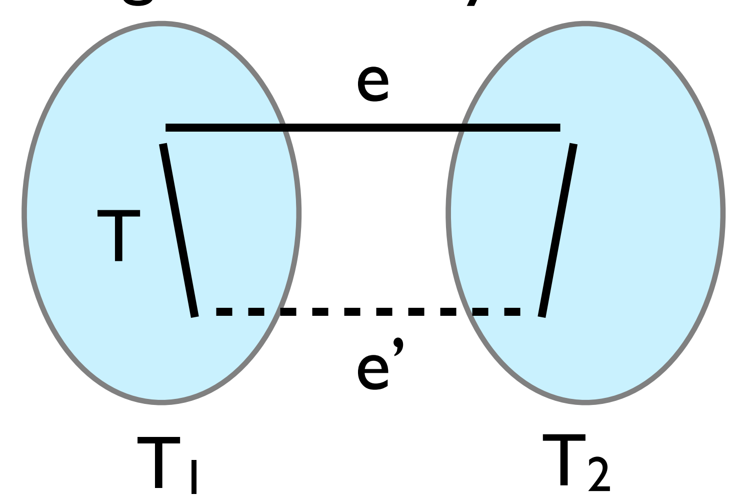
Property 2. The heaviest edge in a cycle never belongs to an MST.

Proof. Suppose not. Let T = MST, e = heaviest edge in some cycle, e in T
Suppose that cost(e) is greater than the cost of other edges in the cycle.
Delete e from T to get subtrees T_1 and T_2

Let e' = lightest edge in the cut (T_1, V \ T_1)

Then, cost(e') < cost(e)

Let T' = T \ {e} + {e'}.



Properties of MSTs

The heaviest edge in a cycle is the maximum cost edge in the cycle.

Property 2. The heaviest edge in a cycle never belongs to an MST.

Proof. Suppose not. Let T = MST, e = heaviest edge in some cycle, e in T.
Suppose that cost(e) is greater than the cost of other edges in the cycle.
Delete e from T to get subtrees T_1 and T_2.

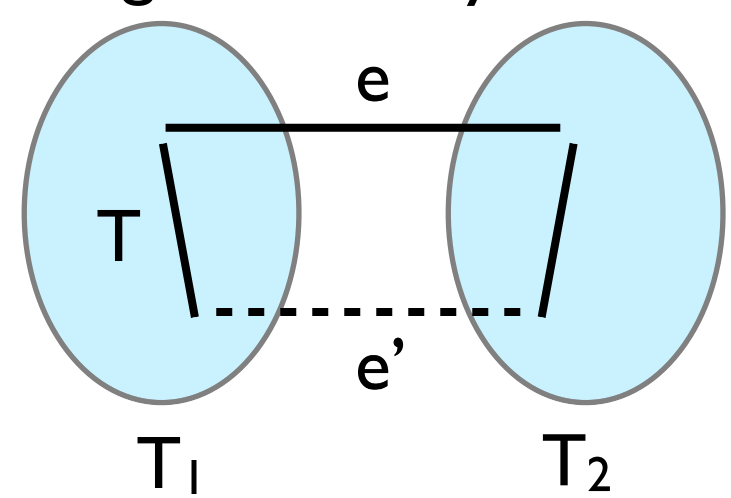
Let e' = lightest edge in the cut (T_1, V \ T_1).

Then, cost(e') < cost(e).

Let T' = T \ {e} + {e'}.

cost(T') = cost(T) + cost(e) - cost(e') < cost(T)

Contradiction.



Summary: Properties of MSTs

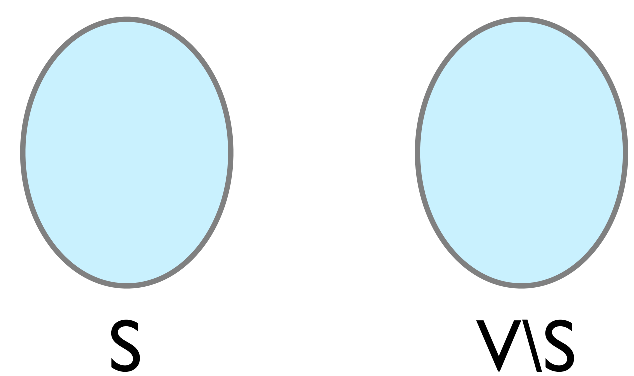
Property 1. A lightest edge in any cut always belongs to an MST.

Property 2. The heaviest edge in a cycle never belongs to an MST.

A Generic MST Algorithm

X = { }
While there is a cut (S, V\S) s.t. X has no edges across it
        X = X + {e}, where e is the lightest edge across (S,V\S).

Does this output a tree?



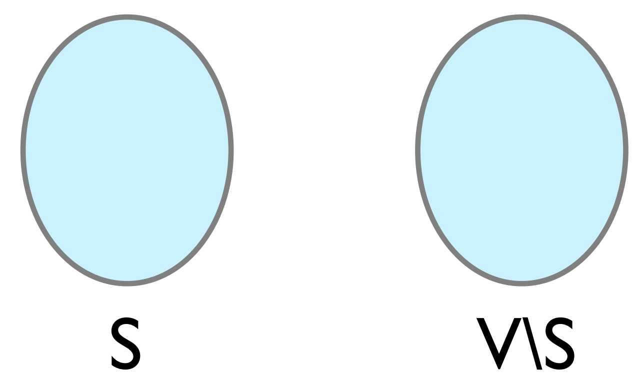
A Generic MST Algorithm

$X = \{\ \}$
While there is a cut $(S, V\backslash S)$ s.t. X has no edges across it
    $X = X + \{e\}$, where e is the lightest edge across $(S, V\backslash S)$.

Does this output a tree?

At each step, no cycle is created.

Continues while there are disconnected components.



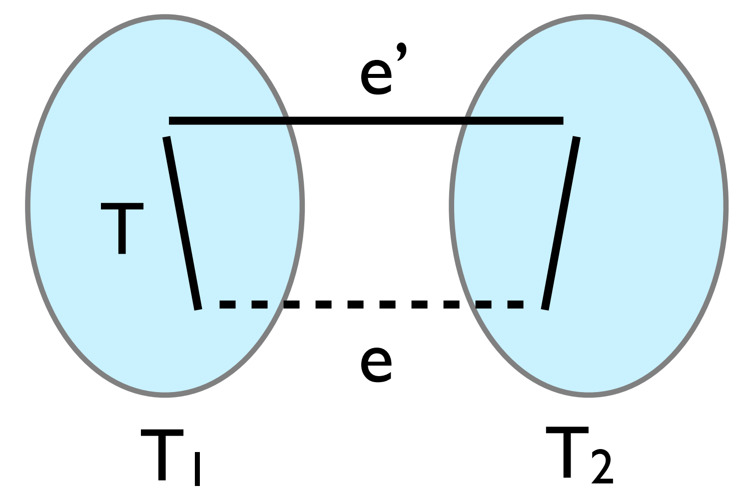
Why does this produce a MST?

