UNIVERSITY OF WASHINGTON · 1861

CSE332: Data Abstractions

Lecture 22: Minimum Spanning Trees

James Fogarty

Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman, Richard Ladner, Steve Seitz

Making Connections

You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

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

Q: Are nodes 2 and 4 connected? Indirectly?
Q: How about nodes 3 and 8?
Q: Are any of the paired connections redundant due to indirect connections?
Q: How many sub-networks do you have?

Making Connections

Answering these questions is much easier if we create disjoint sets of nodes that are connected:

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

Q: Are nodes 2 and 4 connected? Indirectly?

Q: How about nodes 3 and 8?

Q: Are any of the paired connections redundant due to indirect connections?

Q: How many sub-networks do you have?

Union-Find aka Disjoint Set ADT

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
  - **Union(5,1)**

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

  To perform the union operation, we replace sets x and y by (x ∪ y)

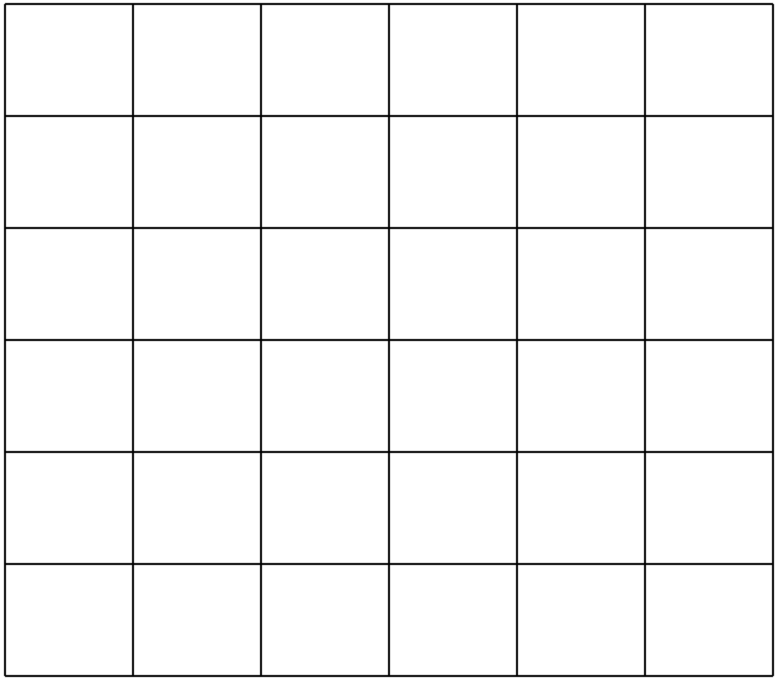
- **Find(x)** – return the name of the set containing x.
  - Given sets: {3,5,7,1,6}, {4,2,8}, {9},
  - **Find(1)** returns 5
  - **Find(4)** returns 8

- We can do Union in constant time.
- We can get Find to be ***amortized*** constant time

  (worst case O(log n) for an individual Find operation).

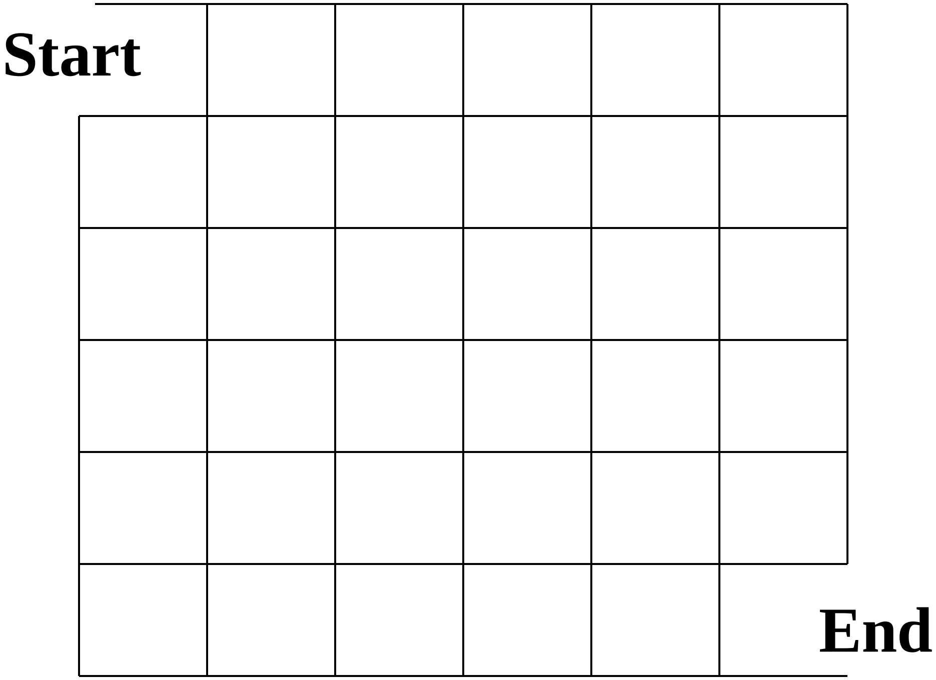
Cute Application

- Build a random maze by erasing edges.



