CSE 332

AUGUST 11TH – MINUMUM SPANNING TREES

TODAY'S LECTURE

- Minimum Spanning Trees

TODAY'S LECTURE

- **Minimum Spanning Trees**
  - Prim's Algorithm (vertex based solution)

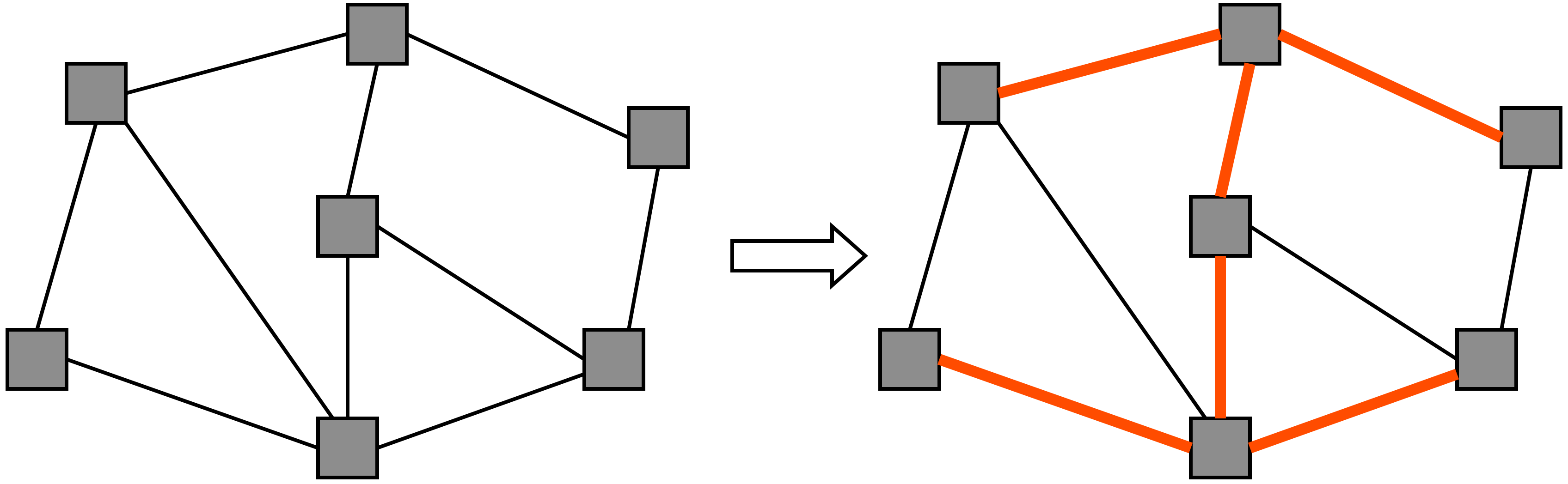
TODAY'S LECTURE

- **Minimum Spanning Trees**
  - Prim's Algorithm (vertex based solution)
  - Kruskal's Algorithm (edge based solution)

PROBLEM STATEMENT

Given a *connected* undirected graph G=(V,E), find a minimal subset of edges such that G is still connected

- A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



OBSERVATIONS

1. Problem not defined if original graph not connected. Therefore, we know |E| >= |V|-1

OBSERVATIONS

1. **Problem not defined if original graph not connected. Therefore, we know |E| >= |V|-1**

2. **Any solution to this problem is a tree**
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected

OBSERVATIONS

1. **Problem not defined if original graph not connected. Therefore, we know |E| >= |V|-1**

2. **Any solution to this problem is a tree**
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected
3. **A tree with |V| nodes has |V|-1 edges**
   - So every solution to the spanning tree problem has **|V|-1** edges

MOTIVATION

A spanning tree connects all the nodes with as few edges as possible

In most compelling uses, we have a *weighted* undirected graph and we want a tree of least total cost

Example: Electrical wiring for a house or clock wires on a chip

Example: A road network if you cared about asphalt cost rather than travel time

This is the minimum spanning tree problem

- Will do that next, after intuition from the simpler case

TWO APPROACHES

Different algorithmic approaches to the spanning-tree problem:

TWO APPROACHES

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree

TWO APPROACHES

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree

2. Iterate through edges; add to output any edge that does not create a cycle

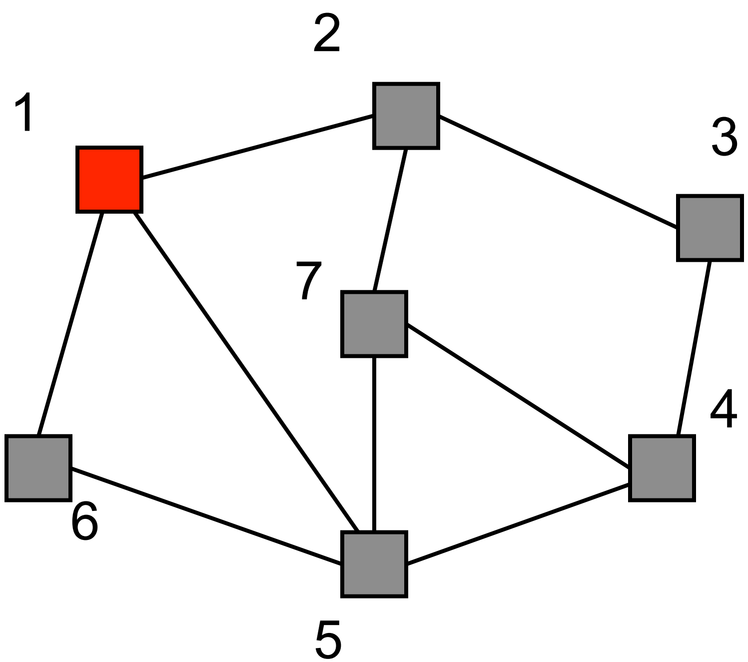
SPANNING TREE VIA DFS

```
spanning_tree(Graph G) {
  for each node v:
      v.marked = false
  dfs(someRandomStartNode)
}
dfs(Vertex a) {  // recursive DFS
  a.marked = true
  for each b adjacent to a:
    if(!b.marked) {
      add(a,b) to output
      dfs(b)
  }
}
```

Correctness: DFS reaches each node in connected graph.
We add one edge to connect it to the already visited nodes.
Order affects result, not correctness.   Runtime: ***O*(|E|)**

EXAMPLE

dfs(1)



Output:

EXAMPLE

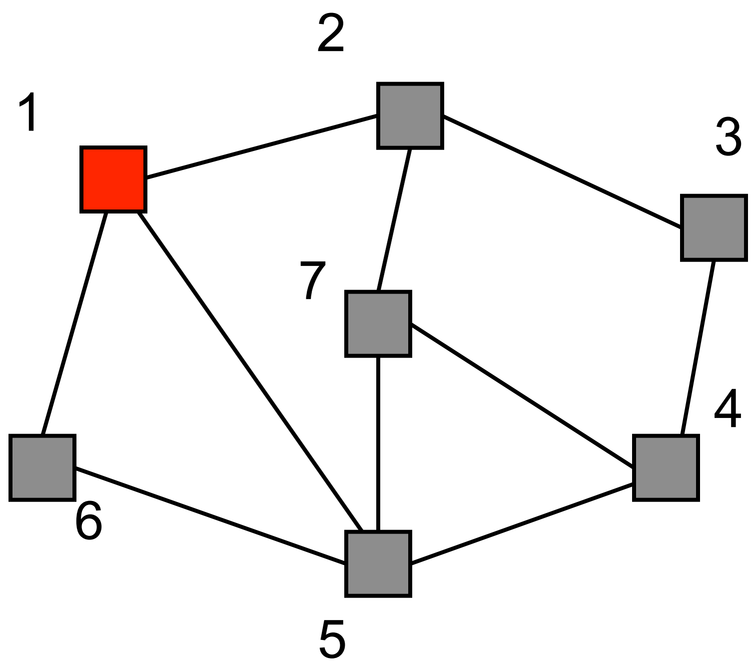
dfs(1)

Pending Callstack:

dfs(2)

dfs(5)

dfs(6)



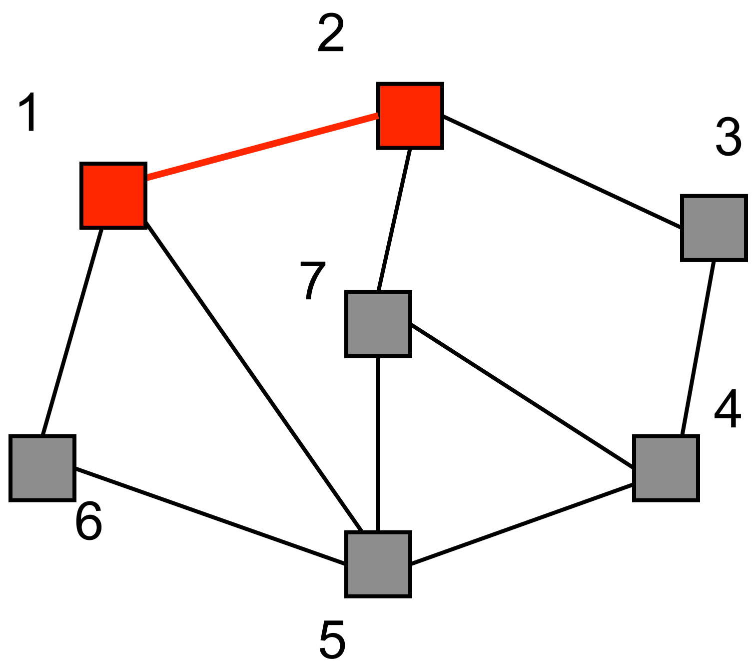
Output:

EXAMPLE

dfs(2)

Pending Callstack:

- **dfs(7)**
- **dfs(3)**
- **dfs(5)**
- **dfs(6)**



Output: (1,2)

EXAMPLE

dfs(7)

Pending Callstack:

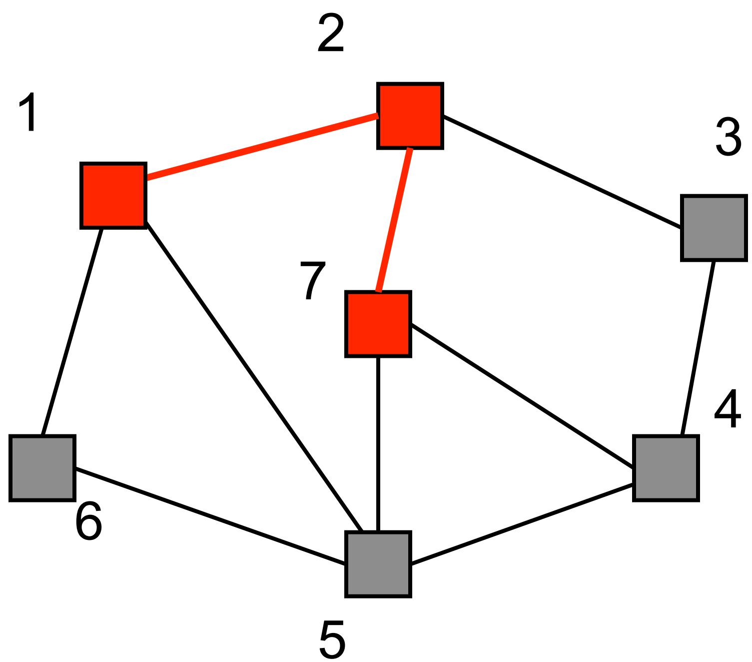
dfs(5)

dfs(4)

dfs(3)

~~dfs(5)~~

dfs(6)



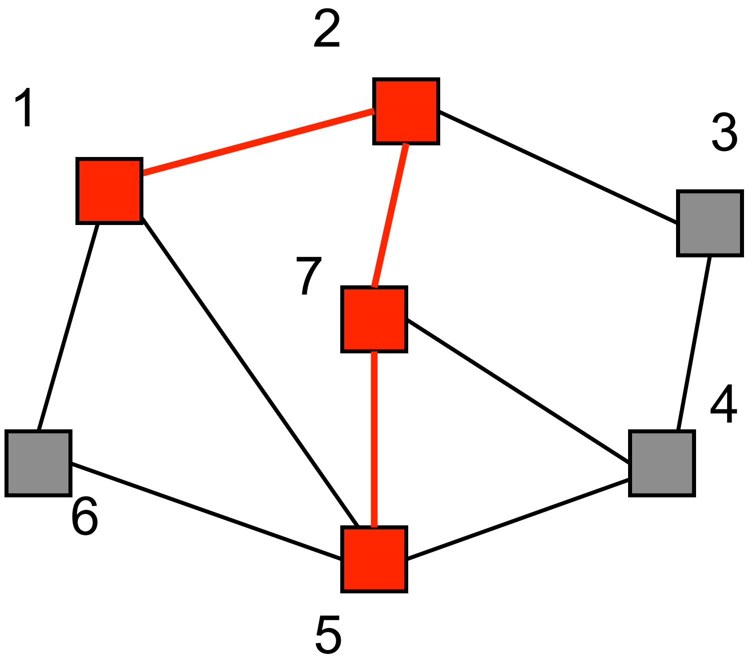
Output: (1,2), (2,7)

EXAMPLE

dfs(5)

Pending Callstack:

- **dfs(4)**
- **dfs(6)**
- **~~dfs(4)~~**
- **dfs(3)**
- **~~dfs(6)~~**



Output: (1,2), (2,7), (7,5)

EXAMPLE

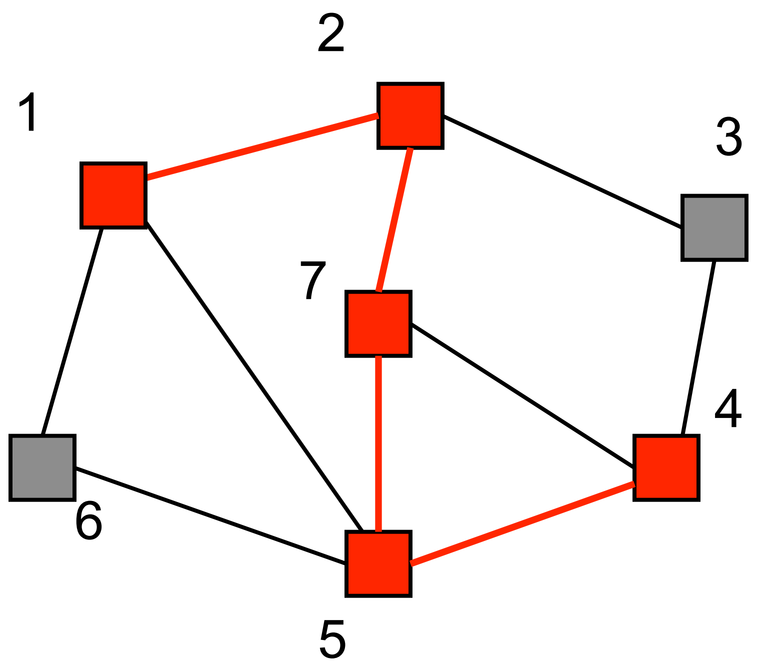
dfs(4)