A Generic MST Algorithm

X = { }
While there is a cut (S, V\S) s.t. X has no edges across it
    X = X + {e}, where e is the lightest edge across (S,V\S).

Proof of correctness by induction.

Base Case: At t=0, X is in some MST T.



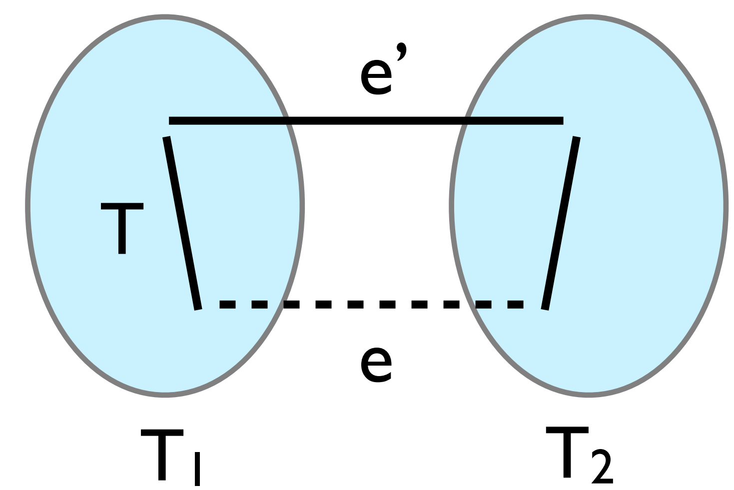
A Generic MST Algorithm

X = { }
While there is a cut (S, V\S) s.t. X has no edges across it
    X = X + {e}, where e is the lightest edge across (S,V\S).

Proof of correctness by induction.

Base Case: At t=0, X is in some MST T.

Induction: Assume at t=k, X is in some MST T.



A Generic MST Algorithm

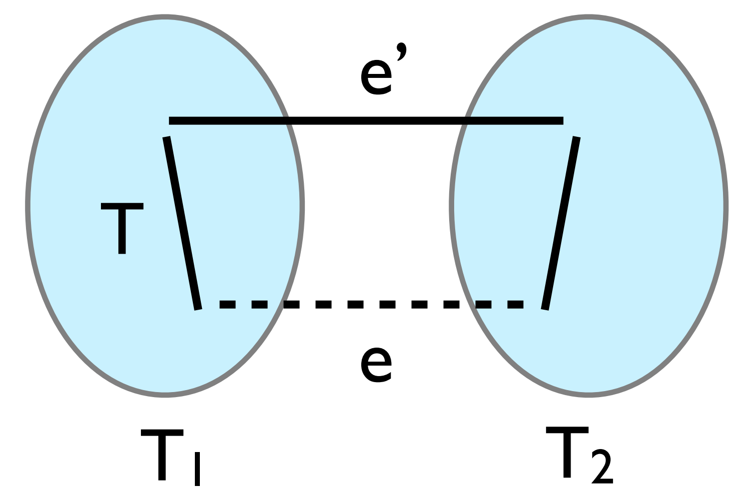
X = { }
While there is a cut (S, V\S) s.t. X has no edges across it
    X = X + {e}, where e is the lightest edge across (S,V\S).

Proof of correctness by induction.

Base Case: At t=0, X is in some MST T.

Induction: Assume at t=k, X is in some MST T.

Suppose we add e to X at t=k+1.



A Generic MST Algorithm

X = { }
While there is a cut (S, V\S) s.t. X has no edges across it
    X = X + {e}, where e is the lightest edge across (S,V\S).

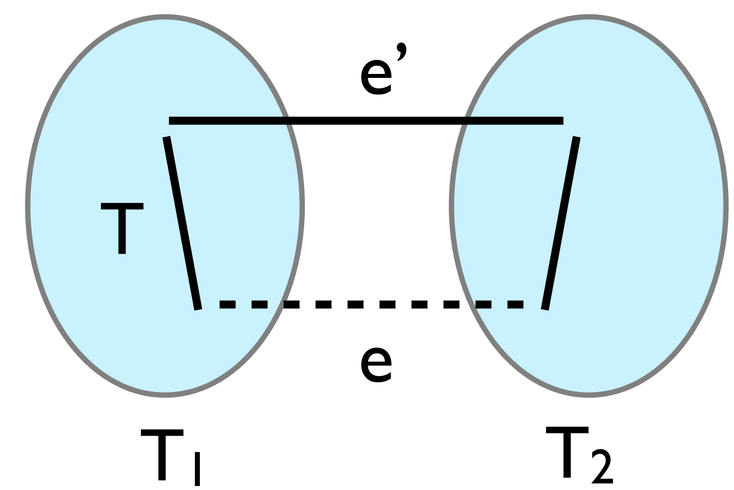
Proof of correctness by induction.

Base Case: At t=0, X is in some MST T.

Induction: Assume at t=k, X is in some MST T.

Suppose we add e to X at t=k+1.

Suppose e is not in T. Adding e to T forms a cycle C.

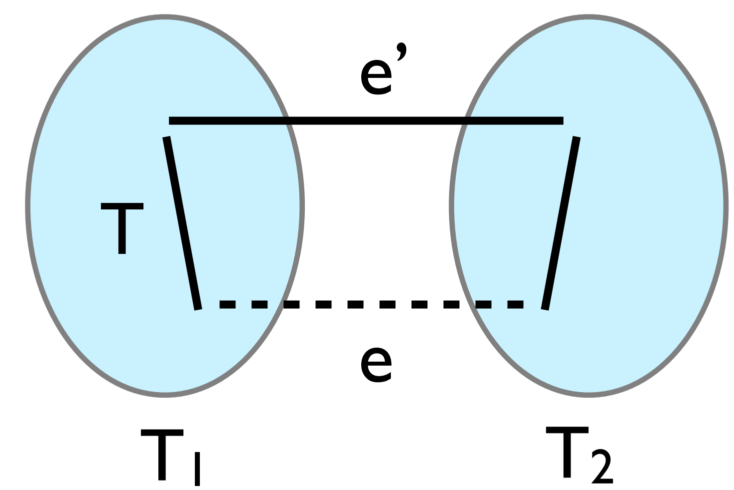


A Generic MST Algorithm

X = { }
While there is a cut (S, V\S) s.t. X has no edges across it
    X = X + {e}, where e is the lightest edge across (S,V\S).