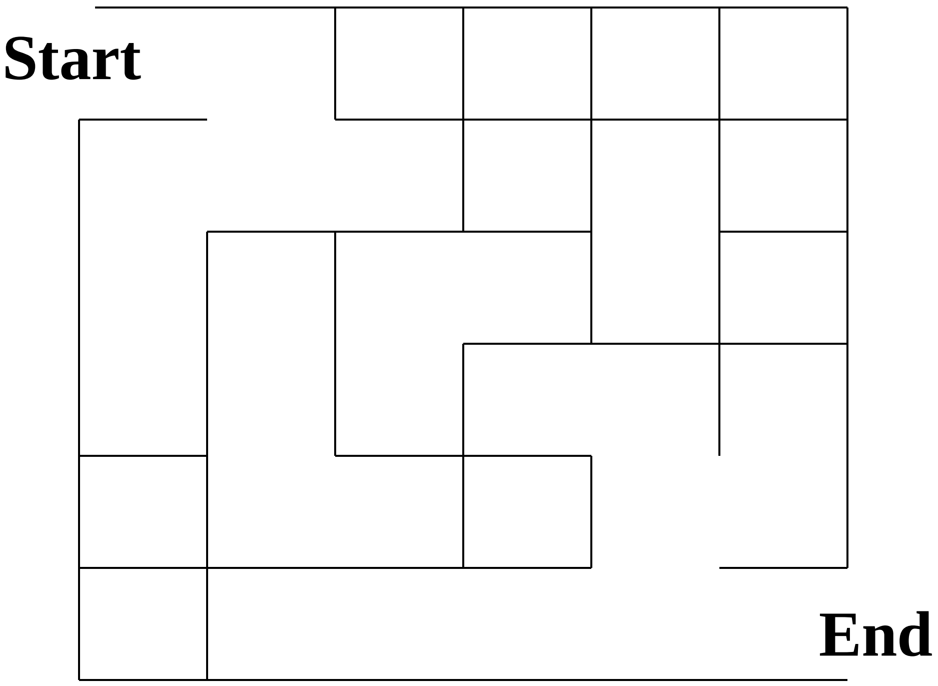
Cute Application

- Pick Start and End



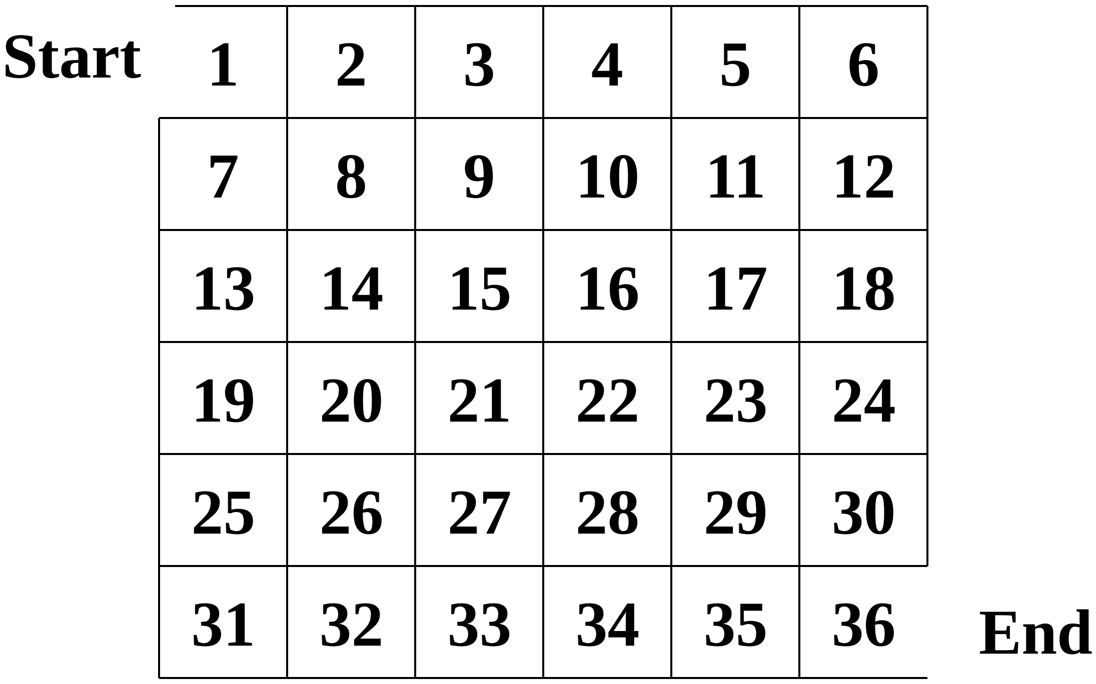
Cute Application

- Repeatedly pick random edges to delete.



Number the Cells

Disjoint sets **S ={ {1}, {2}, {3}, {4},… {36} }**, each cell is unto itself.
We have all edges **W ={ (1,2), (1,7), (2,8), (2,3), … }** 60 walls total.



Start

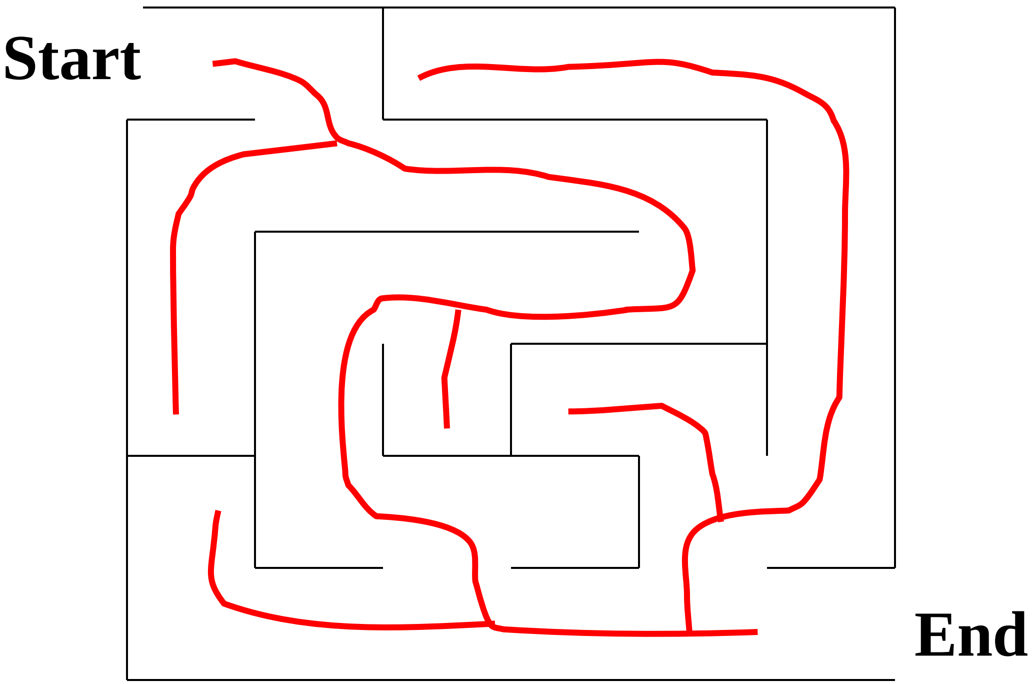
1	**2**	**3**	**4**	**5**	**6**
7	**8**	**9**	**10**	**11**	**12**
13	**14**	**15**	**16**	**17**	**18**
19	**20**	**21**	**22**	**23**	**24**
25	**26**	**27**	**28**	**29**	**30**
31	**32**	**33**	**34**	**35**	**36**

End

Maze Building with Disjoint Union/Find

- Algorithm sketch:
  - Choose wall at random.
    - Boundary walls are not in wall list, because we cannot delete them
  - Erase wall if the neighbors are in disjoint sets
    - Avoids cycles
  - Take union of those sets
  - Repeat until there is only one set
    - Every cell reachable from every other cell