Pending Callstack:

dfs(3)

dfs(6)

~~dfs(3)~~



Output: (1,2), (2,7), (7,5), (5,4)

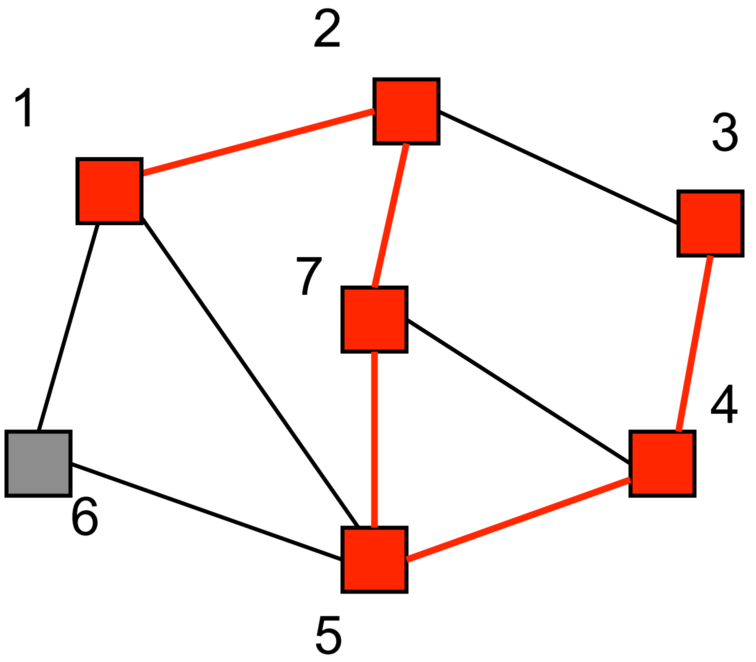
EXAMPLE

dfs(3)

Pending

Callstack:

dfs(6)



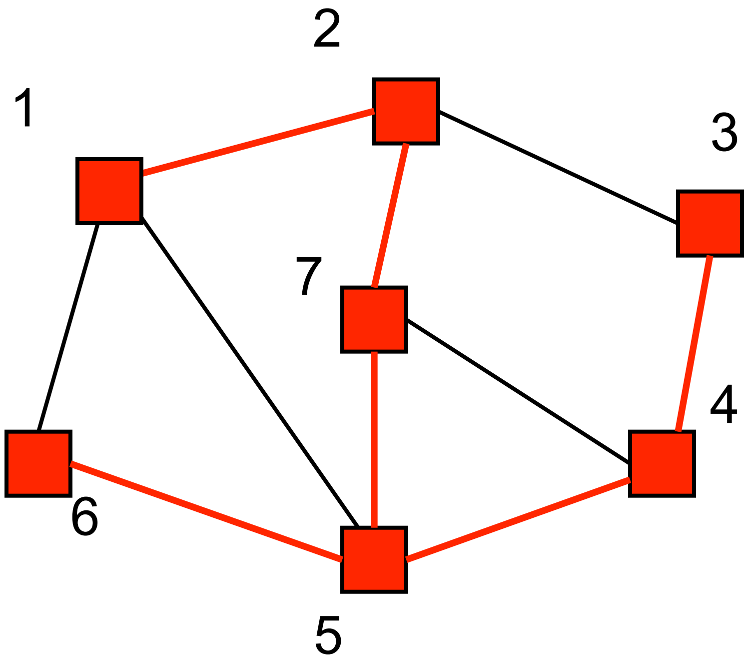
Output: (1,2), (2,7), (7,5), (5,4), (4,3)

EXAMPLE

dfs(6)

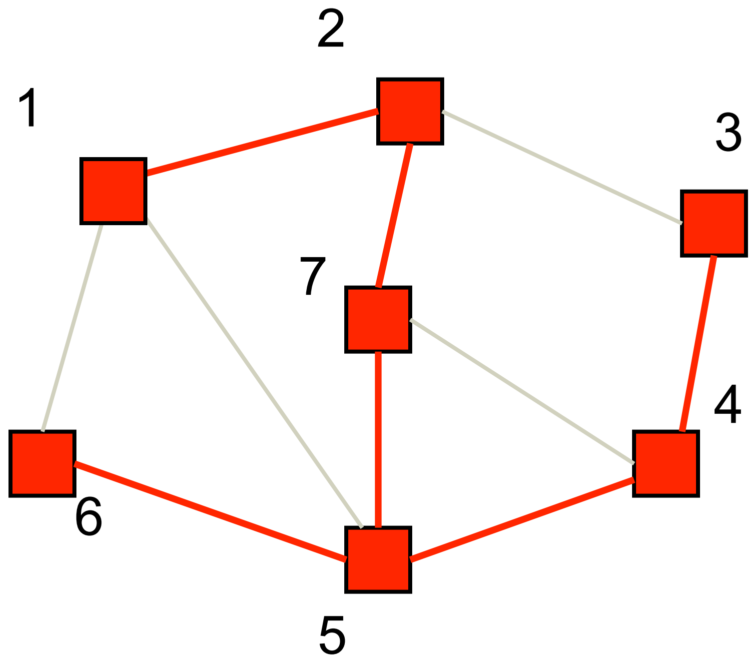
Pending

Callstack:



Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

EXAMPLE



Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

SECOND APPROACH

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):

- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree
- The graph is connected, so we reach all vertices

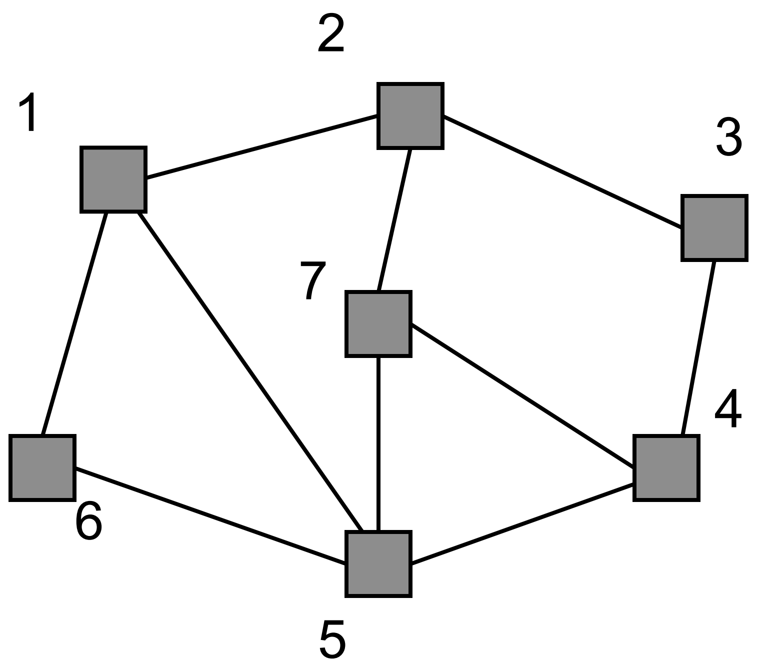
Efficiency:

- Depends on how quickly you can detect cycles
- Reconsider after the example

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

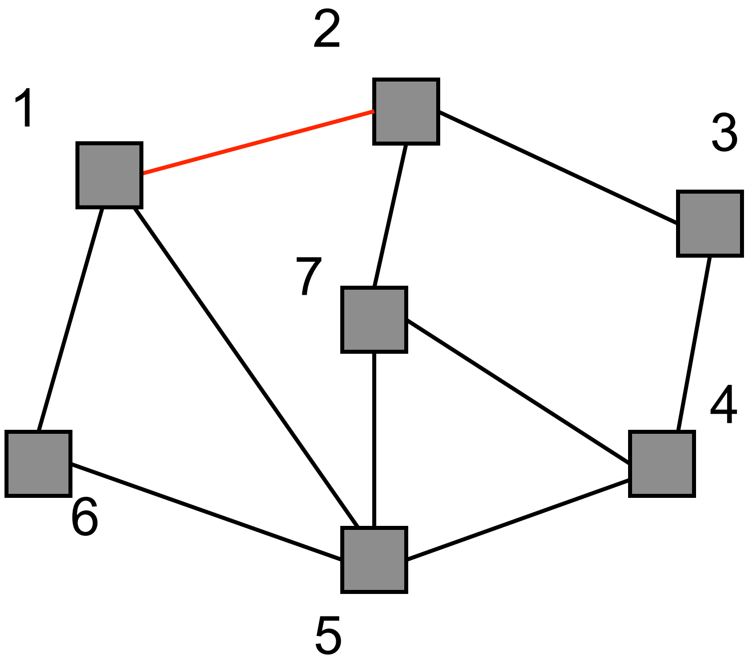


Output:

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

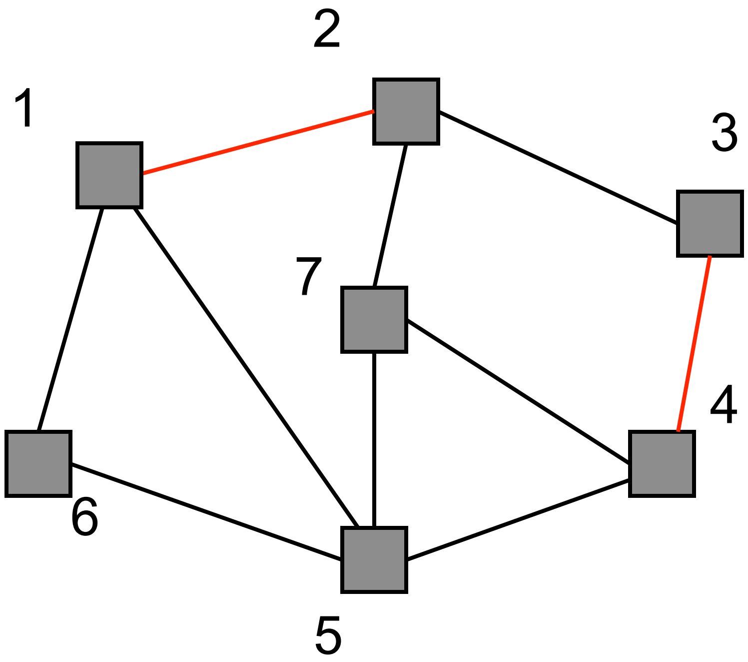


Output: (1,2)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

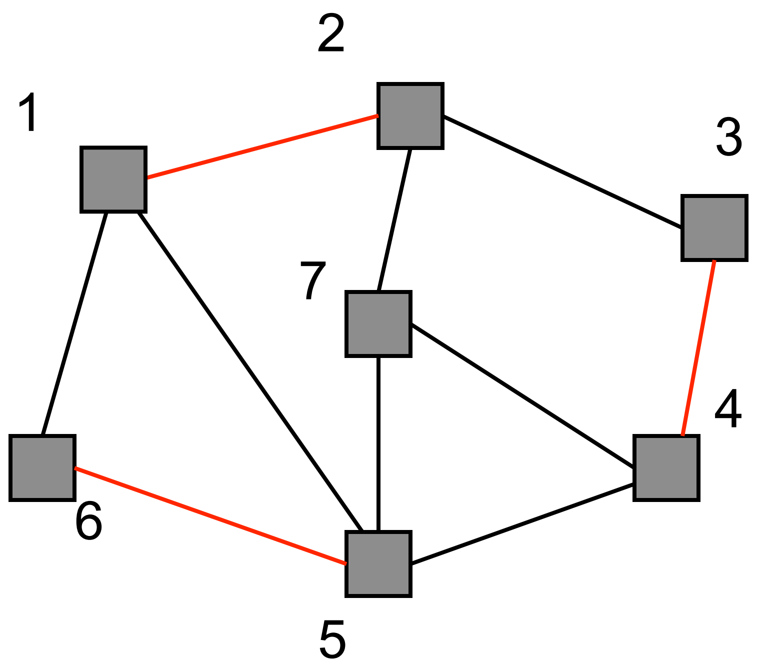


Output: (1,2), (3,4)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

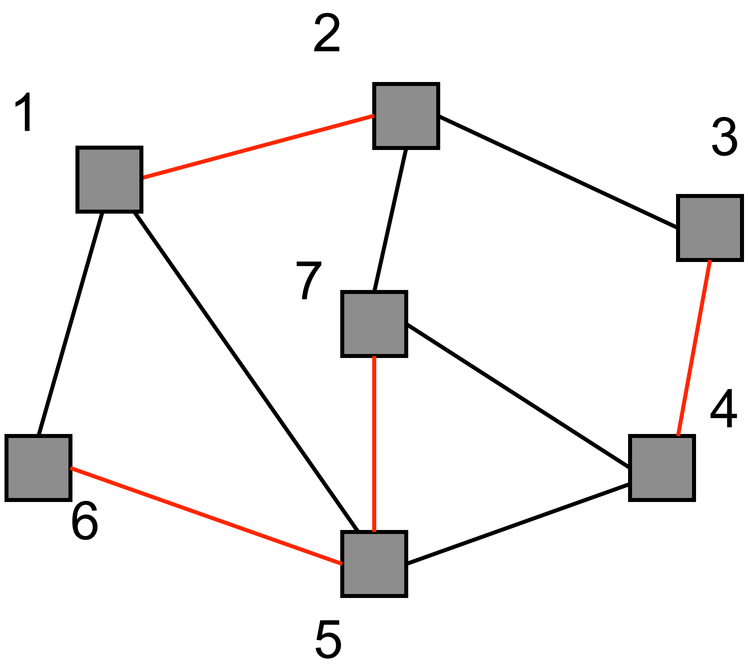


Output: (1,2), (3,4), (5,6),

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

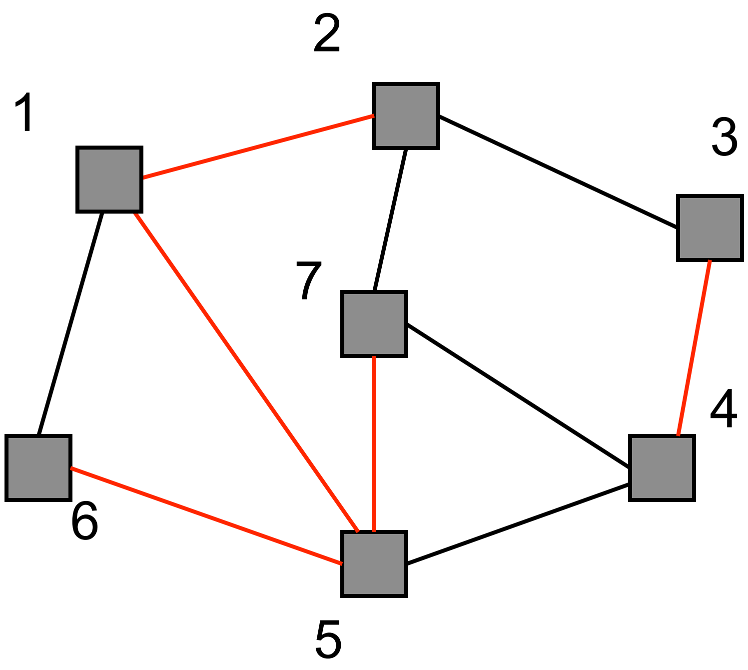


Output: (1,2), (3,4), (5,6), (5,7)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