Proof of correctness by induction.

Base Case: At t=0, X is in some MST T.



Induction: Assume at t=k, X is in some MST T.

Suppose we add e to X at t=k+1.

Suppose e is not in T. Adding e to T forms a cycle C.

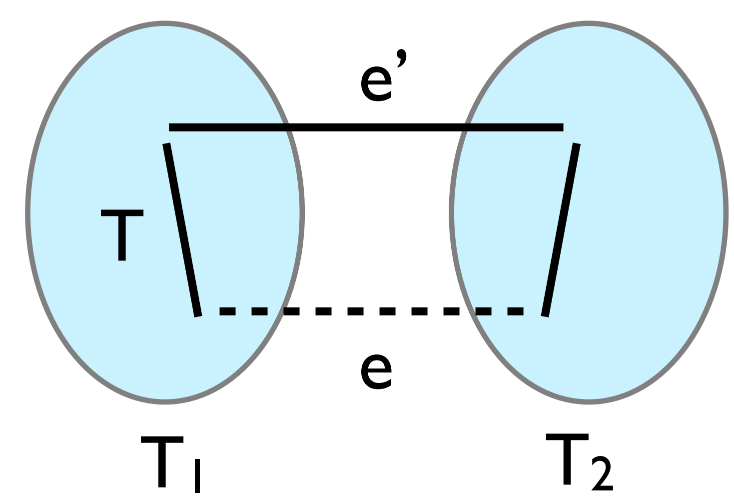
Let e' = another edge in C across (S,V\S), T' = T \ {e'} U {e}.

A Generic MST Algorithm

X = { }
While there is a cut (S, V\S) s.t. X has no edges across it
  X = X + {e}, where e is the lightest edge across (S,V\S).

Proof of correctness by induction.

Base Case: At t=0, X is in some MST T.



Induction: Assume at t=k, X is in some MST T.
Suppose we add e to X at t=k+1.
Suppose e is not in T. Adding e to T forms a cycle C.
Let e' = another edge in C across (S,V\S), T' = T \ {e'} U {e}.
cost(T') = cost(T) - cost(e') + cost(e) <= cost(T).

Kruskal's Algorithm

X = { }
For each edge e in **increasing order** of weight:
    If the end-points of e lie in different components in X,
    Add e to X

Why does this work **correctly**?

Kruskal's Algorithm

X = { }
For each edge e in **increasing order** of weight:
    If the end-points of e lie in different components in X,
    Add e to X

Why does this work **correctly**?

Efficient Implementation: Need a data structure with properties:

- Maintain disjoint sets of nodes
- Merge sets of nodes (union)
- Find if two nodes are in the same set (find)

The Union-Find data structure

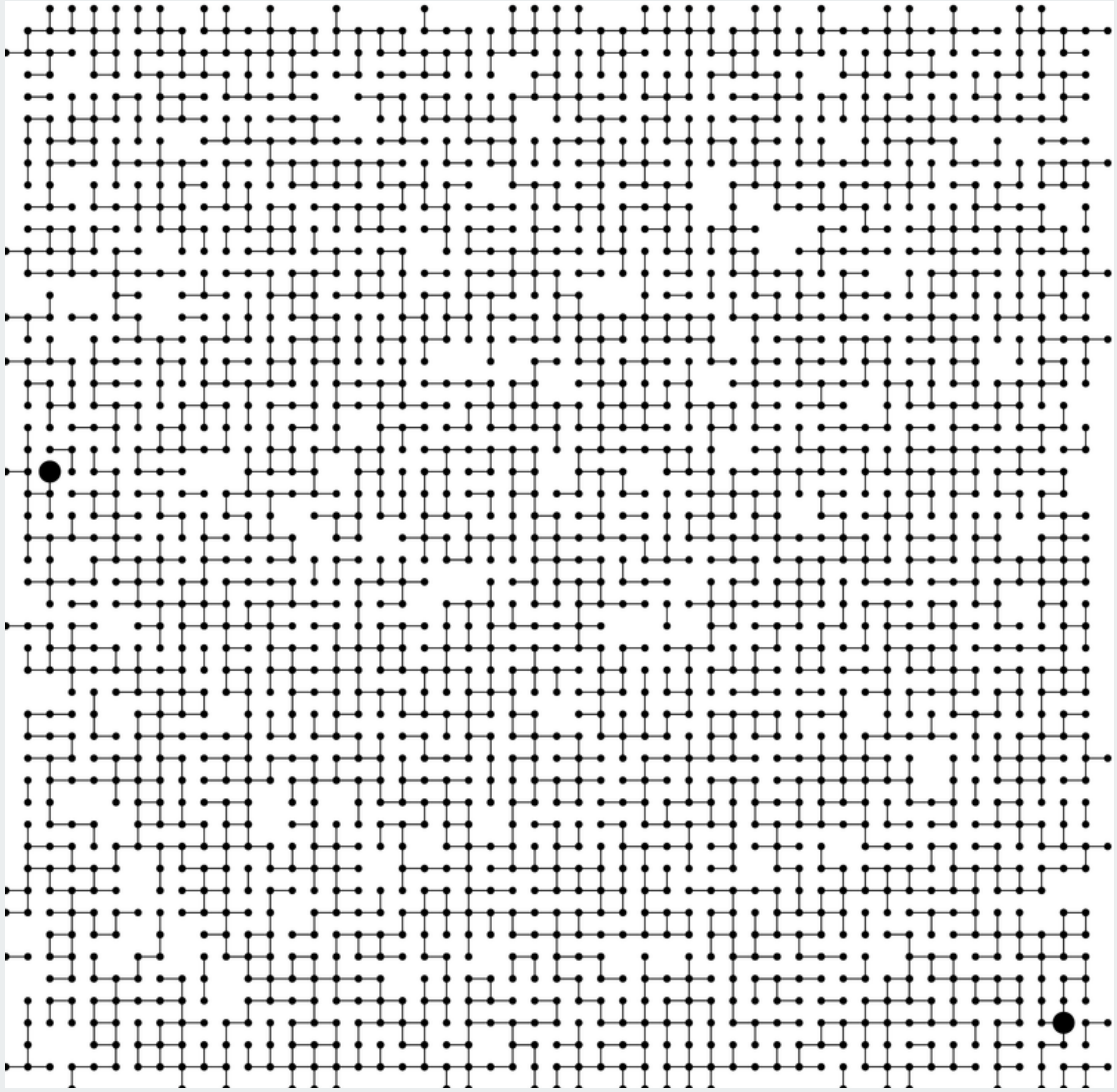
Union-Find Algorithms

- network connectivity
- quick find
- quick union
- improvements
- applications

Network connectivity

Basic abstractions

- set of objects/nodes
- union command: merge two sets
- find query: is there a path connecting one object to another?