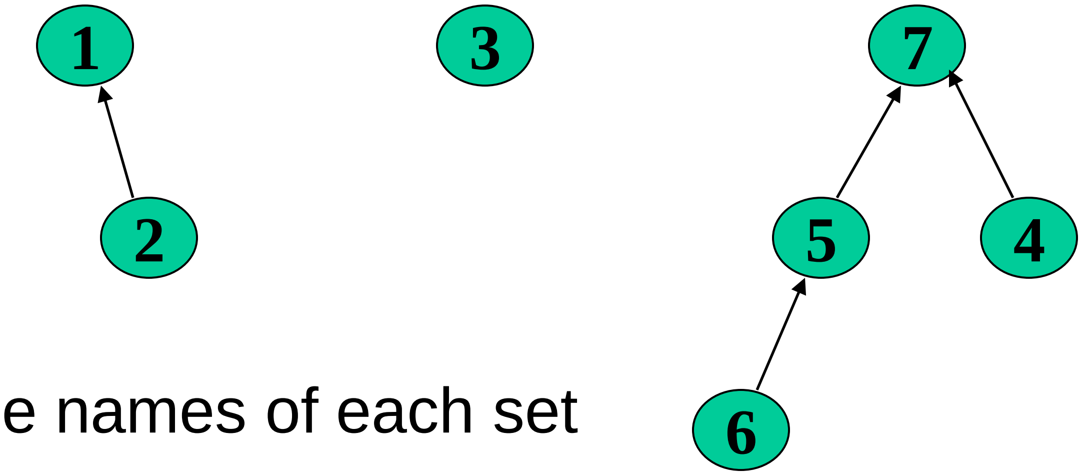
A Hidden Tree



Up-Tree for Disjoin Union/Find

Initial
State

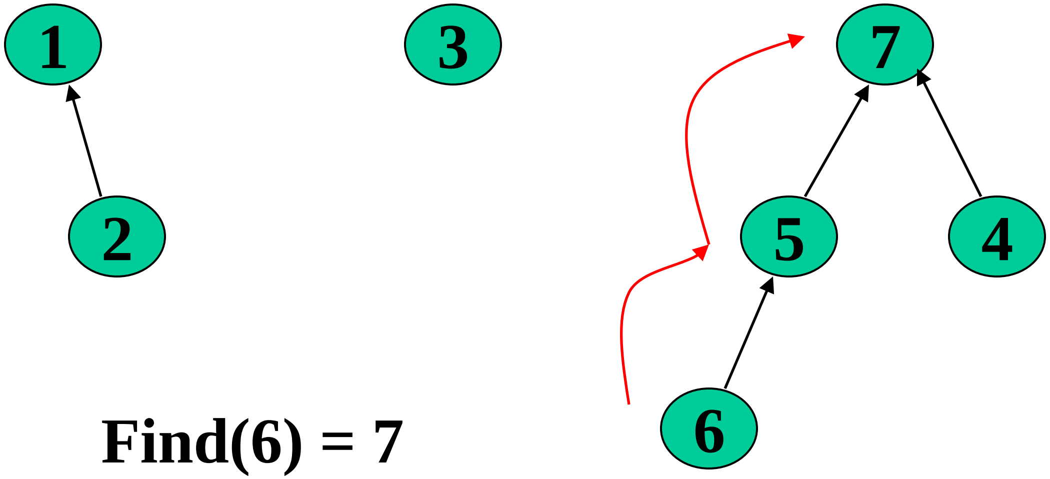
Intermediate
state



Roots are the names of each set

Find Operation

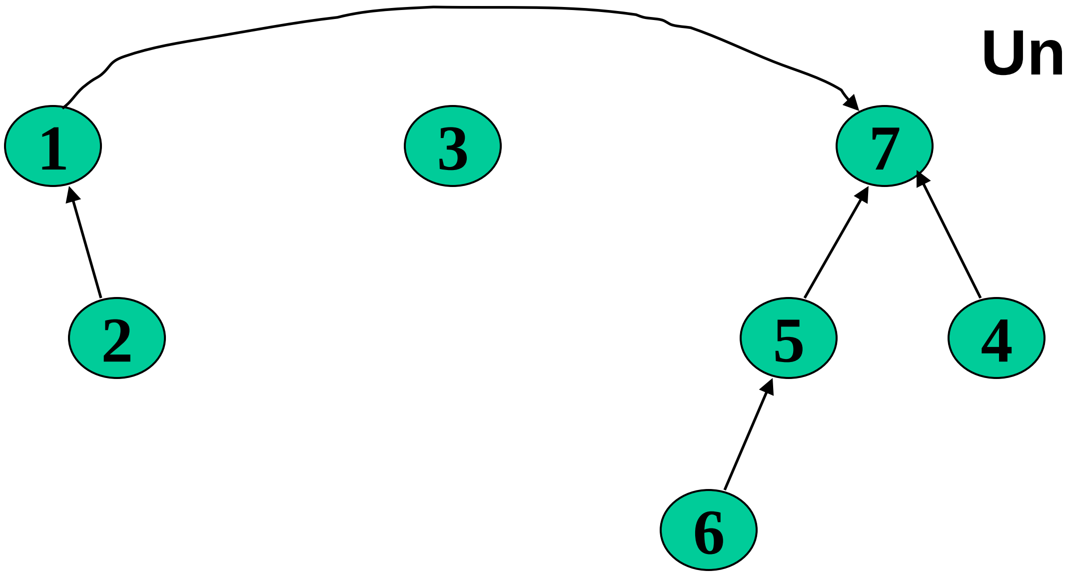
- Find(x):
  follow x to the root and return the root



Find(6) = 7

Union Operation

- Union(i,j):
  assuming i and j roots, point i to j.



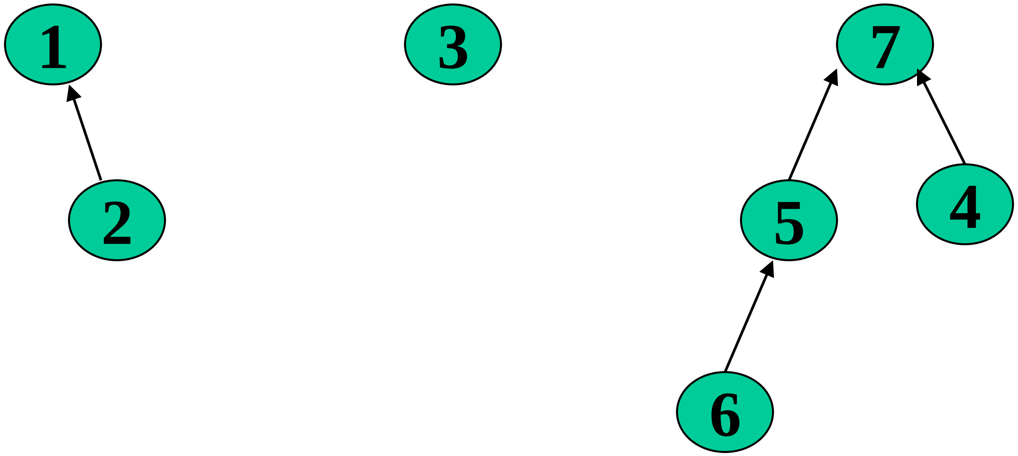
Union(1,7)

Simple Implementation

- Array of indices

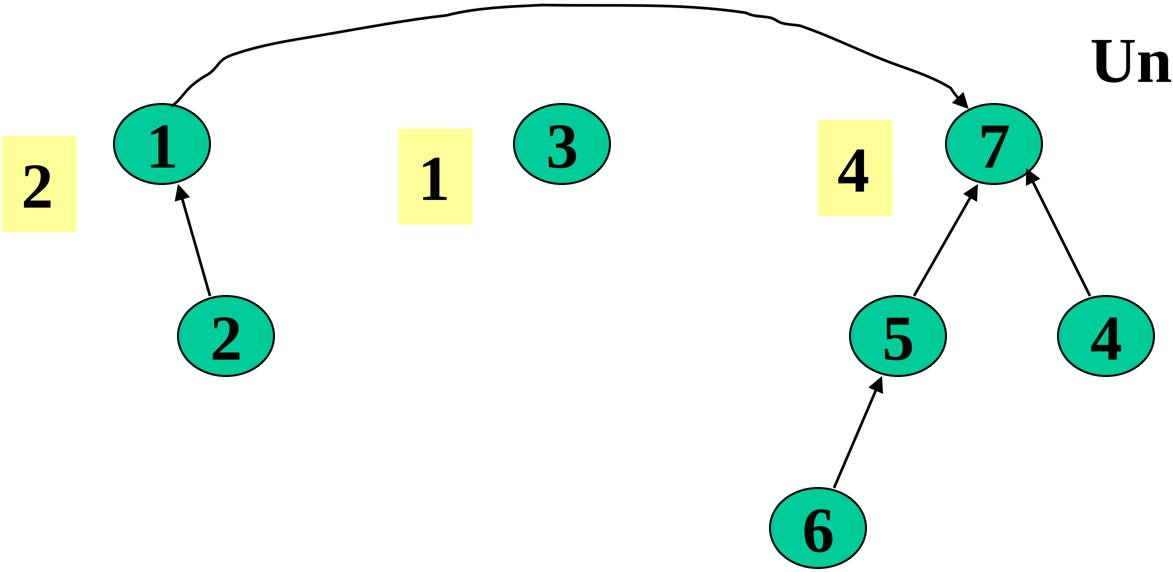
	1	2	3	4	5	6	7
up	**0**	**1**	**0**	**7**	**7**	**5**	**0**

Up[x] =
0 means
x is a root



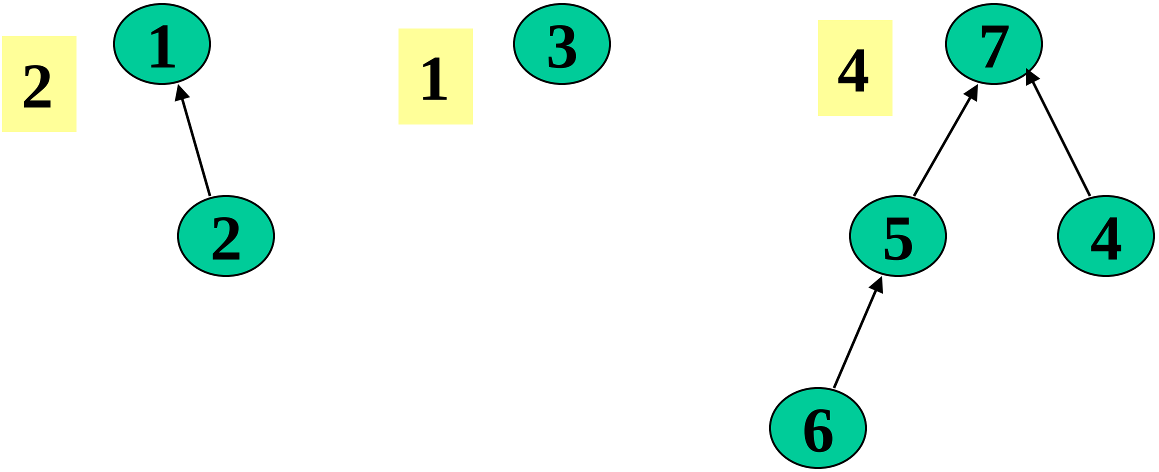
Weighted Union

- Weighted Union
  - Instead of arbitrarily joining two roots, always point the smaller root to the larger root