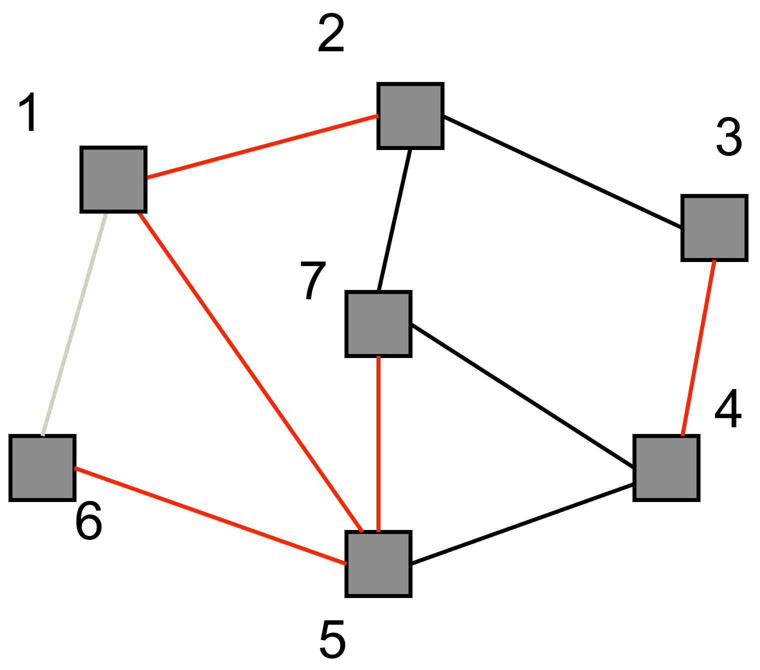


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

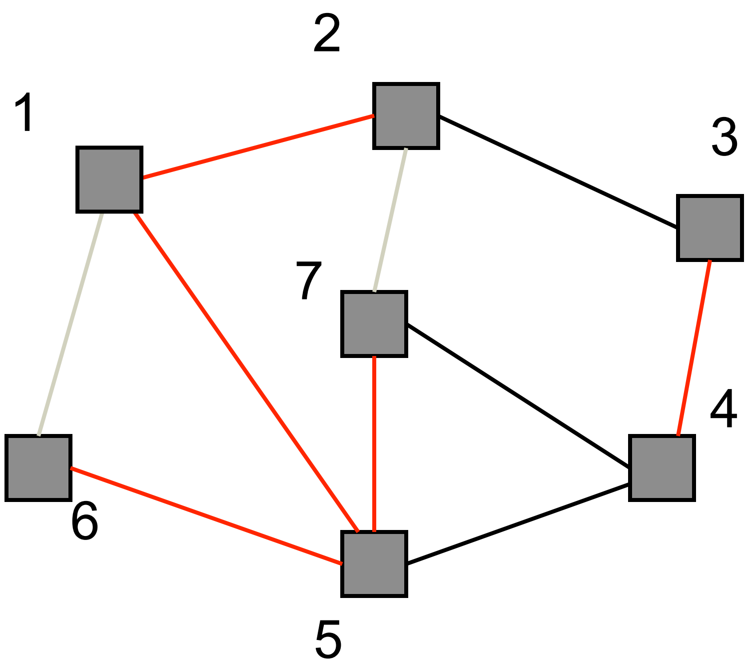


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

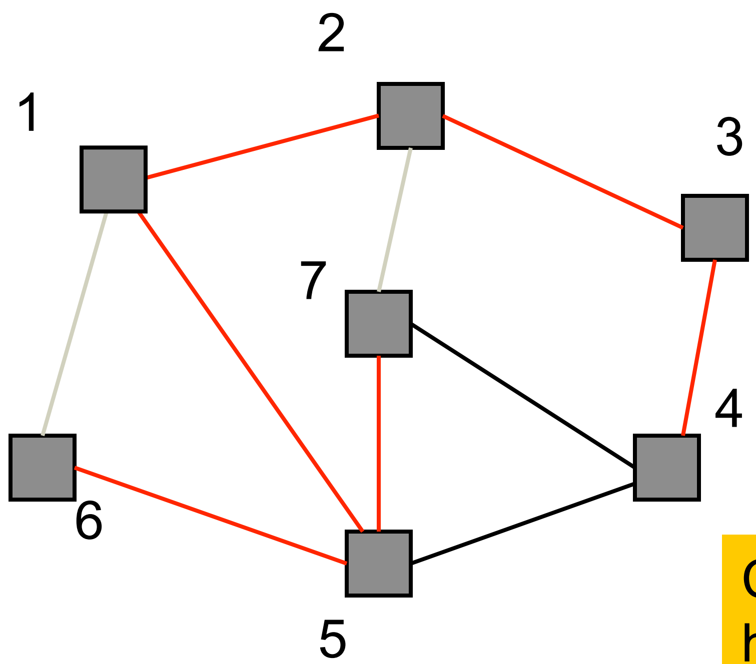


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

EXAMPLE

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Can stop once we have |V|-1 edges

Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

CYCLE DETECTION

To decide if an edge could form a cycle is *O*(|V|) because we may need to traverse all edges already in the output

So overall algorithm would be *O*(|V||E|)

But there is a faster way: union-find!

- Data structure which stores connected sub-graphs
- As we add more edges to the spanning tree, those sub-graphs are joined

DISJOINT SETS AND UNION FIND

What are sets and *disjoint sets*

The union-find ADT for disjoint sets

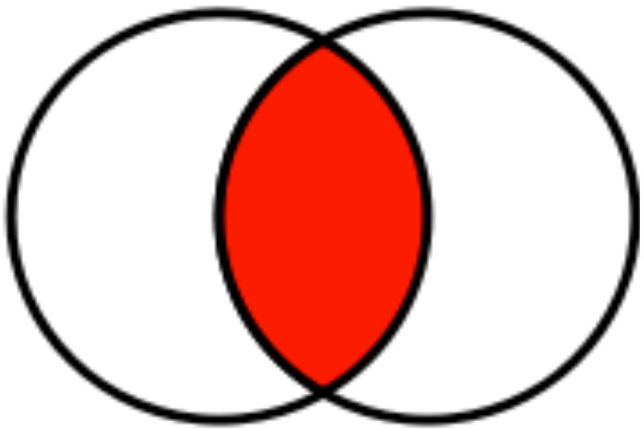
Basic implementation with "up trees"

Optimizations that make the implementation much faster

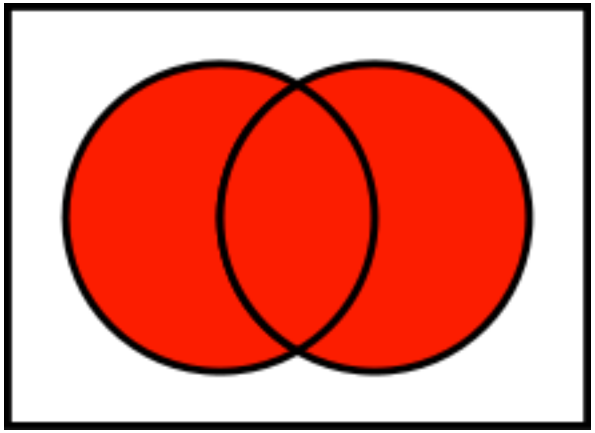
TERMINOLOGY

Empty set: ∅

Intersection ∩



Union ∪



Notation for elements in a set:

Set S containing e1, e2 and e3: **{e1, e2, el3}**

e1 is an element of S: **e1 ∈ S**

DISJOINT SETS

A set is a collection of elements (no-repeats)

Every set contains the empty set by default

Two sets are disjoint if they have no elements in common

- $S_1 \cap S_2 = \varnothing$

Examples:

- {a, e, c} and {d, b} — Disjoint
- {x, y, z} and {t, u, x} — Not disjoint

PARTITIONS

A partition *P* of a set *S* is a set of sets $\{S_1, S_2, \ldots, S_n\}$ such that every element of *S* is in exactly one S_i

Put another way:

- $S_1 \cup S_2 \cup \ldots \cup S_k = S$
- For all i and j, $i \neq j$ implies $S_i \cap S_j = \varnothing$ (sets are disjoint with each other)

Example: Let *S* be {a,b,c,d,e}

- {a}, {d,e}, {b,c} — Partition
- {a,b,c}, ∅, {d}, {e} — Partition
- {a,b,c,d,e} — Partition
- {a,b,d}, {c,d,e} — Not a partition, not disjoint, both sets have d
- {a,b}, {e,c} — Not a partition of S (doesn't have d)

UNION FIND ADT: OPERATIONS

Given an unchanging set *S*, `create` an initial partition of a set

- Typically each item in its own subset: {a}, {b}, {c}, …
- Give each subset a "name" by choosing a *representative element*

Operation `find` takes an element of *S* and returns the representative element of the subset it is in

Operation `union` takes two subsets and (permanently) makes one larger subset

- A different partition with one fewer set
- Affects result of subsequent **`find`** operations
- Choice of representative element up to implementation

EXAMPLE

Let *S* = {1,2,3,4,5,6,7,8,9}

Let initial partition be (will highlight representative elements red)

{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}

`union`(2,5):

{1}, {2, 5}, {3}, {4}, {6}, {7}, {8}, {9}

`find`(4) = 4, `find`(2) = 2, `find`(5) = 2

`union`(4,6), `union`(2,7)

{1}, {2, 5, 7}, {3}, {4, 6}, {8}, {9}

`find`(4) = 6, `find`(2) = 2, `find`(5) = 2

`union`(2,6)

{1}, {2, 4, 5, 6, 7}, {3}, {8}, {9}

NO OTHER OPERATIONS

All that can "happen" is sets get unioned

- No "un-union" or "create new set" or …

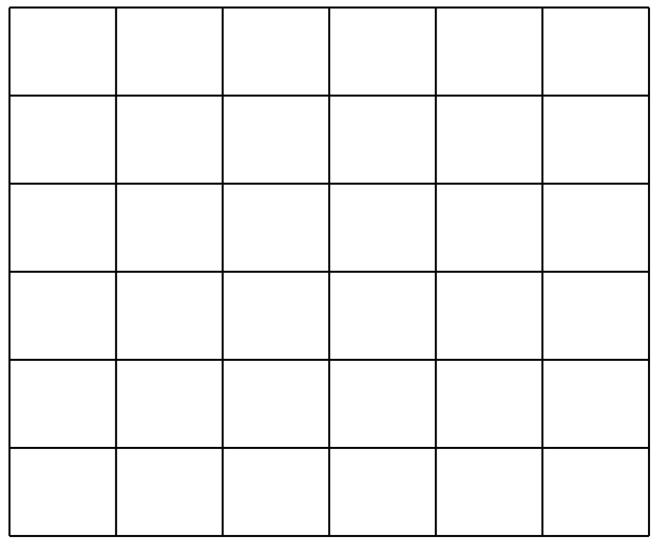
As always: trade-offs – implementations are different

- ideas? How do we maintain "representative" of a subset?