Objects

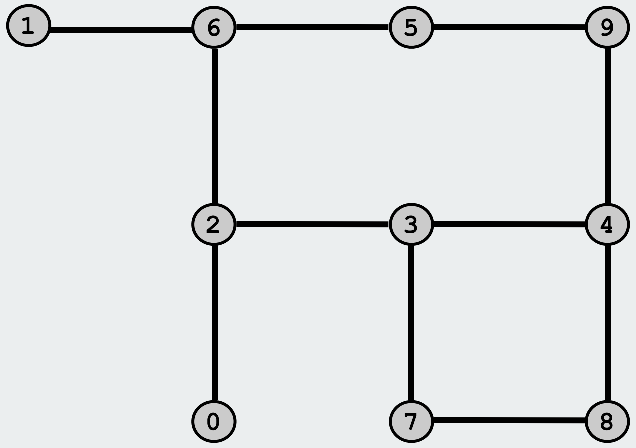
Union-find applications involve manipulating objects of all types.

- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Variable name aliases.
- Pixels in a digital photo.
- Metallic sites in a composite system.

When programming, convenient to name them 0 to N-1.

- Hide details not relevant to union-find.
- Integers allow quick access to object-related info.
- Could use symbol table to translate from object names

use as array index



Union-find abstractions

Simple model captures the essential nature of connectivity.

- Objects.

```
0  1  2  3  4  5  6  7  8  9
```

grid points

- Disjoint sets of objects.

```
0  1  { 2 3 9 }  { 5 6 }  7  { 4 8 }
```

subsets of connected grid points

- Find query: are objects 2 and 9 in the same set?

```
0  1  { 2 3 9 }  { 5 6 }  7  { 4 8 }
```

are two grid points connected?

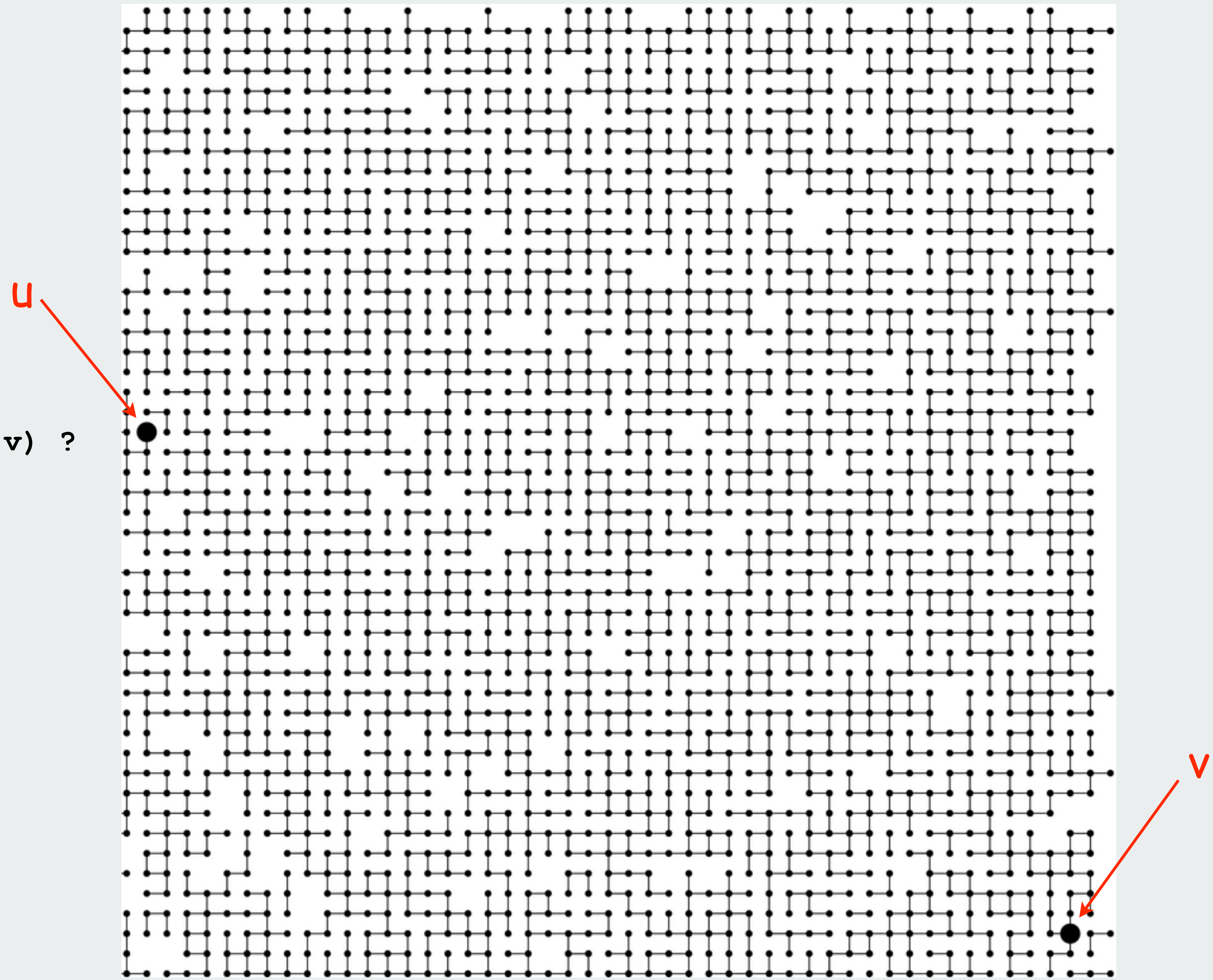
- Union command: merge sets containing 3 and 8.

```
0  1  { 2 3 4 8 9 }  { 5 6 }  7
```