Union(1,7)

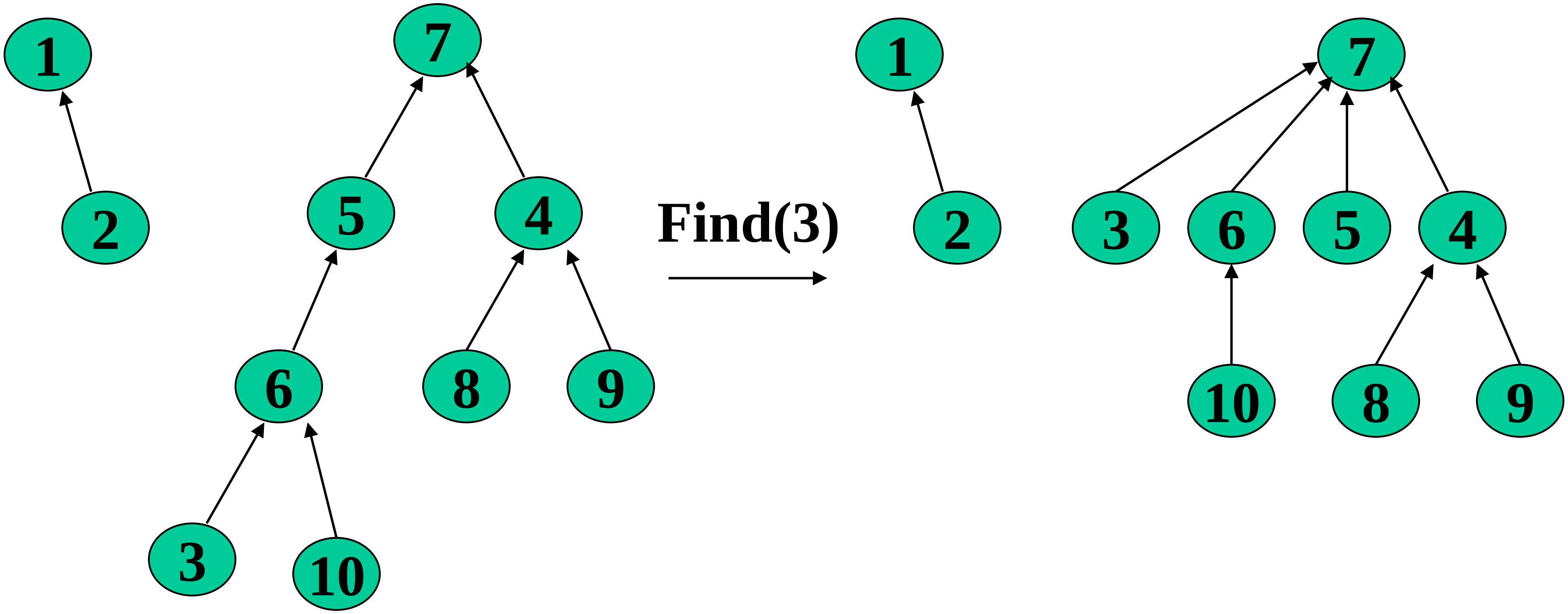
Elegant Array Implementation



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

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

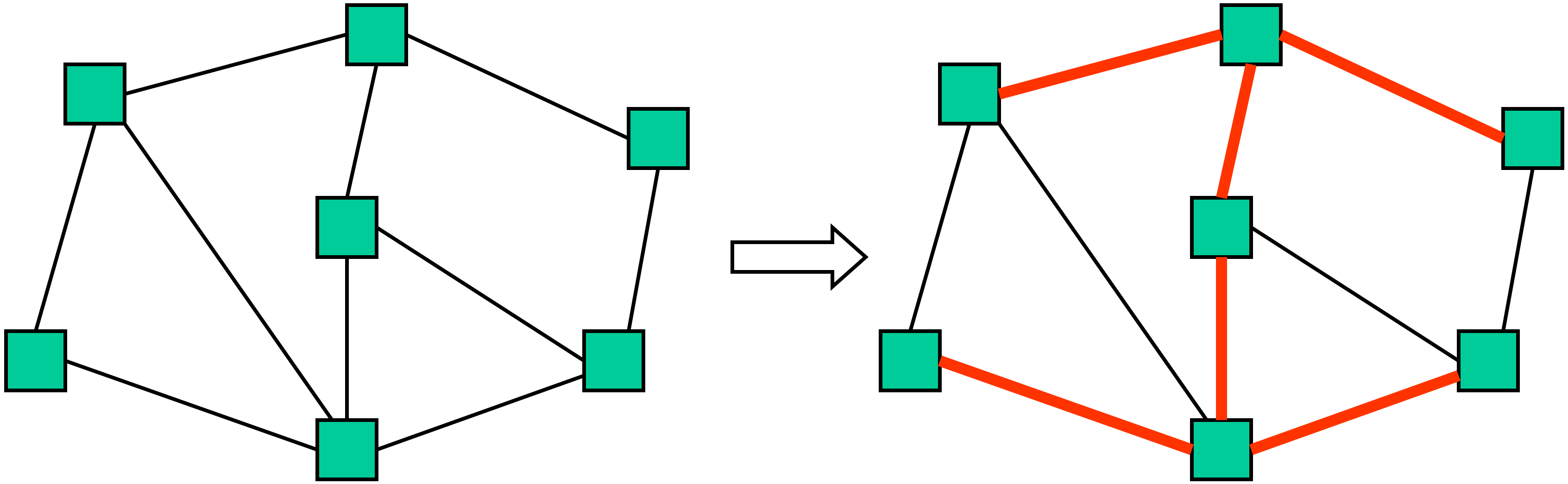


Analyzing Disjoint Sets

- For n elements, total cost of m finds, at most n-1 unions
- Total work is: $O(m+n)$, *i.e.* $O(1)$ amortized
  - With $O(1)$ worst-case for union
  - And $O(\log n)$ worst-case for find
- Find and union *cannot* both be worst-case $O(1)$

Spanning Trees

- A simple problem: Given a *connected* graph **G**=(**V**,**E**), find a minimal subset of the edges such that the graph is still connected
  - A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



Observations

1. 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

2. Solution not unique unless original graph was already a tree

3. Problem ill-defined if original graph not connected

4. A tree with **|V|** nodes has **|V|-1** edges
   - Every spanning tree solution has **|V|-1** edges

Motivation

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

- Example: A “phone tree” so everybody gets the message and no unnecessary calls get made
  - Bad example since would prefer a balanced tree

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: Road network if you cared about asphalt cost

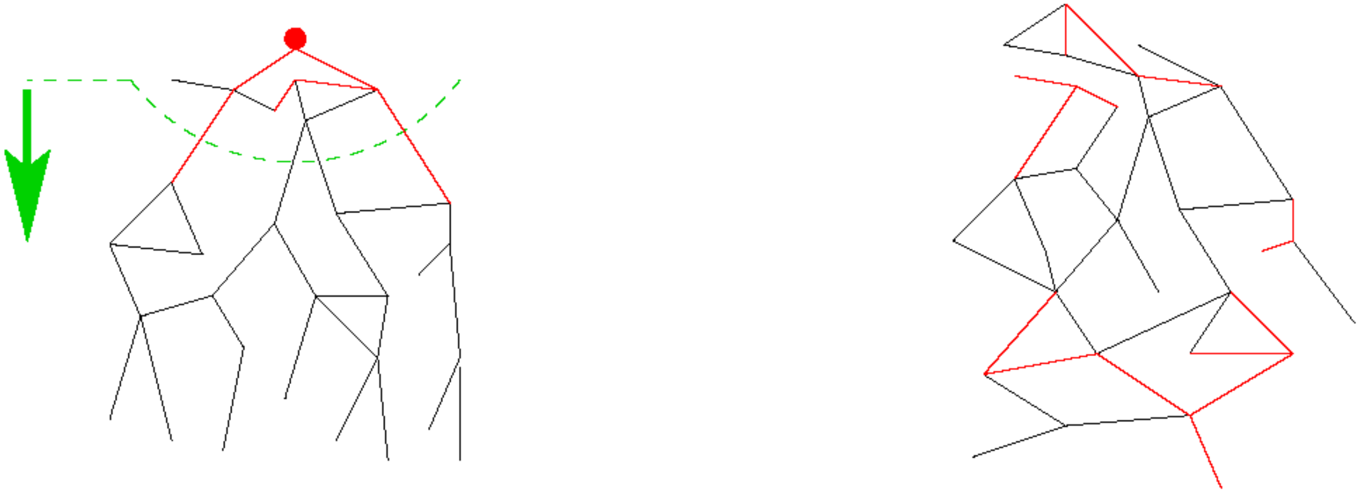
This is the minimum spanning tree problem