Surprisingly useful ADT, but not as common as dictionaries, priority queues / heaps, AVL trees or hashing

EXAMPLE APPLICATION: MAZE-BUILDING

Build a random maze by erasing edges

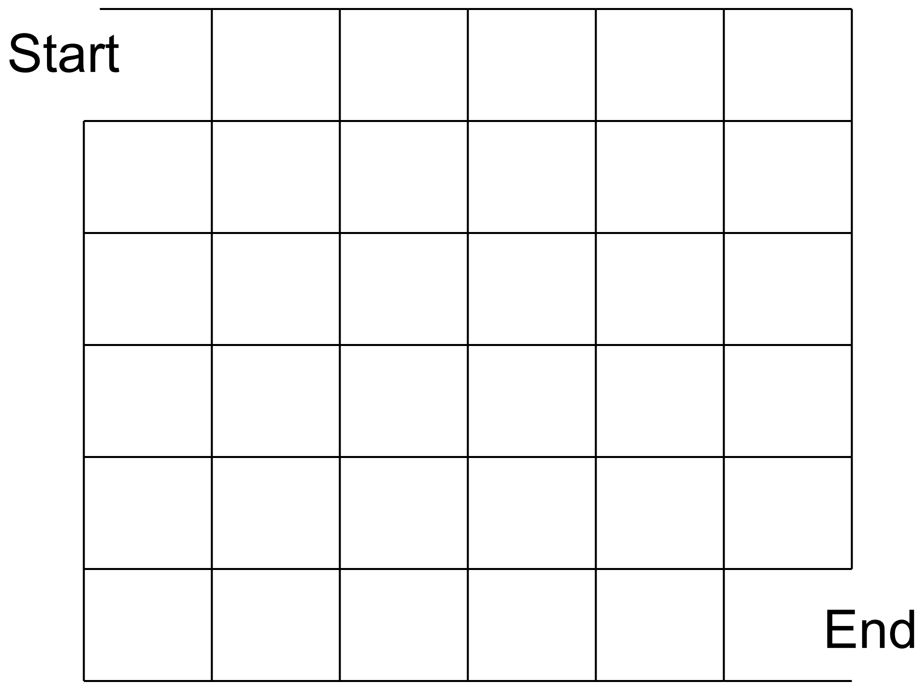


Criteria:

- Possible to get from anywhere to anywhere
- No loops possible without backtracking
    - After a "bad turn" have to "undo"

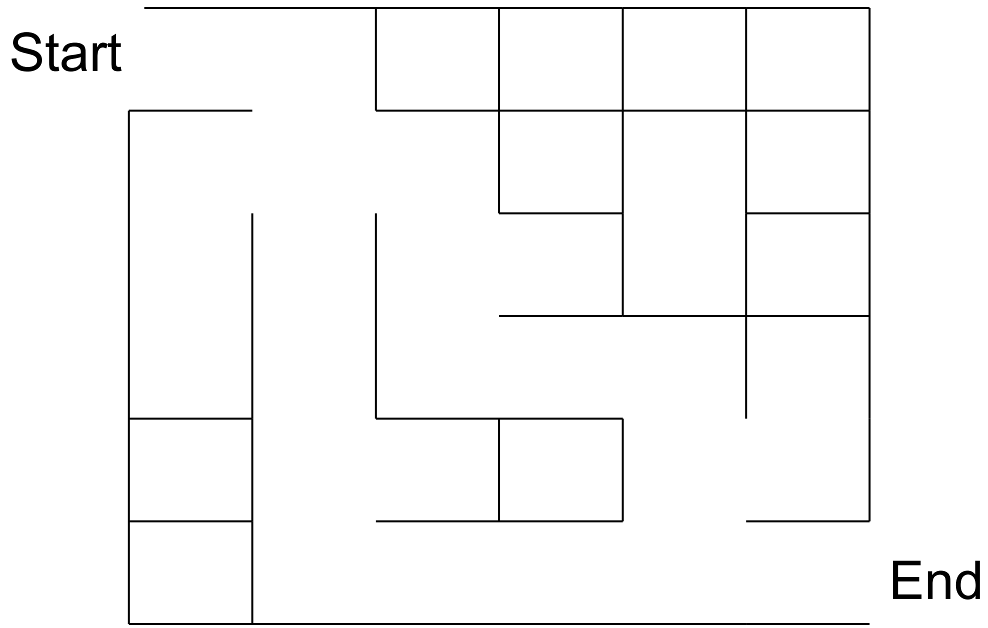
MAZE BUILDING

Pick start edge and end edge



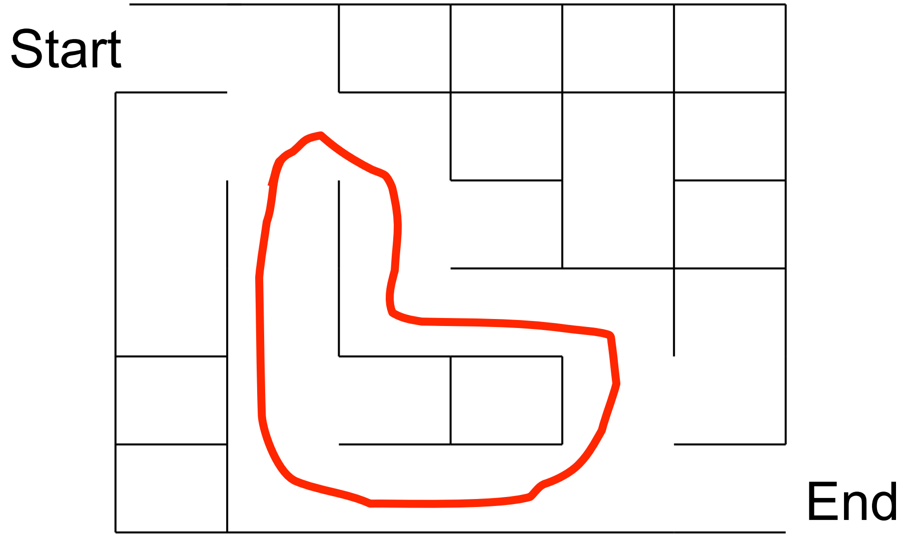
REPEATEDLY PICK RANDOM EDGES TO DELETE

One approach: just keep deleting random edges until you can get from start to finish



PROBLEMS WITH THIS APPROACH

1. **How can you tell when there is a path from start to finish?**
   - We do not really have an algorithm yet (Graphs)
2. **We have *cycles*, which a "good" maze avoids**
3. **We can't get from anywhere to anywhere else**

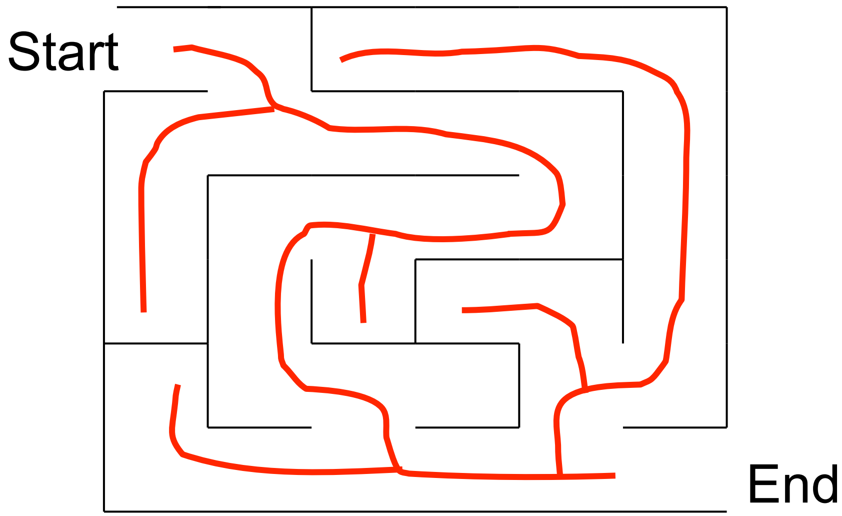


REVISED APPROACH

Consider edges in random order

But only delete them if they introduce no cycles (how? TBD)

When done, will have one way to get from any place to any other place (assuming no backtracking)



Notice the funny-looking *tree* in red

CELLS AND EDGES

Let's number each cell

- 36 total for 6 x 6

An (internal) edge (x,y) is the line between cells x and y

- 60 total for 6x6: (1,2), (2,3), …, (1,7), (2,8), …

Start

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

End

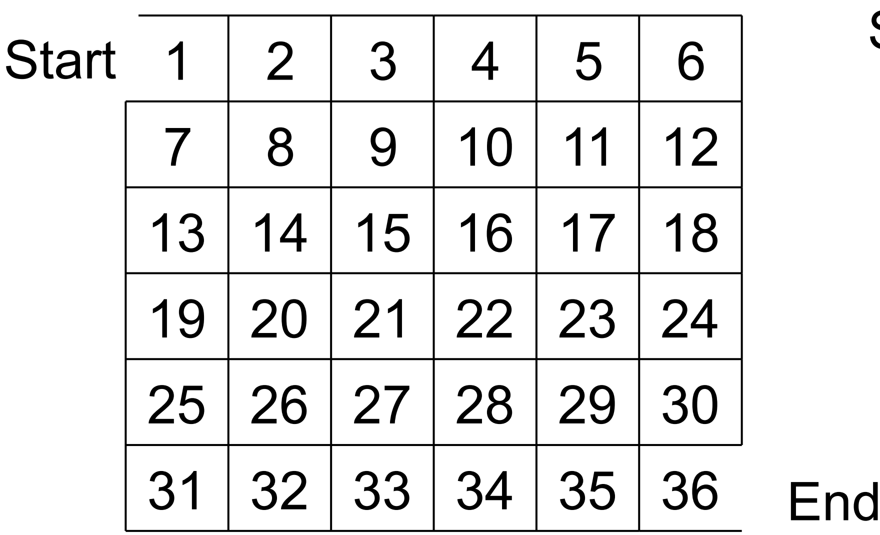
THE TRICK

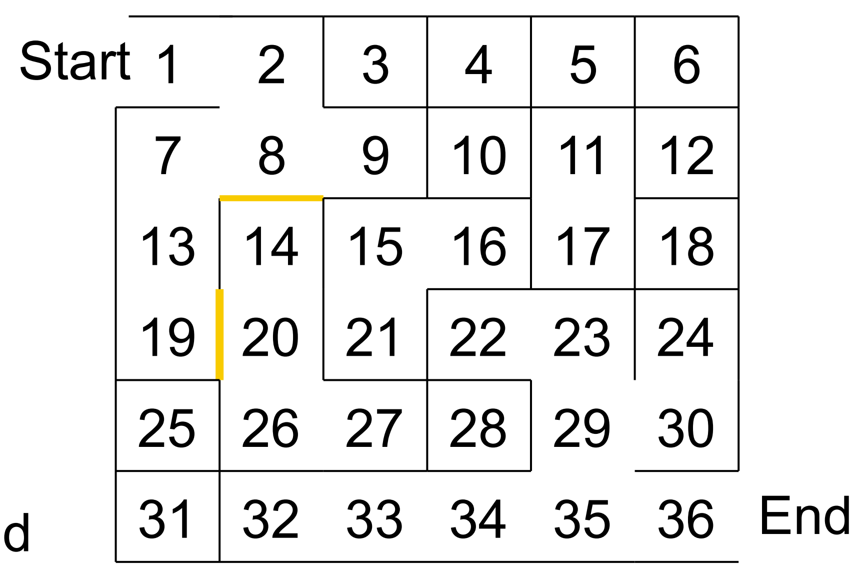
Partition the cells into disjoint sets: "are they connected"

- Initially every cell is in its own subset

If an edge would connect two different subsets:

- then remove the edge and **union** the subsets
- else leave the edge because removing it makes a cycle





IMPLEMENTATION?

How do you store a subset?

How do you know what the “representative” is?

How do you implement union?

How do you pick a new “representative”?

What is the cost of find? Of union? Of create?

IMPLEMENTATION

Start with an initial partition of *n* subsets

- Often 1-element sets, e.g., {1}, {2}, {3}, …, {*n*}

May have *m* `find` operations and up to *n*-1 `union` operations in any order

- After *n*-1 `union` operations, every `find` returns same 1 set

If total for all these operations is *O*(*m*+*n*), then average over the runs is *O*(1)

- We will get very, very close to this
- *O*(1) worst-case is impossible for `find` *and* `union`
  - Trivial for one *or* the other

UP-TREE DATA STRUCTURE

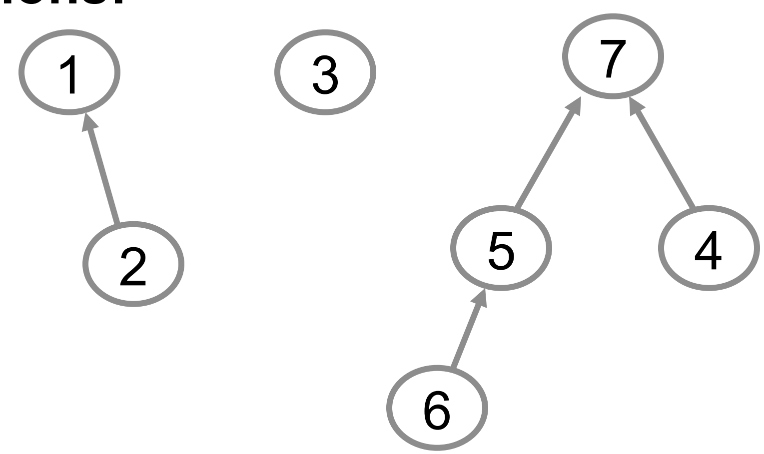
Tree with any number of children at each node

- References from children to parent (each child knows who it's parent is)

Start with *forest (collection of trees)* of 1-node trees

Possible forest after several unions:

- Will use overall roots for the representative element

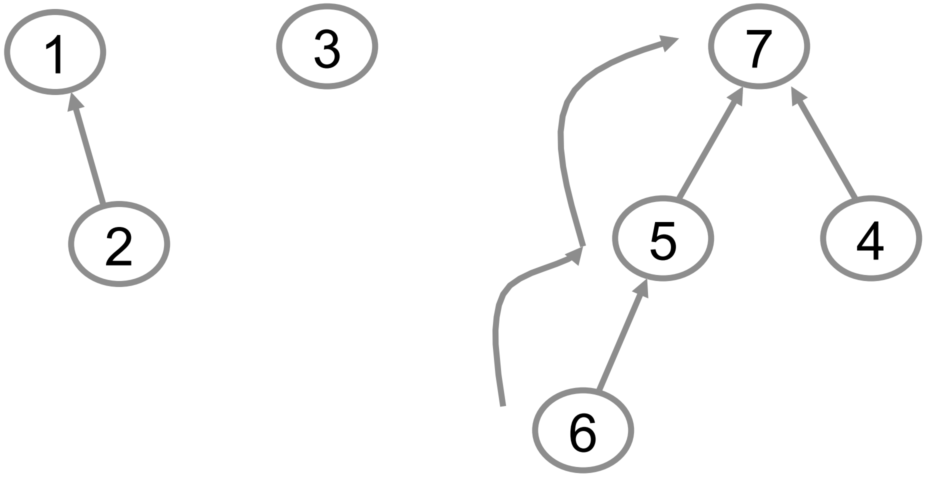


FIND

`find(x)`: (backwards from the tree traversals we've been doing for find so far)

- ***Assume*** we have *O*(1) access to each node
- **Start at `x` and follow parent pointers to root**
- **Return the root**