add a connection between two grid points

Network connectivity: larger example

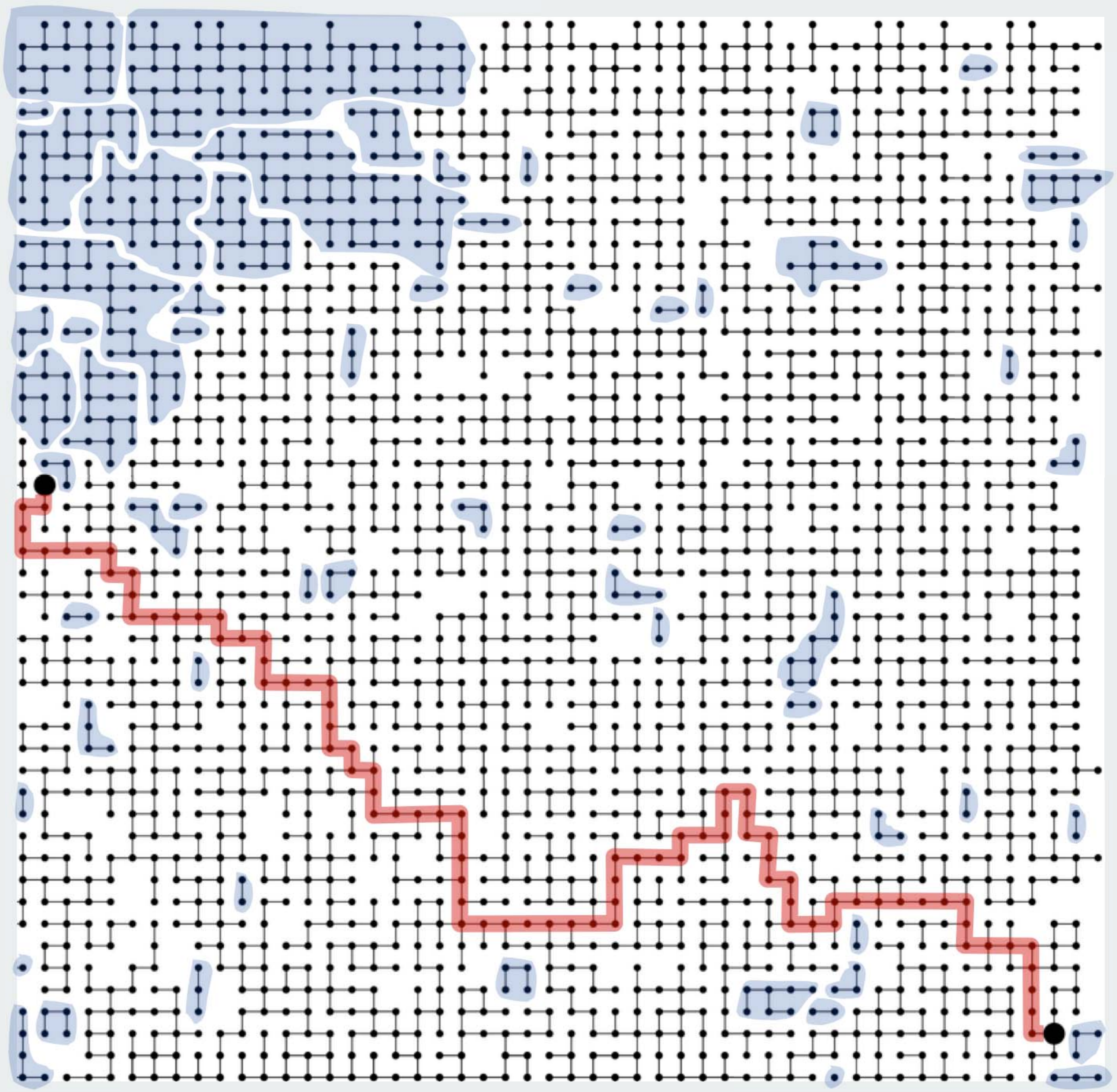
```
find(u, v) ?
```



Network connectivity: larger example

find(u, v) ?

true



63 components

- network connectivity
- quick find
- quick union
- improvements
- applications

Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

id[3] = 9; id[6] = 6
3 and 6 not connected

Union. To merge components containing `p` and `q`, change all entries with `id[p]` to `id[q]`.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

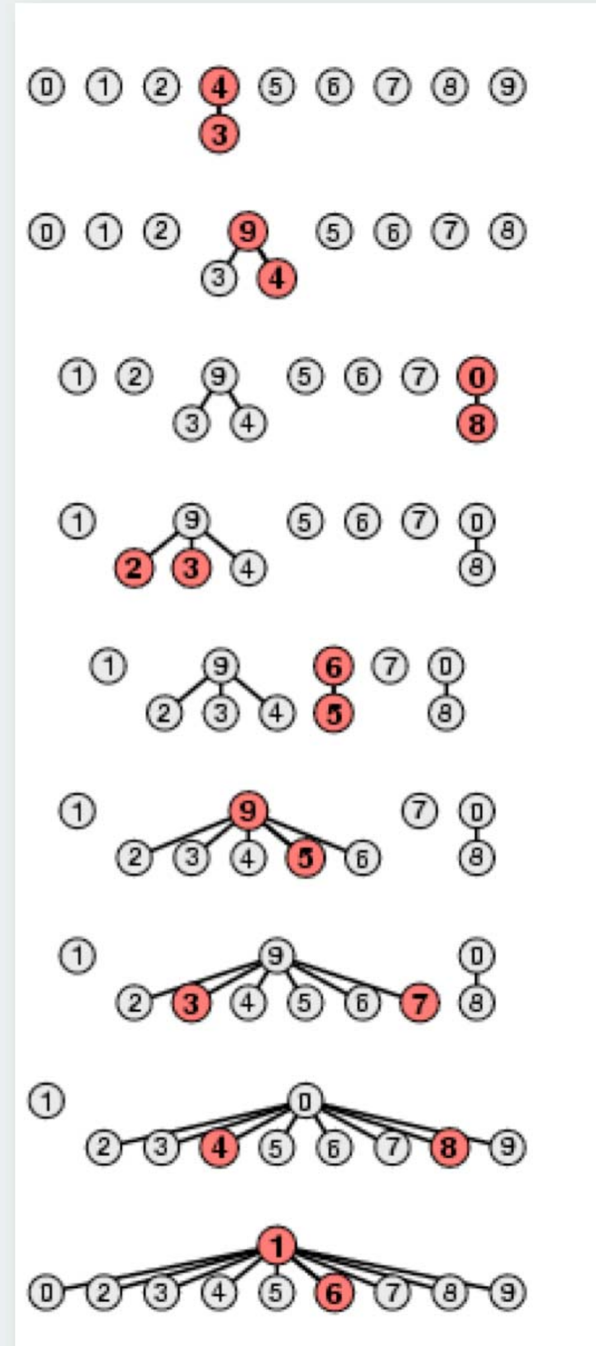
union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change

Quick-find example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	9	9	5	6	7	8	9
8-0	0	1	2	9	9	5	6	7	0	9
2-3	0	1	9	9	9	5	6	7	0	9
5-6	0	1	9	9	9	6	6	7	0	9
5-9	0	1	9	9	9	9	9	7	0	9
7-3	0	1	9	9	9	9	9	9	0	9
4-8	0	1	0	0	0	0	0	0	0	0
6-1	1	1	1	1	1	1	1	1	1	1

problem: many values can change



Quick-find is too slow

Quick-find algorithm may take ~MN steps to process M union commands on N objects

Rough standard (for now).

- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

a truism (roughly) since 1950 !

Ex. Huge problem for quick-find.

- 10^{10} edges connecting 10^9 nodes.
- Quick-find takes more than 10^{19} operations.
- 300+ years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.

- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!

- network connectivity
- quick find
- **quick union**
- improvements
- applications

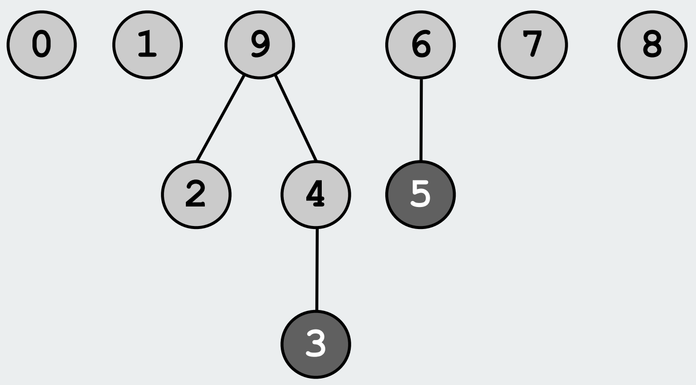
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



3's root is 9; 5's root is 6

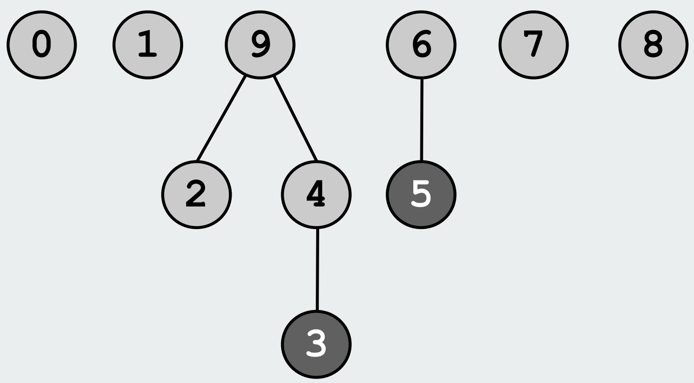
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



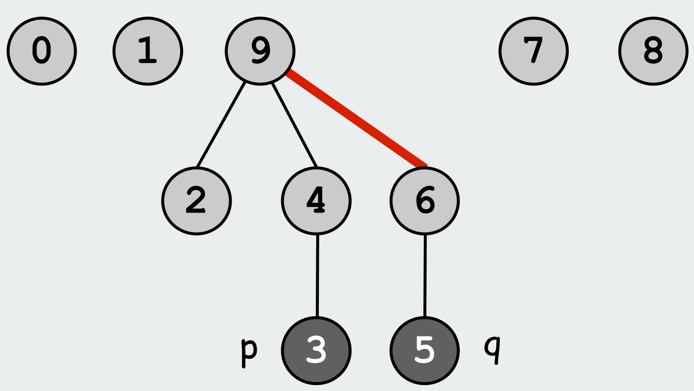
3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if p and q have the same root.

Union. Set the id of q's root to the id of p's root.

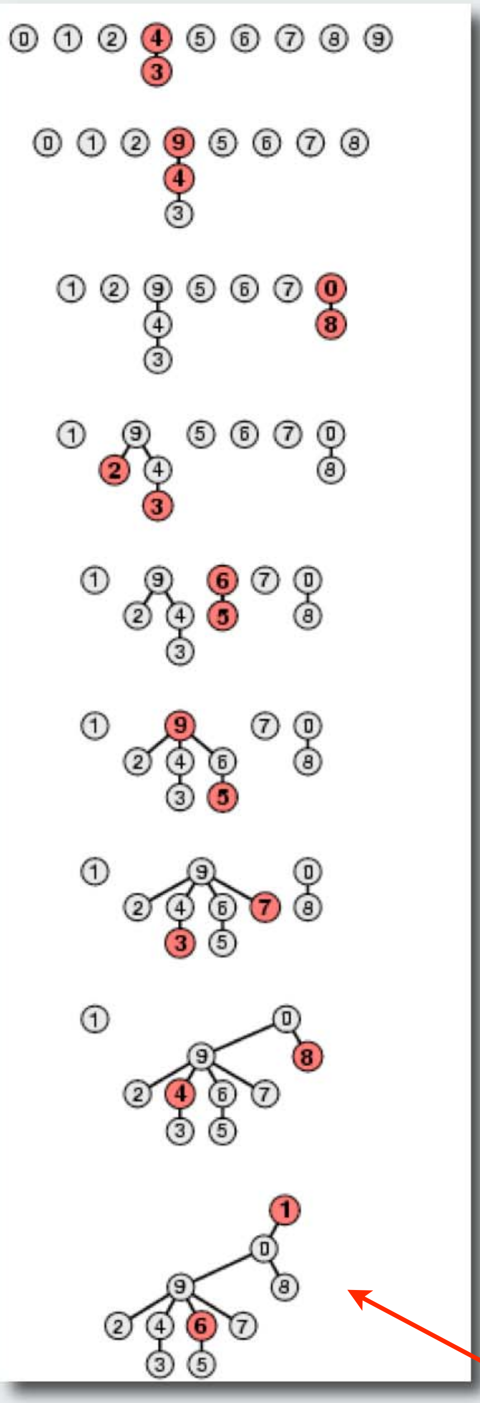
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	9	7	8	9

only one value changes



Quick-union example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



problem: trees can get tall

Quick-union is also too slow

Quick-find defect.