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

Two Approaches

Different algorithmic approaches to the spanning-tree problem:

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 doesn't create a cycle



Spanning Tree via DFS

```
spanning_tree(Graph G) {
  for each node i: i.marked = false
  for some node i: f(i)
}
f(Node i) {
  i.marked = true
  for each j adjacent to i:
    if(!j.marked) {
      add(i,j) to output
      f(j) // DFS
    }
}
```

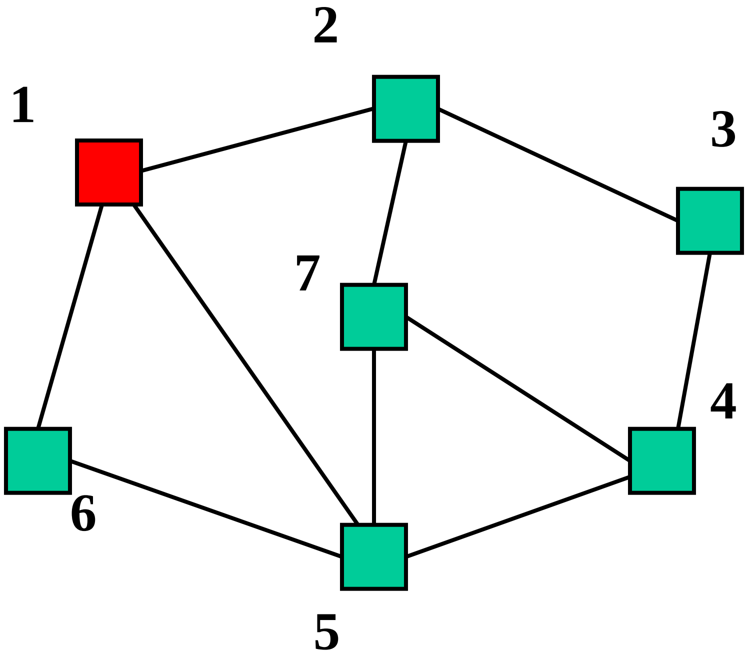
Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|\mathbf{E}|)$

Example

Stack

f(1)



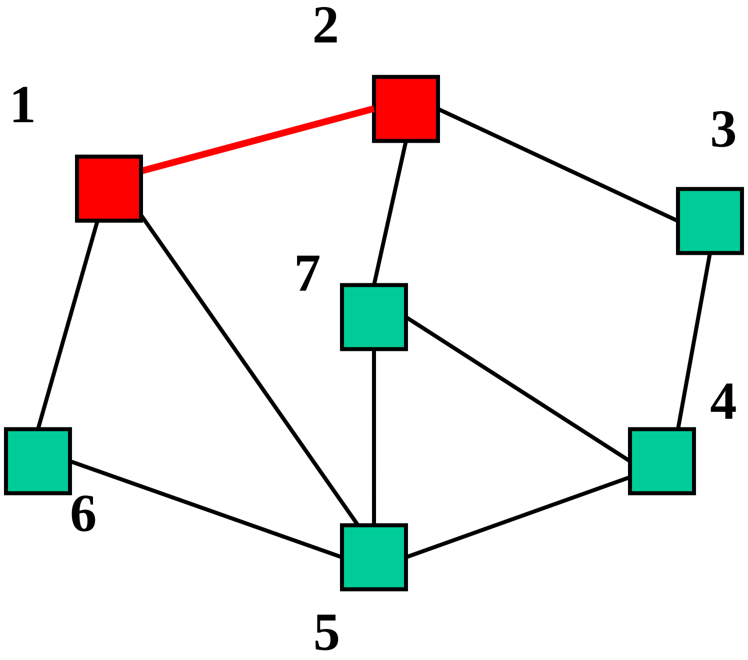
Output:

Example

Stack

f(1)

f(2)



Output: (1,2)

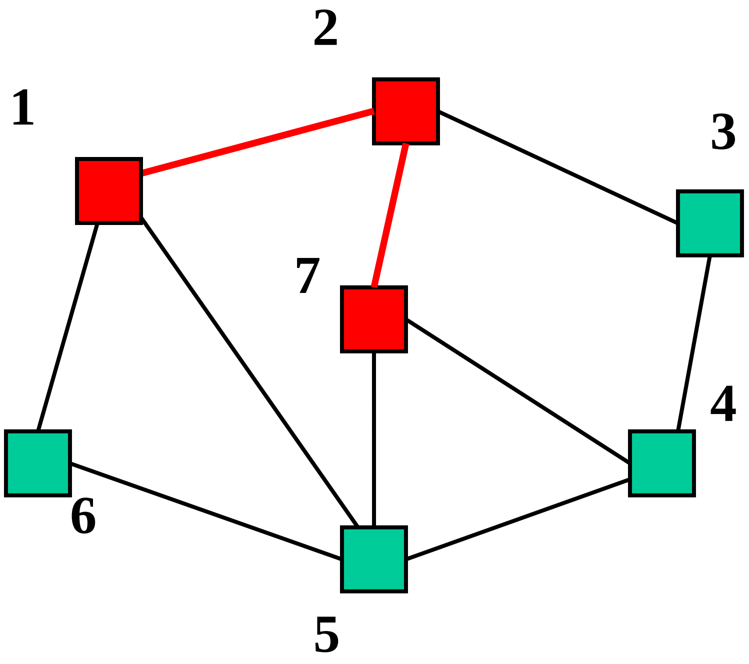
Example

Stack

f(1)

f(2)

f(7)



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

Example

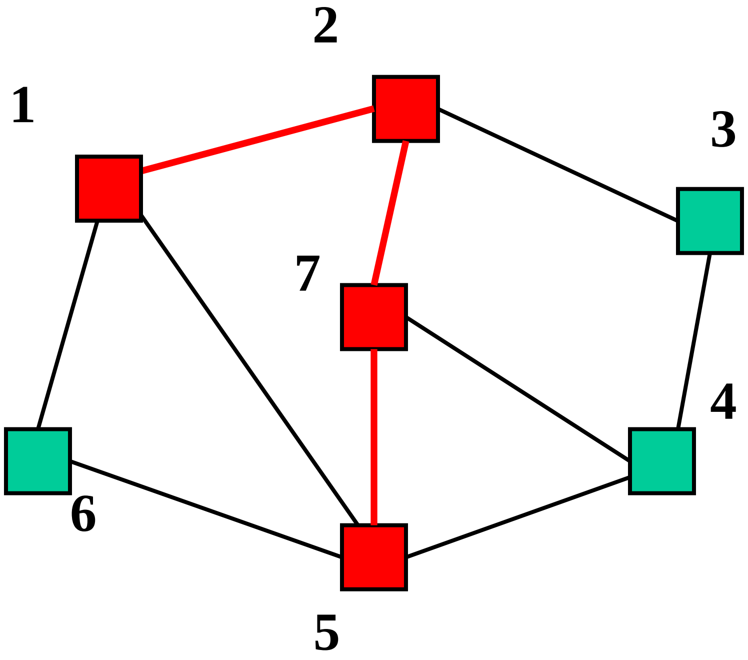
Stack

f(1)

f(2)

f(7)

f(5)