`find`(6) = 7

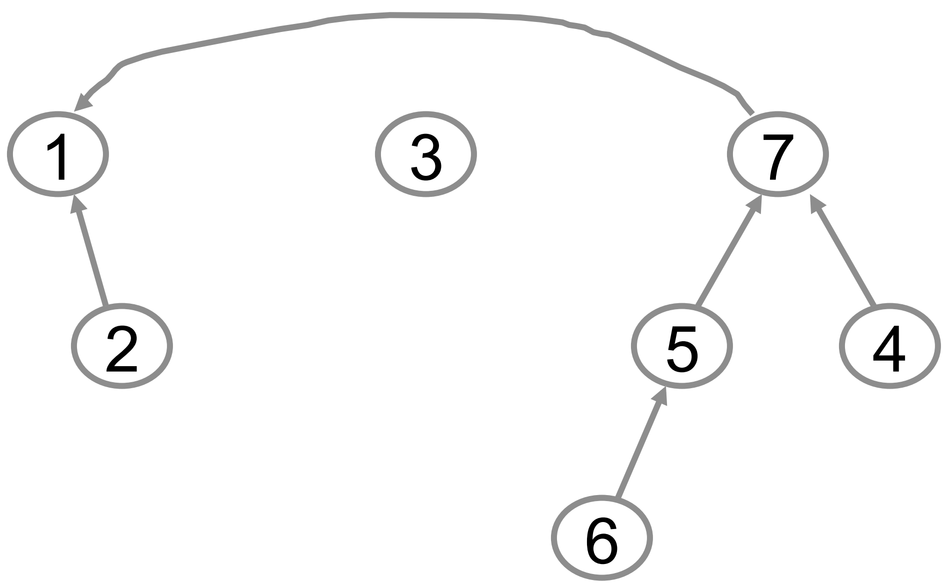


UNION

`union(x,y)`:

- Find the roots of **`x`** and **`y`**
- if distinct trees, we merge, if the same tree, do nothing
- Change root of one to have parent be the root of the other

`union`(1,7)



REPRESENTATION

Important to remember from the operations:

- We assume O(1) access to *each* node
- Ideally, we want the traversal from leaf to root of each tree to be as short as possible (the find operation depends on this traversal)
- We don't want to copy a bunch of nodes to a new tree on each union, we only want to modify one pointer (or a small constant number of them)

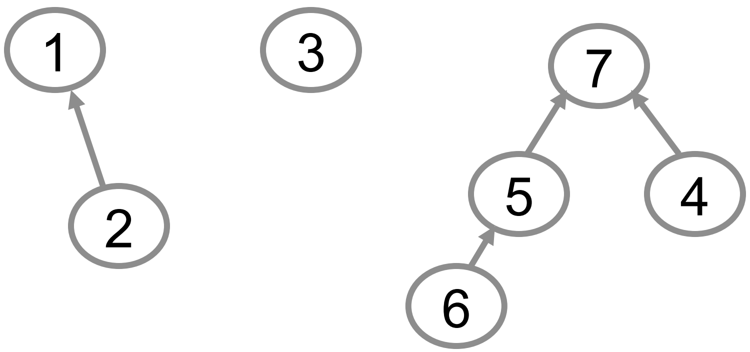
SIMPLE IMPLEMENTATION

If set elements are contiguous numbers (e.g., 1,2,…,*n*), use an array of length *n* called `up`

- Starting at index 1 on slides
- Put in array index of parent, with 0 (or -1, etc.) for a root

Example:

	1	2	3	4	5	6	7
up	0	0	0	0	0	0	0



	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0

If set elements are not contiguous numbers, could have a separate dictionary hash map to map elements (keys) to numbers (values)

IMPLEMENT OPERATIONS

```
// assumes x in range 1,n
int find(int x) {
  while(up[x] != 0) {
    x = up[x];
  }
  return x;
}
```

```
// assumes x,y are roots
void union(int x, int y){
  // y = find(y)
  // x = find(x)
  up[y] = x;
}
```

Worst-case run-time for `union`?

Worst-case run-time for `find`?

Worst-case run-time for *m* `finds` and *n*-1 `unions`?

IMPLEMENT OPERATIONS

```
// assumes x in range 1,n
int find(int x) {
  while(up[x] != 0) {
     x = up[x];
  }
  return x;
}
```

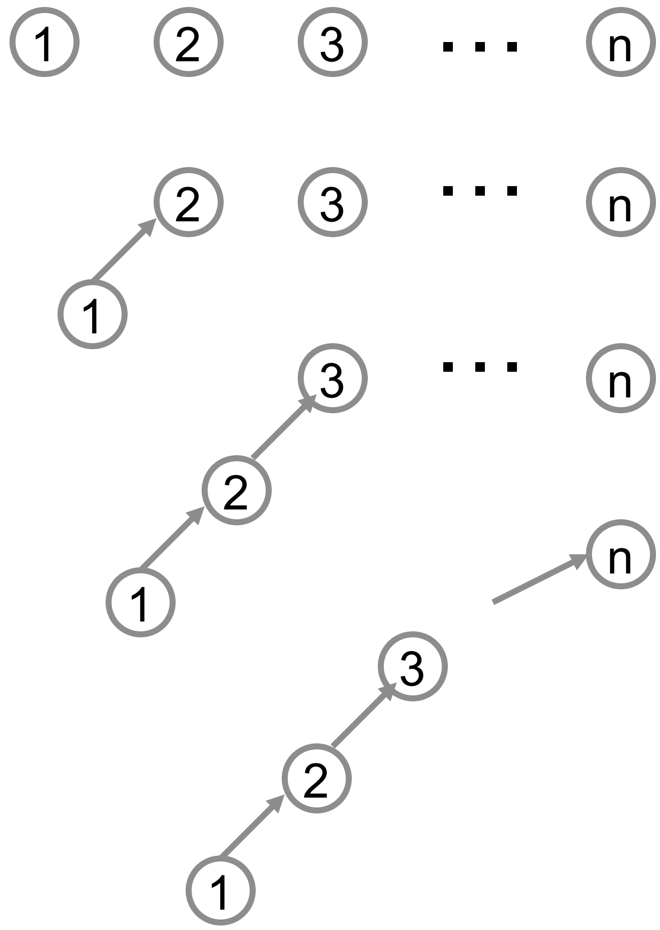
```
// assumes x,y are roots
void union(int x, int y){
  // y = find(y)
  // x = find(x)
  up[y] = x;
}
```

Worst-case run-time for `union`? $O(1)$

Worst-case run-time for `find`? $O(n)$

Worst-case run-time for *m* `finds` and *n*-1 `unions`? $O(m*n)$

THE BAD CASE TO AVOID



`union`(2,1)

`union`(3,2)

:

.

`union`(n,n-1)

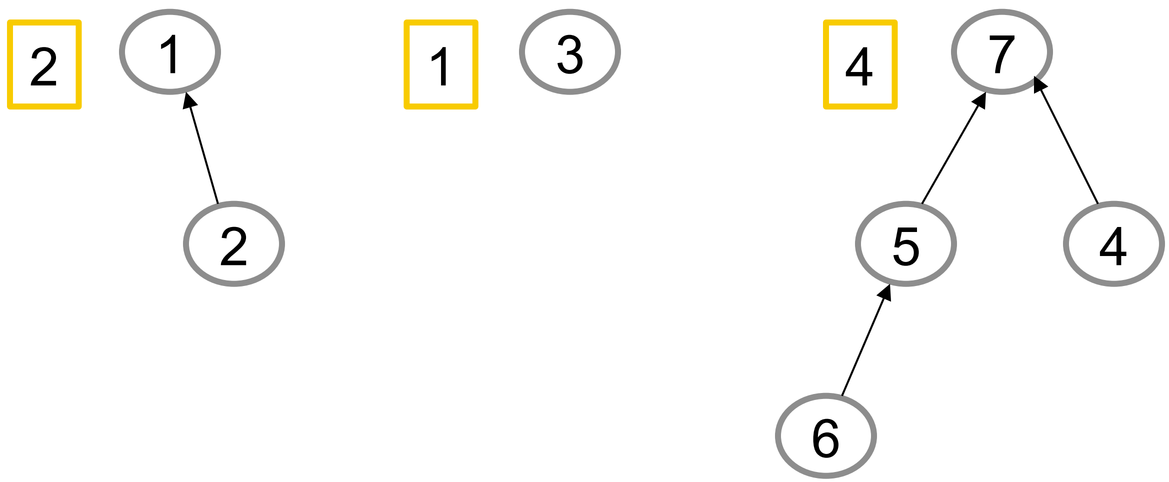
`find`(1)   *n* steps!!

WEIGHTED UNION

Weighted union:

- Always point the *smaller* (total # of nodes) tree to the root of the larger tree

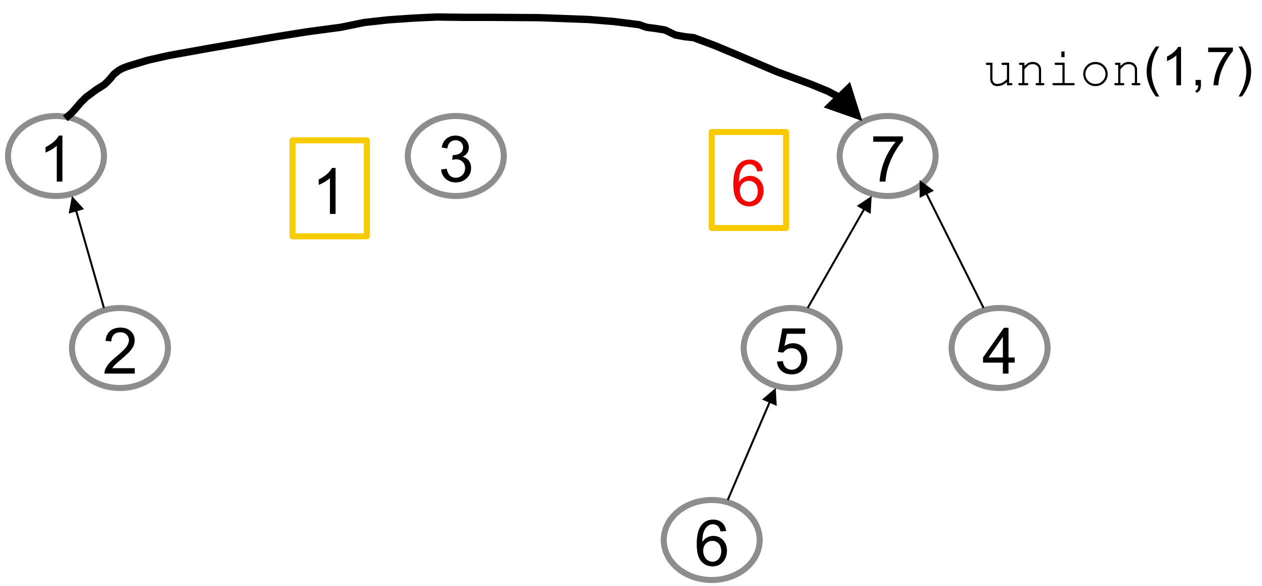
`union`(1,7)



WEIGHTED UNION

Weighted union:

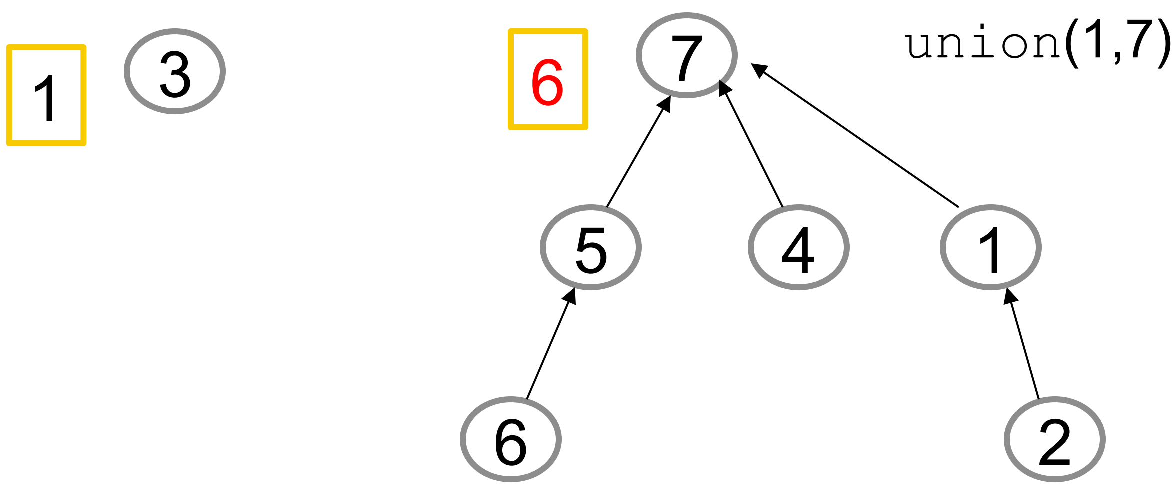
- Always point the *smaller* (total # of nodes) tree to the root of the larger tree
- What just happened to the height of the larger tree?



WEIGHTED UNION

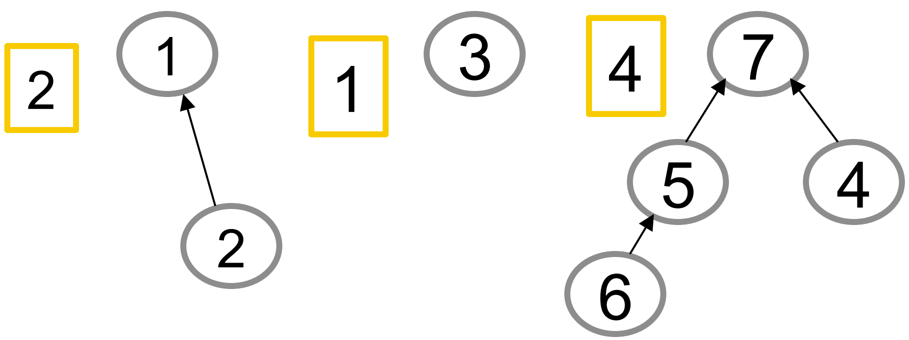
Weighted union:

- Like balancing on an AVL tree, we're trying to keep the traversal from leaf to overall root short

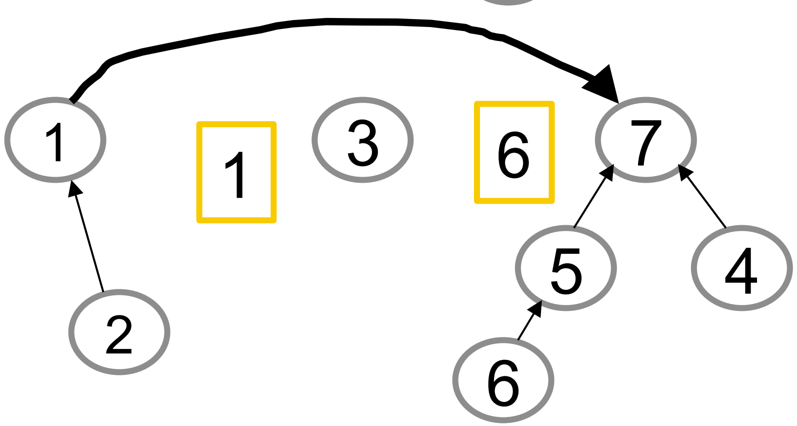


ARRAY IMPLEMENTATION

Keep the *weight* (number of nodes in a second array). Or have one array of objects with two fields. Could keep track of *height*, but that's harder. *Weight* gives us an approximation.