- Union too expensive (N steps).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N steps)
- Need to do find to do union

algorithm	union	find
Quick-find	N	1
Quick-union	N*	N ← worst case

* includes cost of find

- network connectivity
- quick find
- quick union
- **improvements**
- applications

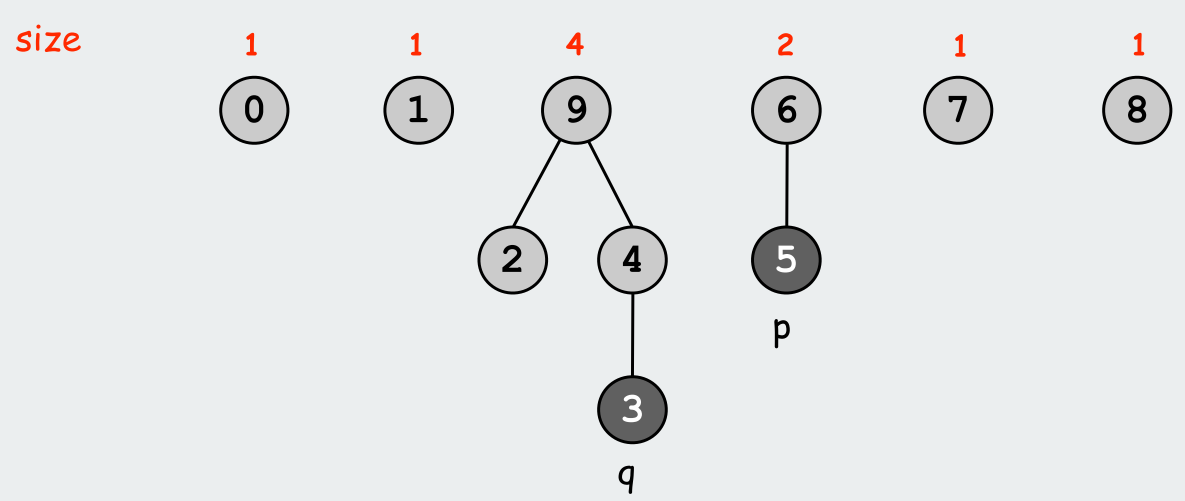
Improvement 1: Weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each component.
- Balance by linking small tree below large one.

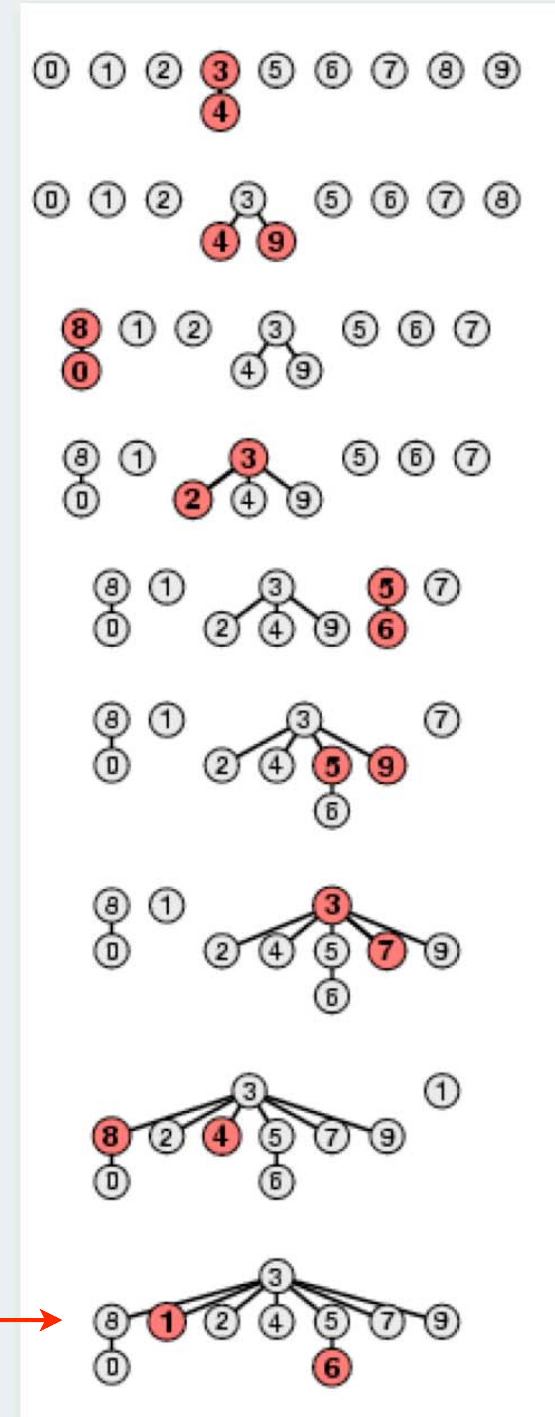
Ex. Union of 5 and 3.

- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.



Weighted quick-union example

3-4	0	1	2	3	3	5	6	7	8	9
4-9	0	1	2	3	3	5	6	7	8	3
8-0	8	1	2	3	3	5	6	7	8	3
2-3	8	1	3	3	3	5	6	7	8	3
5-6	8	1	3	3	3	5	5	7	8	3
5-9	8	1	3	3	3	3	5	7	8	3
7-3	8	1	3	3	3	3	5	3	8	3
4-8	8	1	3	3	3	3	5	3	3	3
6-1	8	3	3	3	3	3	5	3	3	3



no problem: trees stay flat

Weighted quick-union: Java implementation

Java implementation.

- Almost identical to quick-union.
- Maintain extra array `sz[]` to count number of elements in the tree rooted at i.

Find. Identical to quick-union.

Union. Modify quick-union to

- merge smaller tree into larger tree
- update the `sz[]` array.

```
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else sz[i] < sz[j] { id[j] = i; sz[i] += sz[j]; }
```

Weighted quick-union analysis

Analysis.

- Find: takes time proportional to depth of `p` and `q`.
- Union: takes constant time, given roots.
- Fact: depth is at most lg N. [needs proof]