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

Example

Stack

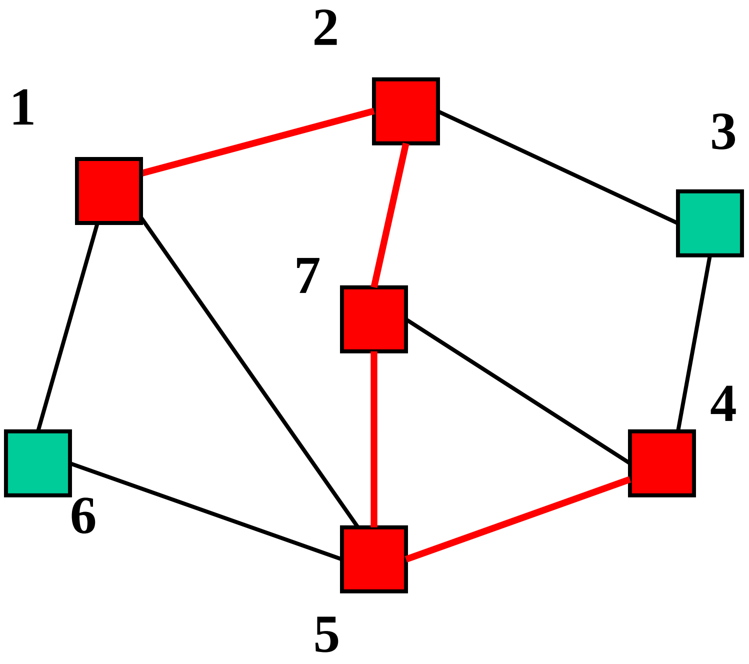
f(1)

f(2)

f(7)

f(5)

f(4)



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

Example

Stack

f(1)

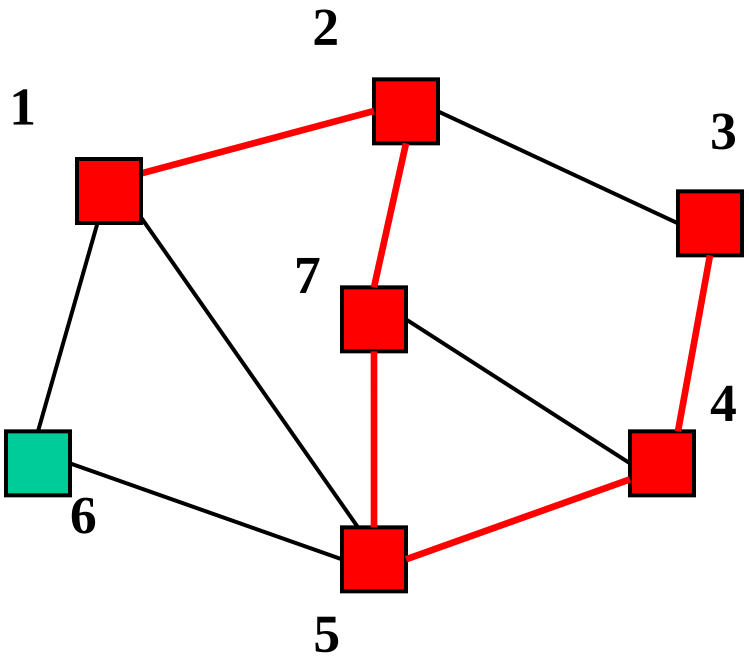
f(2)

f(7)

f(5)

f(4)

f(3)



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

Example

Stack

f(1)

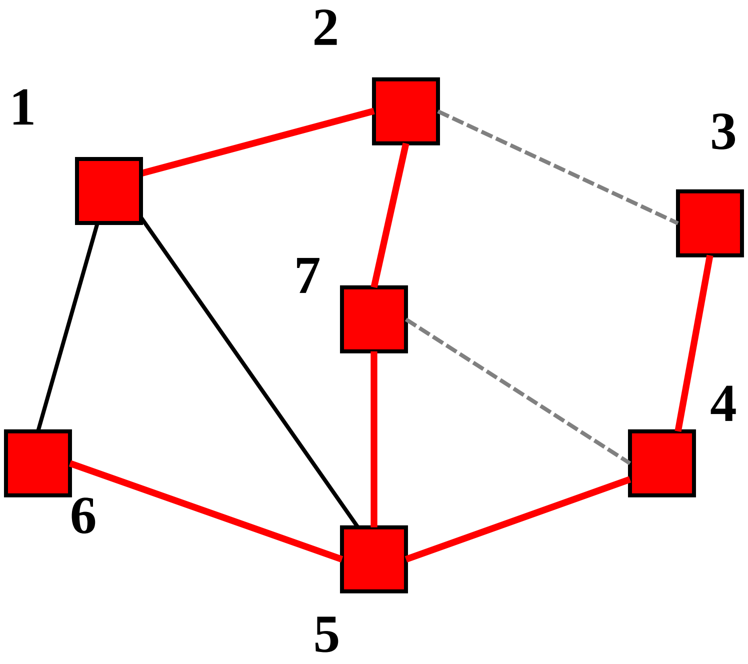
f(2)

f(7)

f(5)

f(4) f(6)

f(3)



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

Example

Stack

f(1)

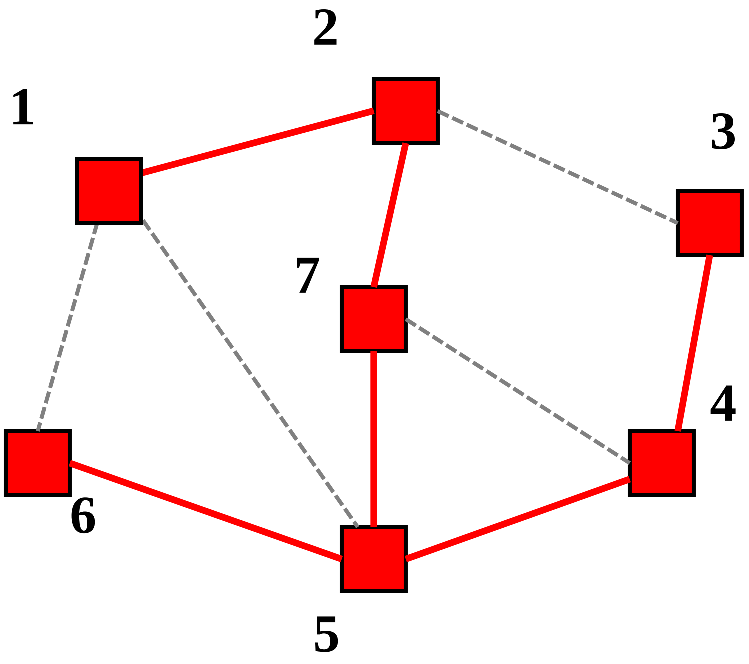
f(2)

f(7)

f(5)

f(4) f(6)

f(3)



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
  (or else it would have created a cycle)
- The graph is connected, we consider all edges

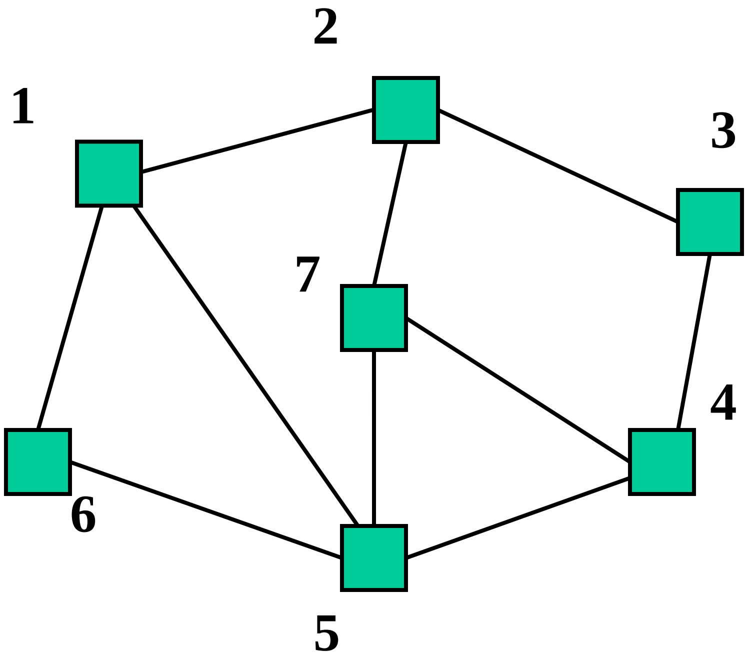
Efficiency:

- 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)

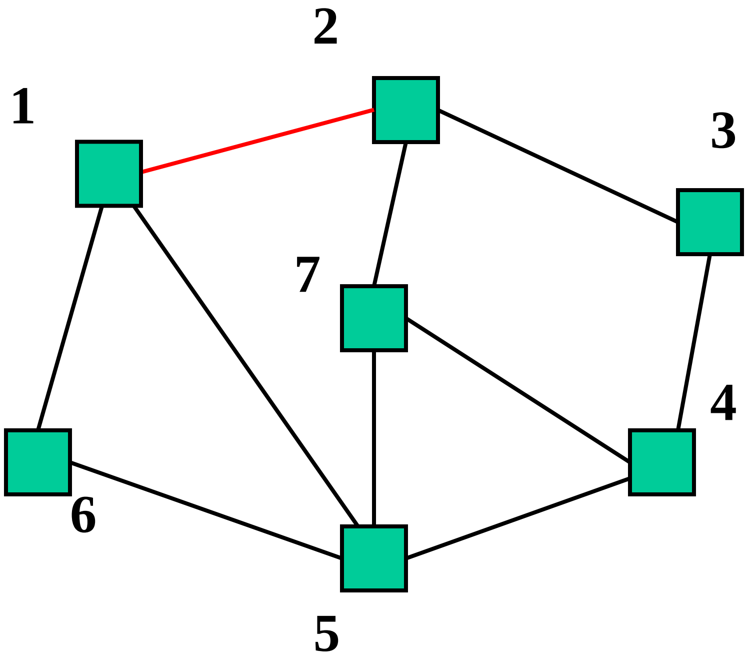


Output:

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)

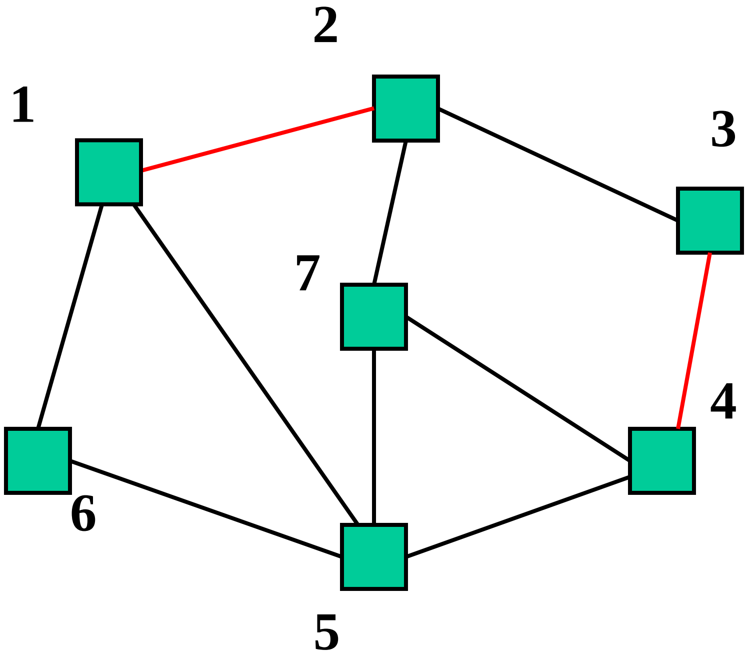


Output: (1,2)

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)

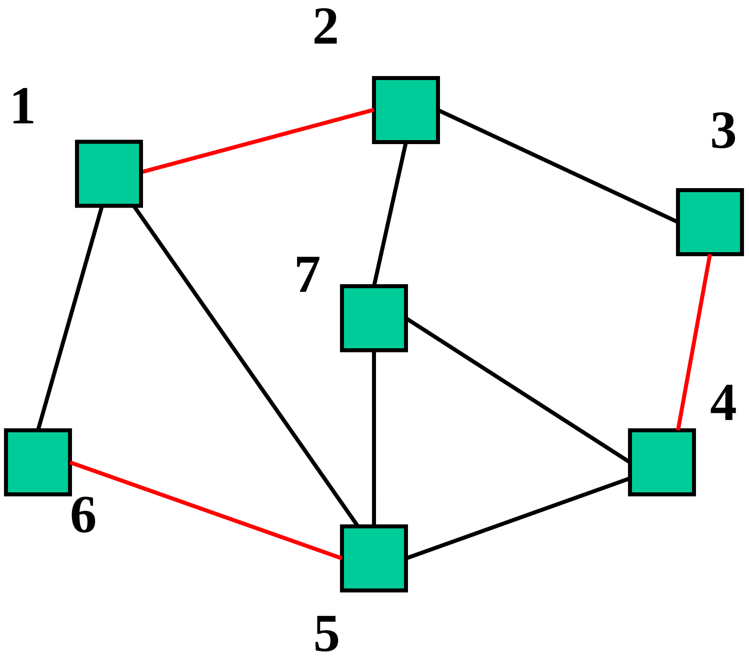


Output: (1,2), (3,4)

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)

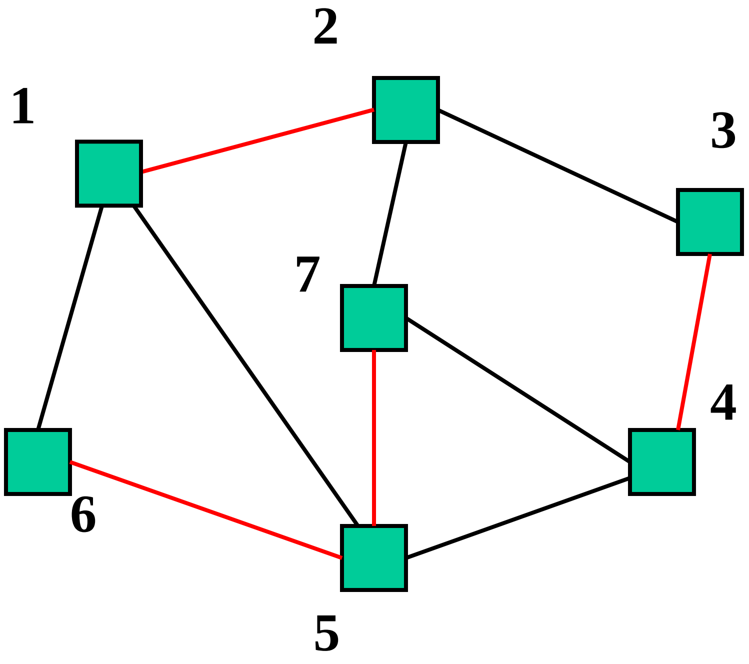


Output: (1,2), (3,4), (5,6),

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)