	1	2	3	4	5	6	7
parent	0	1	0	7	7	5	0
weight	2		1				4

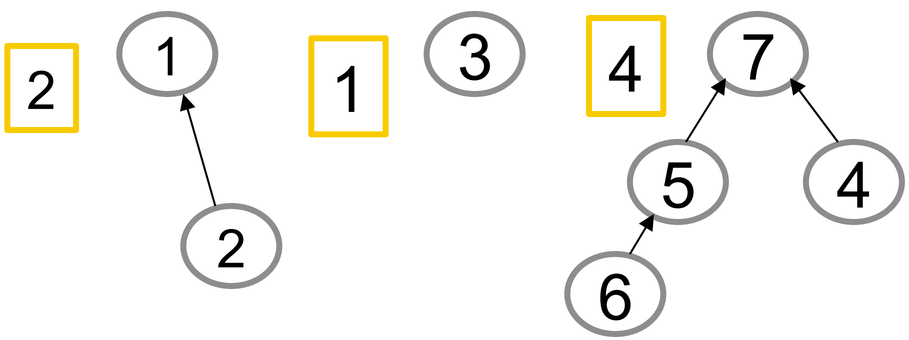


	1	2	3	4	5	6	7
parent	7	1	0	7	7	5	0
weight	2		1				6

NIFTY TRICK

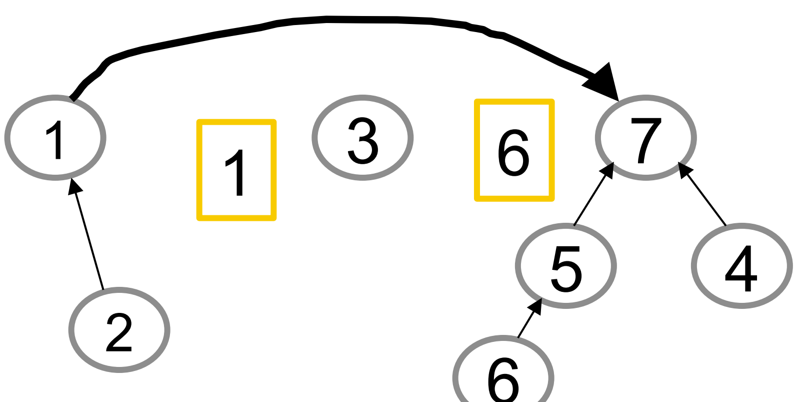
Actually we do not need a second array…

- Instead of storing 0 for a root, store negation of weight. So parent value < 0 means a root.



parent or weight

1	2	3	4	5	6	7
-2	1	-1	7	7	5	-4

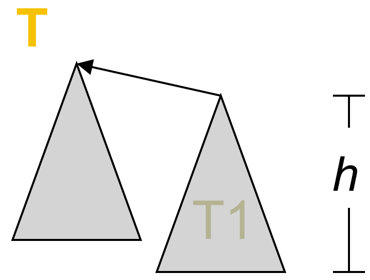


parent or weight

1	2	3	4	5	6	7
7	1	-1	7	7	5	-6

INTUITION: THE KEY IDEA

Intuition behind the proof: No one child can have more than half the nodes



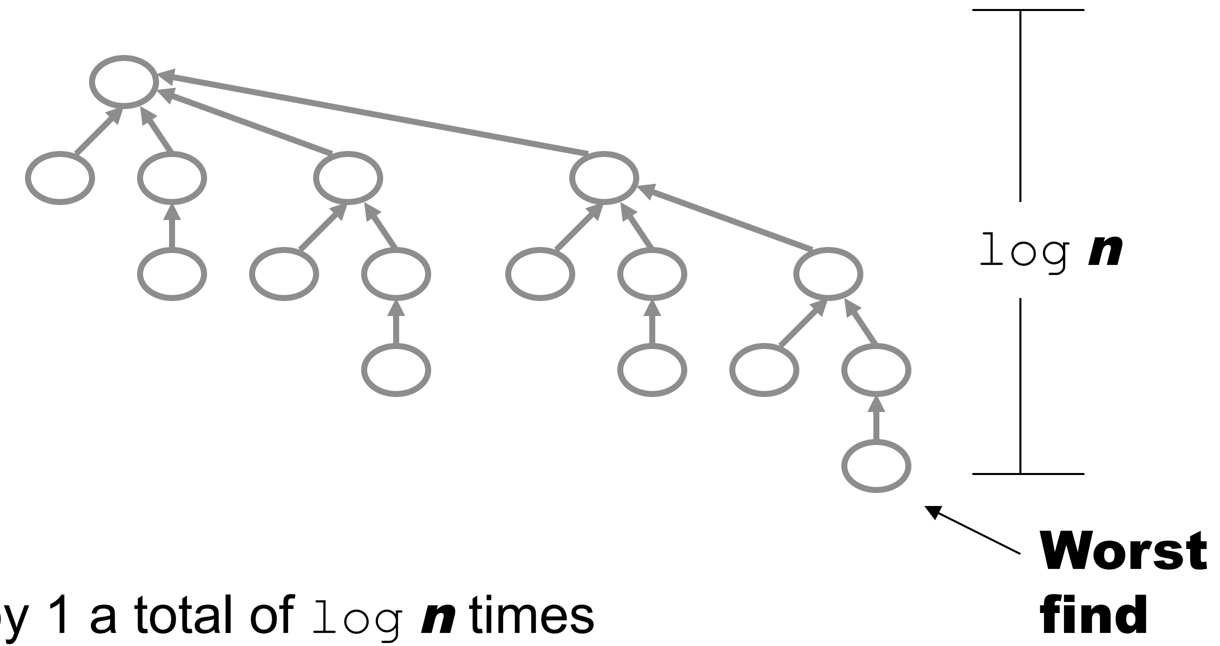
So, as usual, if number of nodes is exponential in height,

then height is logarithmic in number of nodes. The height is log(N) where N is the number of nodes.

So `find` is *O*(`log` *n*)

THE NEW WORST CASE FIND

After n/2 + n/4 + …+ 1 Weighted Unions:

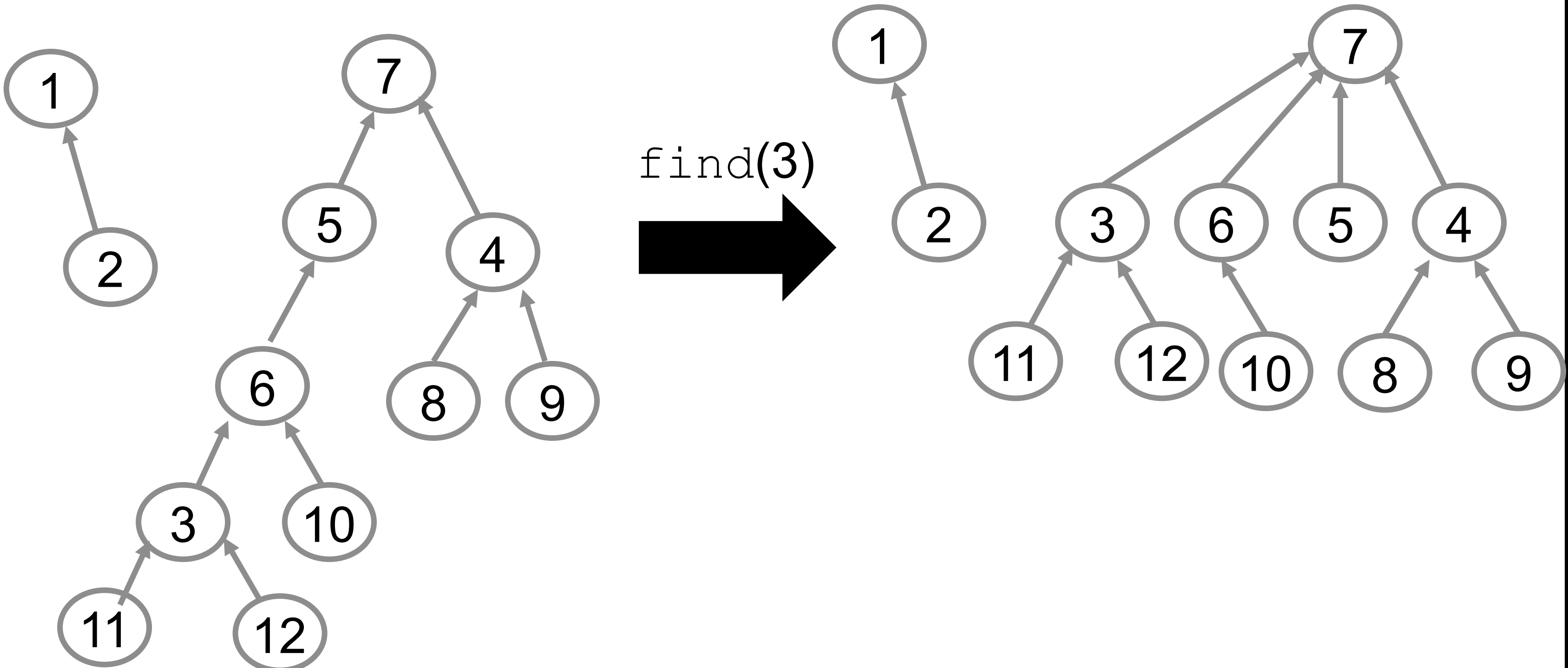


Height grows by 1 a total of `log` ***n*** times

PATH COMPRESSION

Simple idea: As part of a `find`, change each encountered node's parent to point directly to root

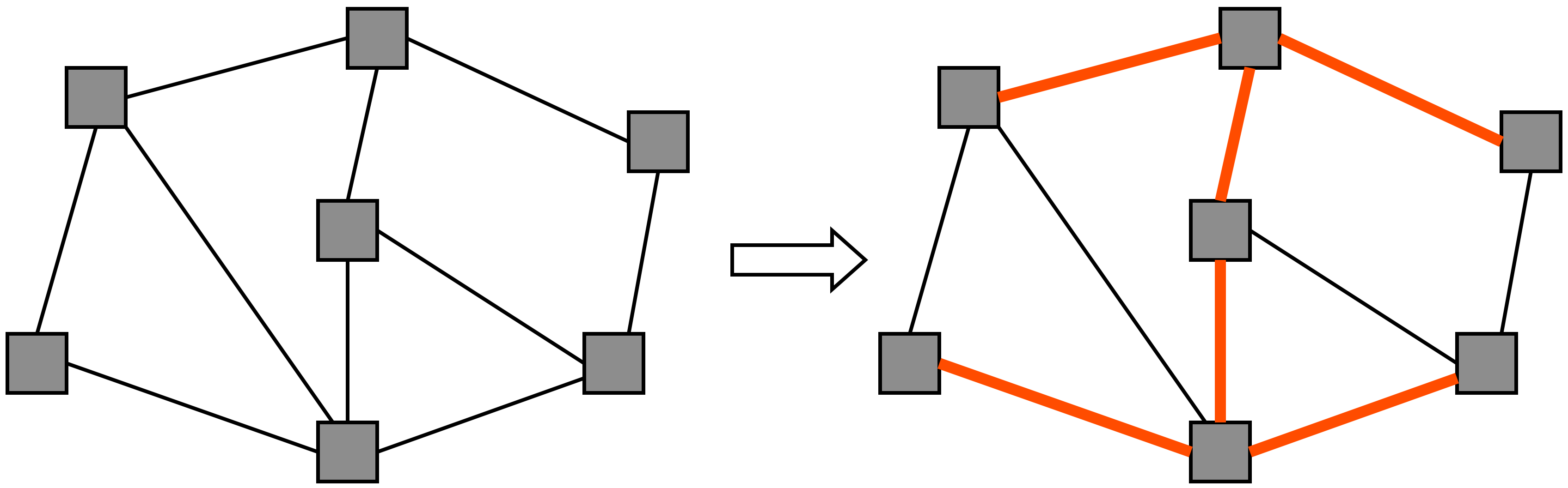
- Faster future `find`s for everything on the path (and their descendants)



SPANNING TREES

Given a *connected* undirected graph G=(V,E), find a subset of edges such that G is still connected

- A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



MINIMAL SPANNING TREES

- How do we get a minimal spanning tree from a traversal?

MINIMAL SPANNING TREES

- **How do we get a minimal spanning tree from a traversal?**
  - What parts of a traversal can we change?

MINIMAL SPANNING TREES

- **How do we get a minimal spanning tree from a traversal?**
  - What parts of a traversal can we change?
  - Select which vertex we visit next by which is closest to an old vertex

PRIM'S ALGORITHM

- A traversal

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node
  - Keep track of all of the vertices you can reach

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node
  - Keep track of all of the vertices you can reach
  - Add the vertex that is closest (has the edge with smallest weight) to the current spanning tree.