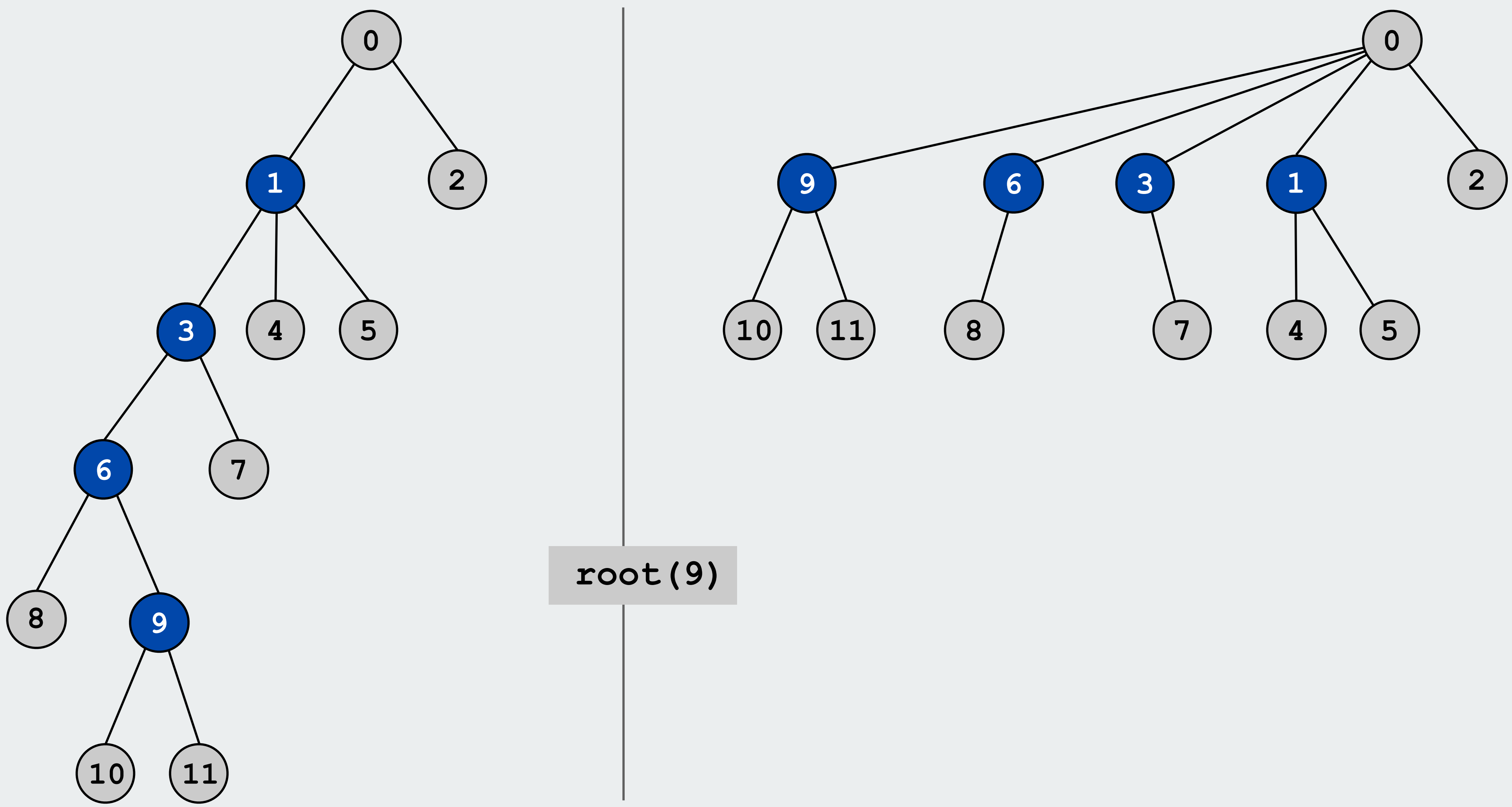
Data Structure	Union	Find
Quick-find	N	1
Quick-union	N *	N
Weighted QU	lg N *	lg N

* includes cost of find

Stop at guaranteed acceptable performance? No, easy to improve further.

Improvement 2: Path compression

Path compression. Just after computing the root of `i`, set the `id` of each examined node to `root(i)`.



Weighted quick-union with path compression

Path compression.

- Standard implementation: add second loop to `root()` to set the id of each examined node to the root.
- Simpler one-pass variant: make every other node in path point to its grandparent.

```
public int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

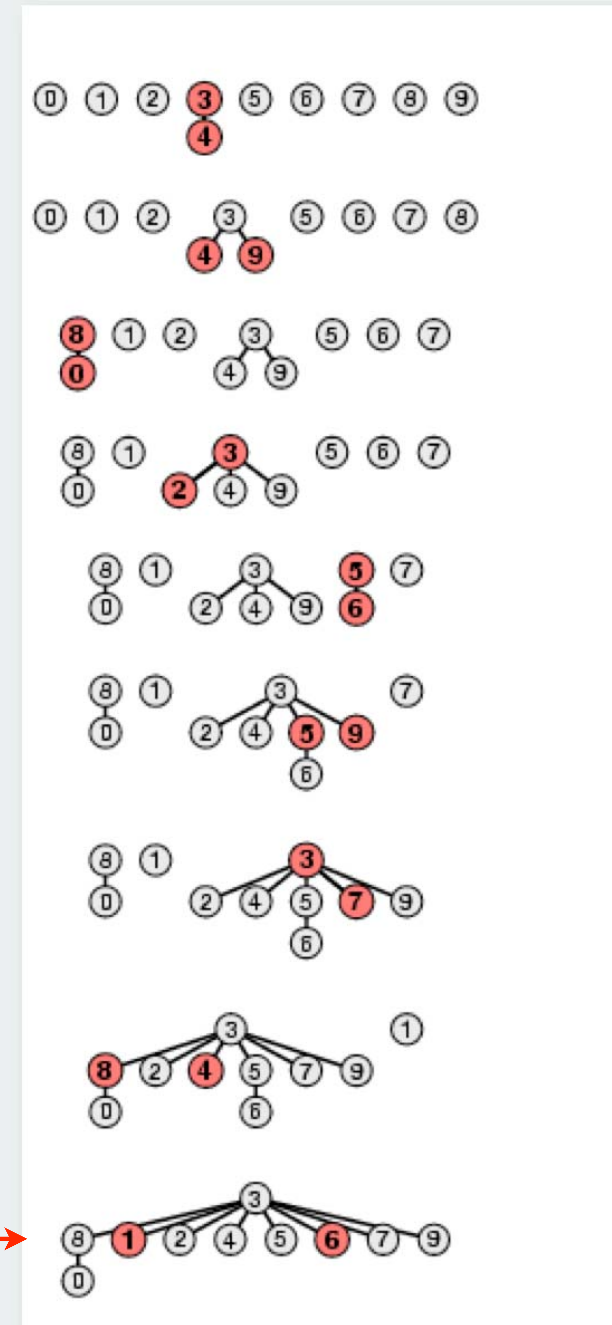
only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression

3-4	0	1	2	3	3	5	6	7	8	9
4-9	0	1	2	3	3	5	6	7	8	3
8-0	8	1	2	3	3	5	6	7	8	3
2-3	8	1	3	3	3	5	6	7	8	3
5-6	8	1	3	3	3	5	5	7	8	3
5-9	8	1	3	3	3	3	5	7	8	3
7-3	8	1	3	3	3	3	5	3	8	3
4-8	8	1	3	3	3	3	5	3	3	3
6-1	8	3	3	3	3	3	3	3	3	3

no problem: trees stay VERY flat



WQUPC performance

Theorem. Starting from an empty data structure, any sequence of M union and find operations on N objects takes O(N + M lg* N) time.

- Proof is very difficult.
- But the algorithm is still simple!

number of times needed to take the lg of a number until reaching 1

Linear algorithm?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

because lg* N is a constant in this universe

N	lg* N
1	0
2	1
4	2
16	3
65536	4
265536	5

Amazing fact:

- In theory, no linear linking strategy exists