PRIM'S ALGORITHM

- **A traversal**
  - Pick a start node
  - Keep track of all of the vertices you can reach
  - Add the vertex that is closest (has the edge with smallest weight) to the current spanning tree.
- **Is this similar to something we've seen before?**

PRIM'S ALGORITHM

- Modify Dijkstra's algorithm

PRIM'S ALGORITHM

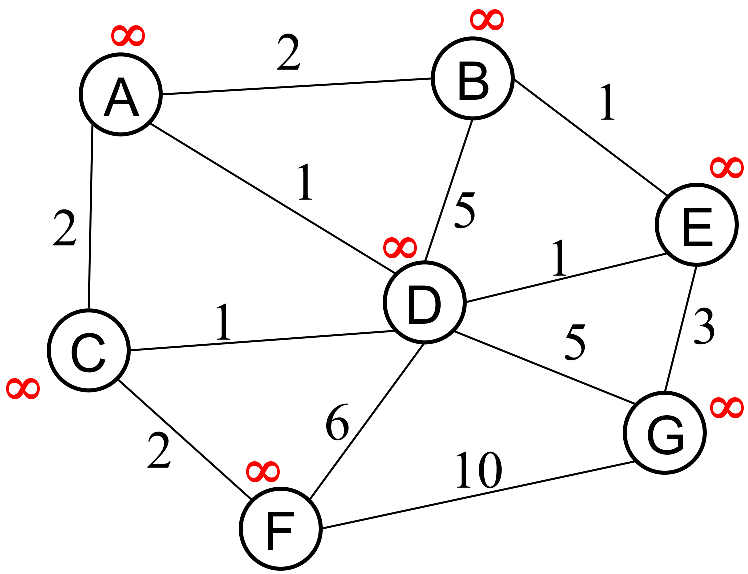
- **Modify Dijkstra's algorithm**
  - Instead of measuring the total length from start to the new vertex, now we only care about the edge from our current spanning tree to new nodes

THE ALGORITHM

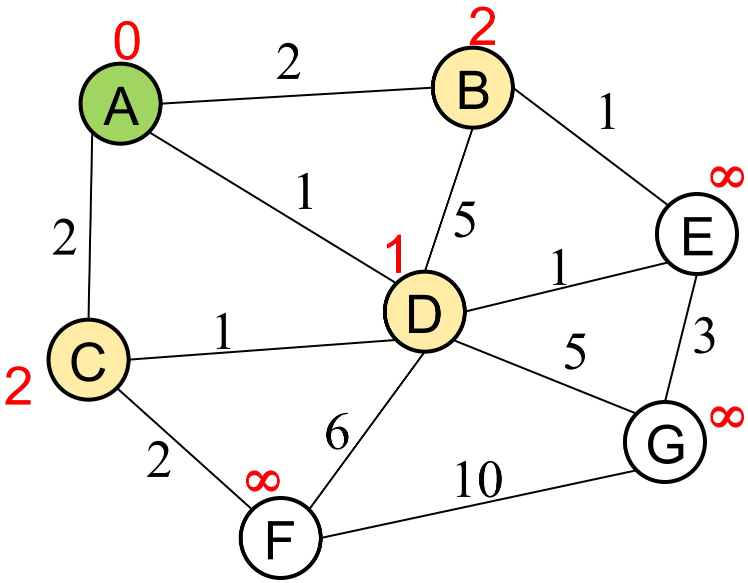
1. **For each node v, set v.cost = ∞ and v.known = false**
2. **Choose any node v**
   a) Mark **v** as known
   b) For each edge **(v,u)** with weight **w**, set **u.cost=w** and **u.prev=v**
3. **While there are unknown nodes in the graph**
   a) Select the unknown node **v** with lowest cost
   b) Mark **v** as known and add **(v, v.prev)** to output
   c) For each edge **(v,u)** with weight **w**,

```
if(w < u.cost) {
    u.cost = w;
  u.prev = v;
}
```

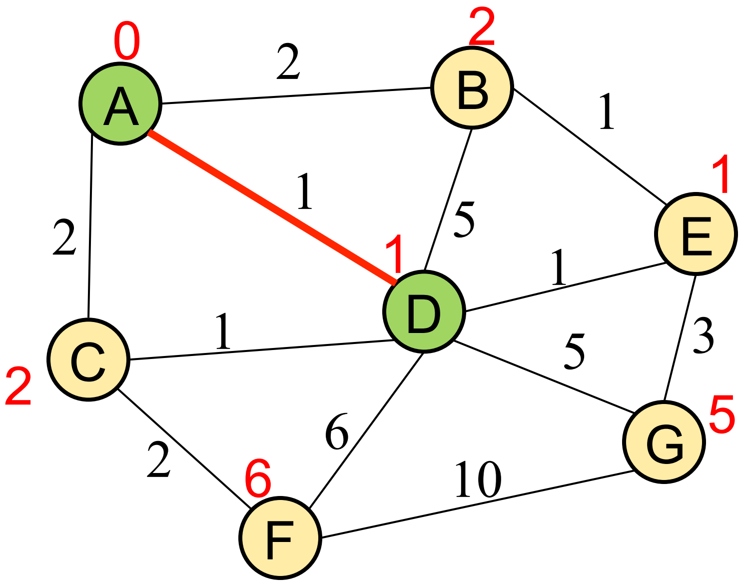
EXAMPLE



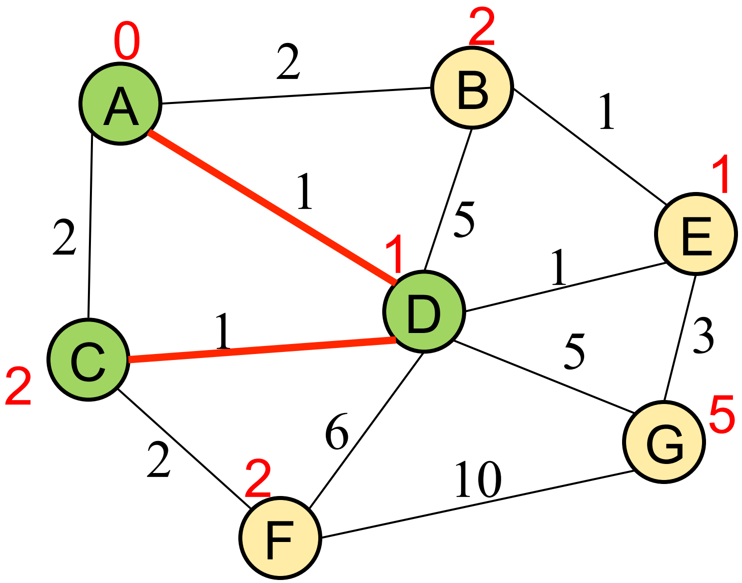
vertex	known?	cost	prev
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	



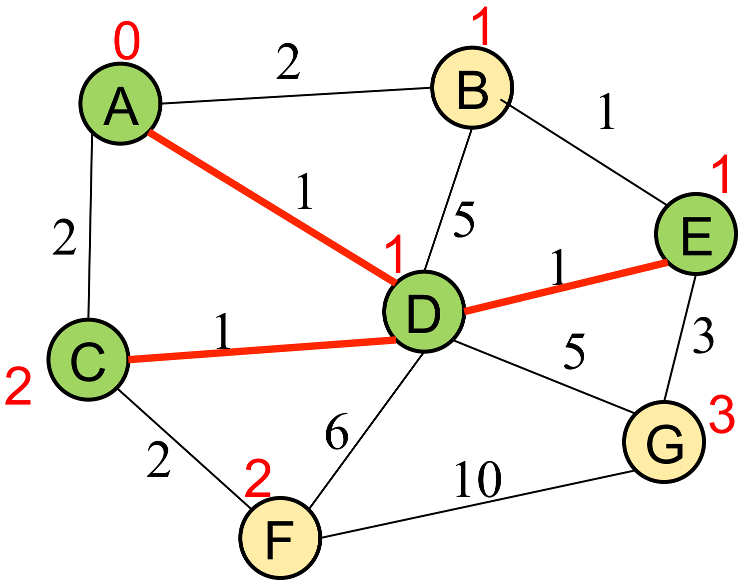
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		2	A
D		1	A
E		∞	
F		∞	
G		∞	



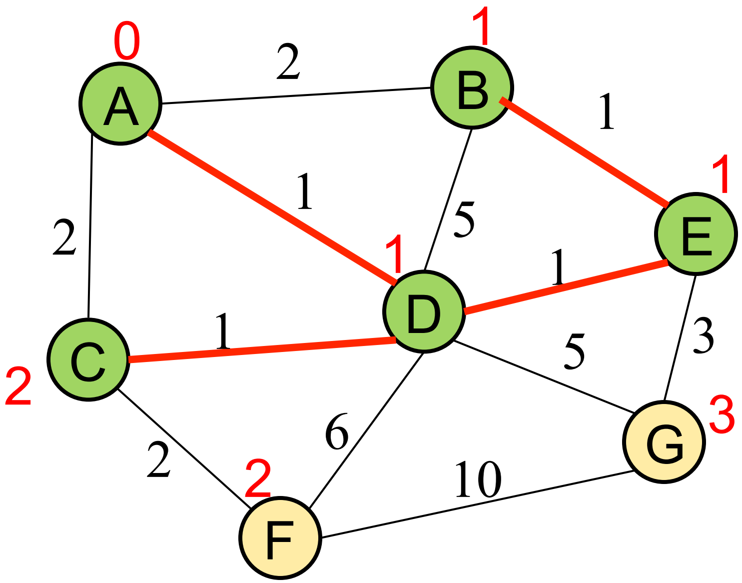
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D



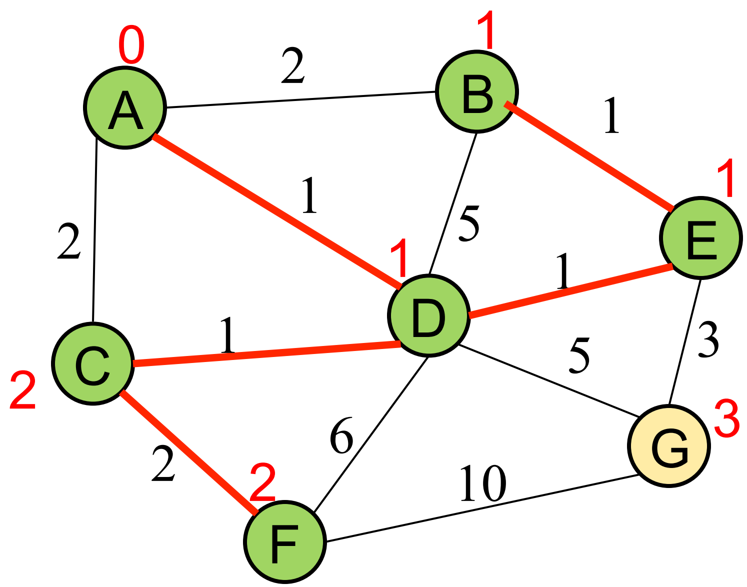
vertex	known?	cost	prev
A	Y	0	
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D



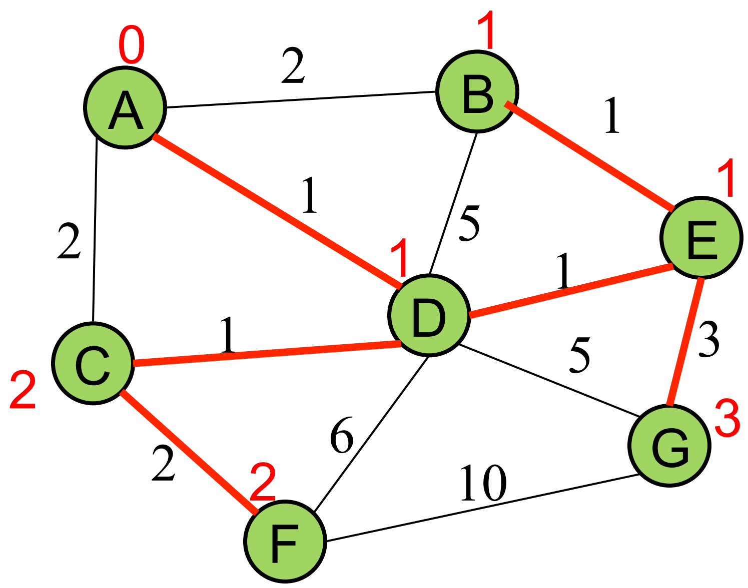
vertex	known?	cost	prev
A	Y	0	
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

PRIM'S ALGORITHM

- Does this give us the correct solution? Why?

PRIM'S ALGORITHM

- **Does this give us the correct solution? Why?**
  - If we consider the "known" cloud as a single vertex, we will never add edges that form a cycle

PRIM'S ALGORITHM

- **Does this give us the correct solution? Why?**
  - If we consider the "known" cloud as a single vertex, we will never add edges that form a cycle
  - Each time, we take the edge that has minimal weight going out of the vertex.

PRIM’S ALGORITHM

- **Does this give us the correct solution? Why?**
  - If we consider the “known” cloud as a single vertex, we will never add edges that form a cycle
  - Each time, we take the edge that has minimal weight going out of the vertex.
  - This is the cheapest way of connecting the two subgraphs.

PRIM'S ALGORITHM

- What is the runtime?

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.

PRIM’S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the “known cloud”

PRIM’S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the “known cloud” **O(1)**

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the "known cloud" **O(1)**
    - How long to check if it is minimal?

PRIM'S ALGORITHM

- **What is the runtime?**
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the "known cloud" **O(1)**
    - How long to check if it is minimal? **O(log |V|)** if we use a priority queue

PRIM'S ALGORITHM

- **O(|E| log |V|)**
  - We can use a priority queue to store all of our vertices, and let the edges to them be the priority.

PRIM'S ALGORITHM

- **O(|E| log |V|)**
  - We can use a priority queue to store all of our vertices, and let the edges to them be the priority.
  - Use the decreaseKey() function when the edge to a vertex changes.

PRIM'S ALGORITHM

- **O(|E| log |V|)**
  - We can use a priority queue to store all of our vertices, and let the edges to them be the priority.
  - Use the decreaseKey() function when the edge to a vertex changes.
  - Without the priority queue, both Prim's and Dijkstra's run in **O(|E||V|)**

KRUSKAL'S ALGORITHM

- Prim's algorithm works from the vertices, and builds a contiguous spanning tree

KRUSKAL’S ALGORITHM

- **Prim’s algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex

KRUSKAL'S ALGORITHM

- **Prim's algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex
- **Kruskal's Algorithm adds edges based on their weight**

KRUSKAL’S ALGORITHM

- **Prim’s algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex
- **Kruskal’s Algorithm adds edges based on their weight**
  - Must check for cycles

KRUSKAL’S ALGORITHM

- **Prim’s algorithm works from the vertices, and builds a contiguous spanning tree**
  - The spanning tree grows out from a single vertex
- **Kruskal’s Algorithm adds edges based on their weight**
  - Must check for cycles
  - Use the union-find data structure to speed up this operation

KRUSKAL'S ALGORITHM

- **Pseudocode:**

KRUSKAL’S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices

KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if it does not form a cycle

KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if the two vertices don't have the same representative in the union find

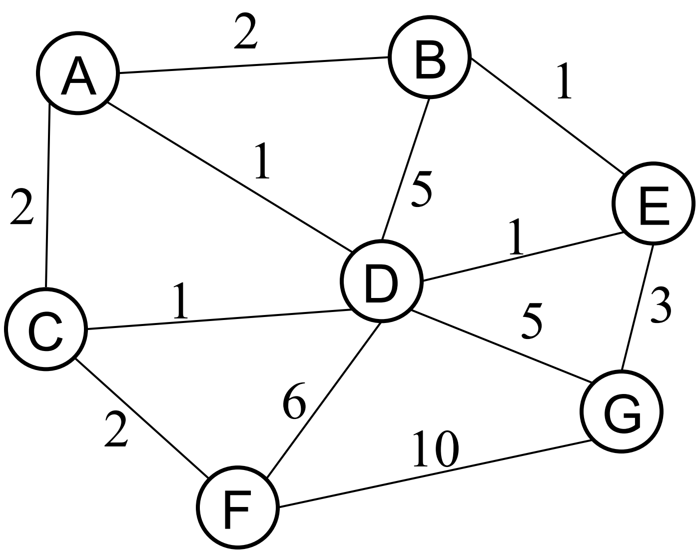
KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if the two vertices don't have the same representative in the union find
  - Union the two vertices in the union find

KRUSKAL'S ALGORITHM

- **Pseudocode:**
  - Sort the edges (or place them into a heap)
  - Create a union-find data structure with all separate vertices
  - For each edge, add it to the minimum spanning tree if the two vertices don't have the same representative in the union find
  - Union the two vertices in the union find
  - Stop after you've added |V|-1 edges

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

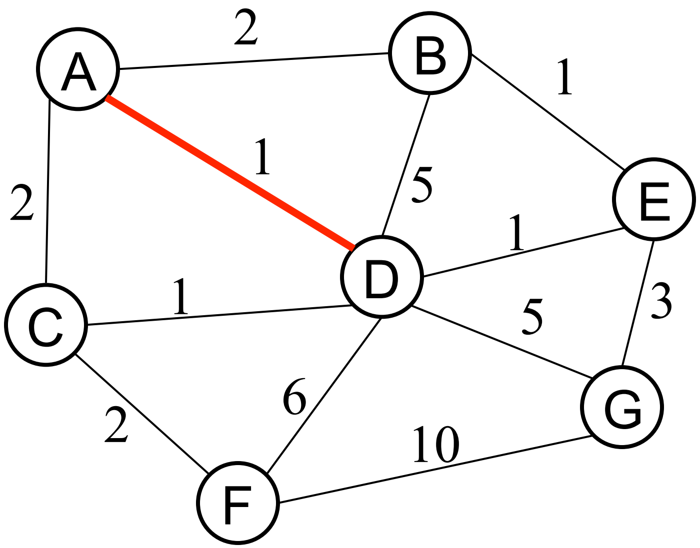
6: (D,F)

10: (F,G)

Output:

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

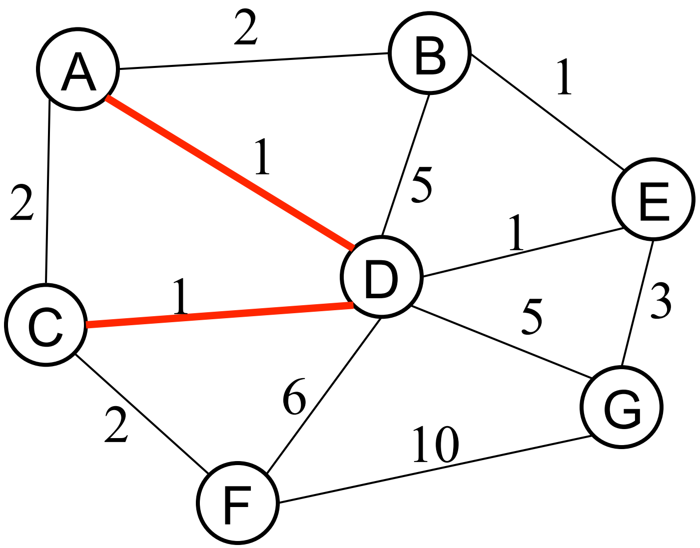
6: (D,F)

10: (F,G)

Output: (A,D)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

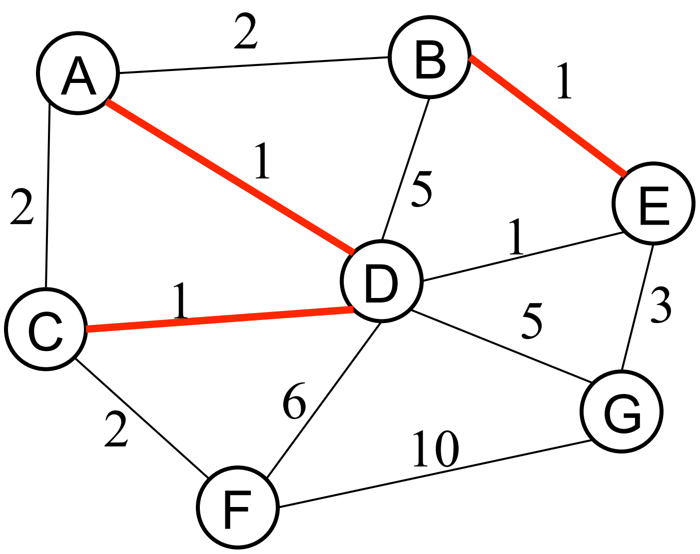
6: (D,F)

10: (F,G)

Output: (A,D), (C,D)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

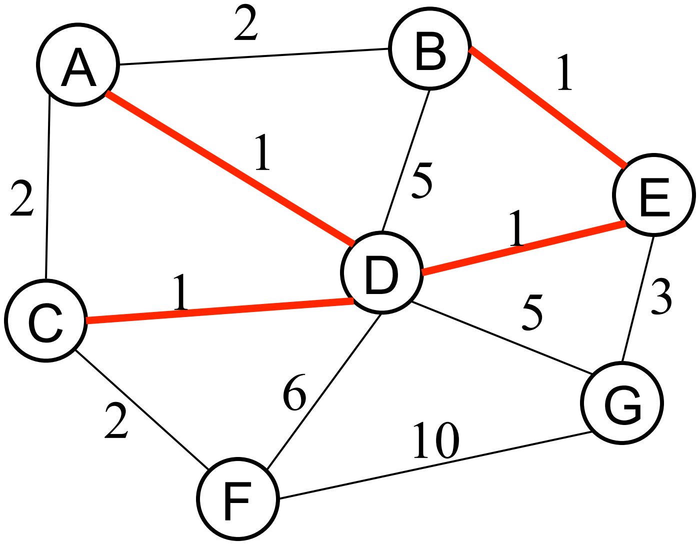
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

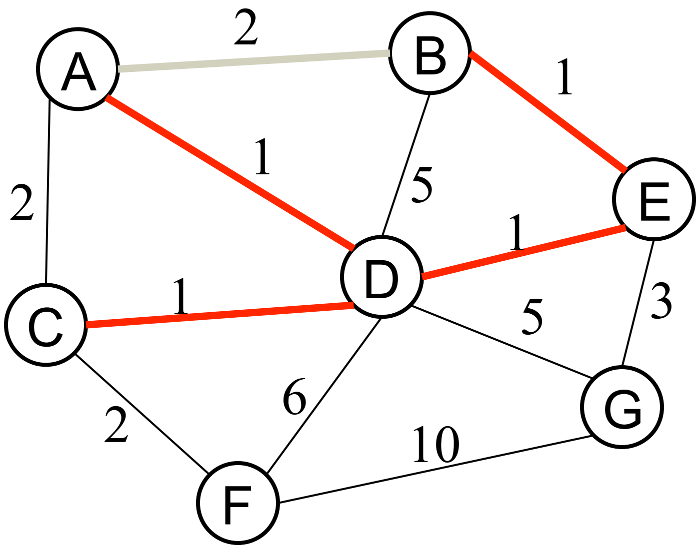
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

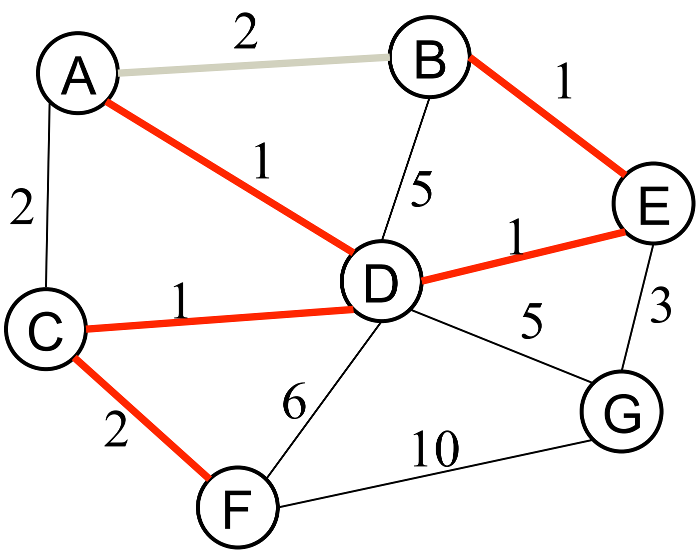
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

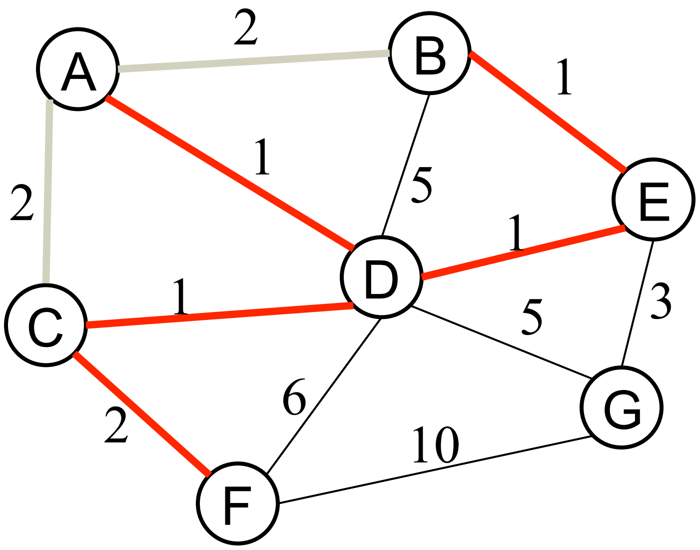
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

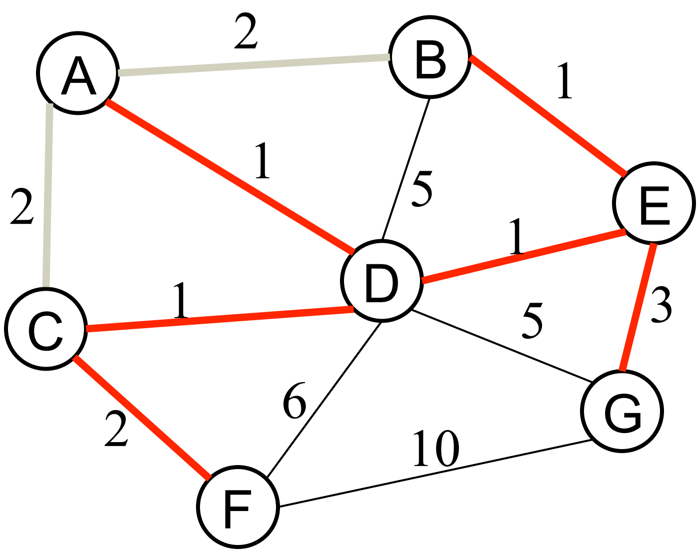
6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

EXAMPLE



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Note: At each step, the union/find sets are the trees in the forest

KRUSKAL'S ALGORITHM

- Runtime

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap

KRUSKAL’S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd’s method!

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap

KRUSKAL’S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd’s method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**
  - **How many times does the loop run?**

KRUSKAL'S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd's method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**
  - **How many times does the loop run? O(E)**

KRUSKAL’S ALGORITHM

- **Runtime**
  - Put edges into a heap **O(|E|)** Floyd’s method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap **O(log |E|)**
    - Check if it forms a cycle **O(log |V|)**
  - **How many times does the loop run? O(E)**
  - **O(|E| log |E|)**

COMPARISONS

- Prim's
  - O(|E| log |V|)
- Kruskal's
  - O(|E| log |E|)
- Since |E| must be at least |V|-1 for the graph to be connected, which do we prefer?

COMPARISONS

- **Prim's**
  - **O(|E| log |V|)**
- **Kruskal's**
  - **O(|E| log |E|)**
- **Since |E| must be at least |V|-1 for the graph to be connected, which do we prefer?**
  - Since |E| is at most $|V|^2$, log|E| is at most $\log(|V|^2)$ which is 2log|V|.

COMPARISONS

- **Prim's**
  - **O(|E| log |V|)**
- **Kruskal's**
  - **O(|E| log |E|)**
- **Since |E| must be at least |V|-1 for the graph to be connected, which do we prefer?**
  - Since |E| is at most $|V|^2$, log|E| is at most $\log(|V|^2)$ which is 2log|V|.
  - So log|E| is O(log|V|)

CONCLUSIONS

- **Prim's and Kruskal's both run in O(|E| log |V|)**

CONCLUSIONS

- Prim's and Kruskal's both run in O(|E| log |V|)
- An undirected graph has a unique minimum spanning tree if all of its edge weights are unique.

CONCLUSIONS

- Prim's and Kruskal's both run in O(|E| log |V|)
- An undirected graph has a unique minimum spanning tree if all of its edge weights are unique.
- If graphs have multiple edges of the same weight, it is possible (but not necessary) that there are many spanning trees of the same weight

CONCLUSIONS

- Prim's and Kruskal's both run in O(|E| log |V|)
- An undirected graph has a unique minimum spanning tree if all of its edge weights are unique.
- If graphs have multiple edges of the same weight, it is possible (but not necessary) that there are many spanning trees of the same weight