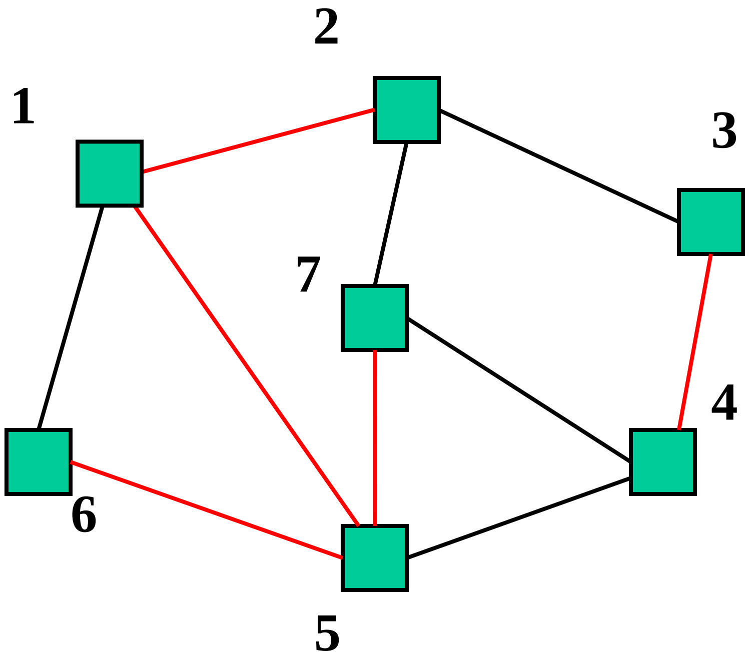


Output: (1,2), (3,4), (5,6), (5,7)

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)

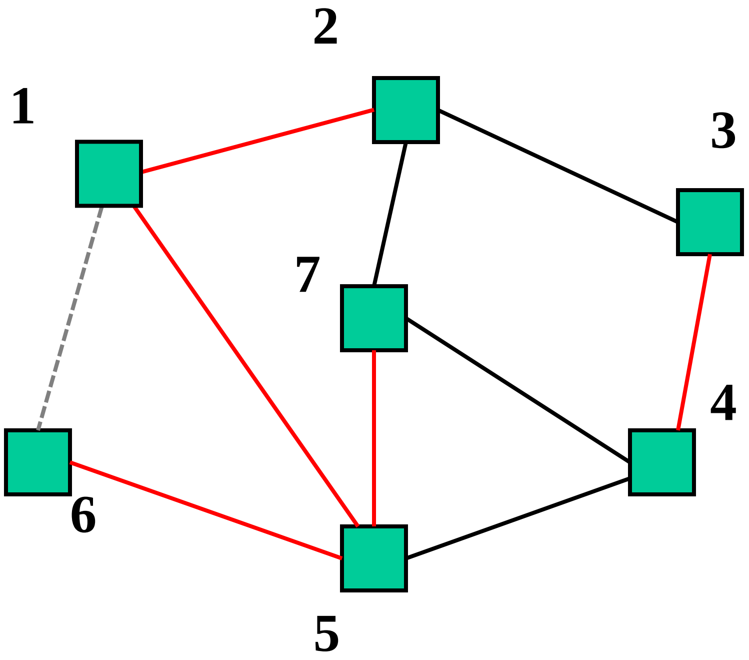


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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)

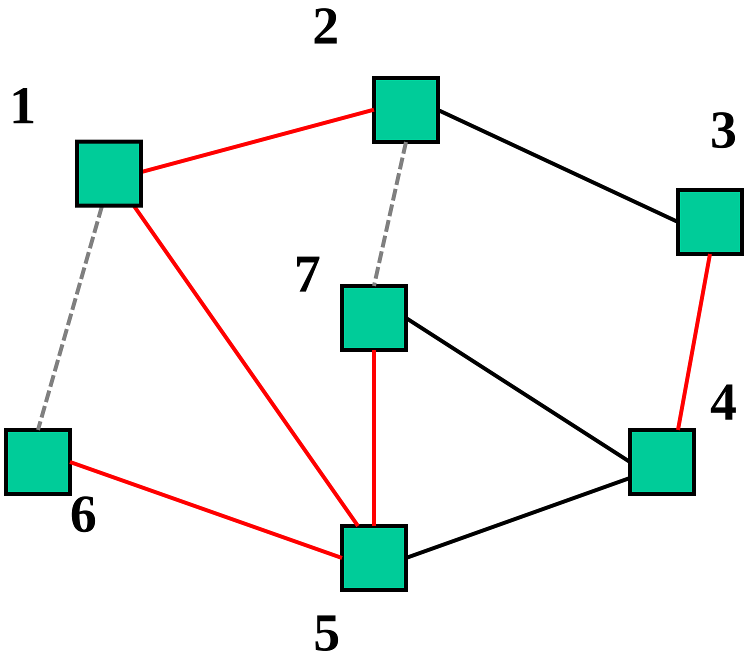


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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)

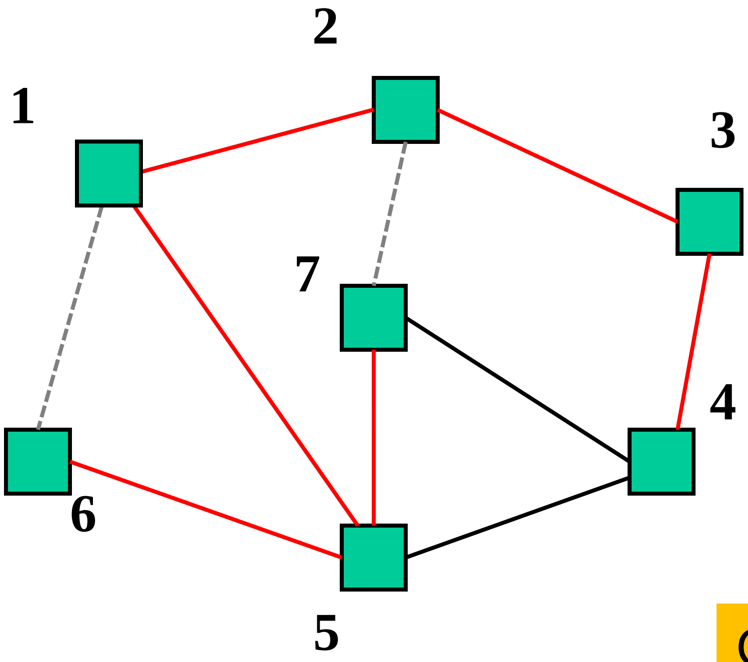


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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)



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

Can stop once we have **|V|-1** edges

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 using the disjoint-set ADT
  - Initially, each item is in its own 1-element set
  - **`find(u)`**: what set contains **`u`**?
  - **`union(u,v)`**: union (combine) the sets containing **`u`** and **`v`**

Aside: Union-Find aka Disjoint Set ADT

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
  - **Union(5,1)**

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

  To perform the union operation, we replace sets x and y by (x ∪ y)

- **Find(x)** – return the name of the set containing x.
  - Given sets: {3,5,7,1,6}, {4,2,8}, {9},
  - **Find(1)** returns 5
  - **Find(4)** returns 8

- We can do Union in constant time.
- We can get Find to be ***amortized*** constant time

  (worst case O(log n) for an individual Find operation).

Using Disjoint-Set

Can use a disjoint-set implementation
in our spanning-tree algorithm to detect cycles:

Invariant: **`u`** and **`v`** are connected in output-so-far

iff

`u` and **`v`** in the same set

- Initially, each node is in its own set
- When processing edge **`(u,v)`**:
  - If **`find(u)==find(v)`**, then do not add the edge
  - Else add the edge and **`union(u,v)`**

Summary so Far

The spanning-tree problem

- Add nodes to partial tree approach is *O*(**|E|**)
- Add acyclic edges approach is *O*(**|E|** **`log`** **|V|**)
  - Using the disjoint-set ADT “as a black box”

But really want to solve the minimum-spanning-tree problem

- Given a weighted undirected graph,
  give a spanning tree of minimum weight
- Same two approaches will work with minor modifications
- Both will be *O*(**|E|** **`log`** **|V|**)

Getting to the Point

Algorithm #1

Shortest-path is to Dijkstra’s Algorithm

as

Minimum Spanning Tree is to Prim’s Algorithm

(Both based on expanding cloud of known vertices, basically using a priority queue instead of a DFS stack)

Algorithm #2

Kruskal’s Algorithm for Minimum Spanning Tree

is

Exactly our forest-merging approach to spanning tree

but process edges in cost order

Prim’s Algorithm Idea

Idea: Grow a tree by adding an edge from the “known” vertices to the “unknown” vertices. *Pick the edge with the smallest weight that connects “known” to “unknown.”*

Recall Dijkstra picked “edge with closest known distance to source.”

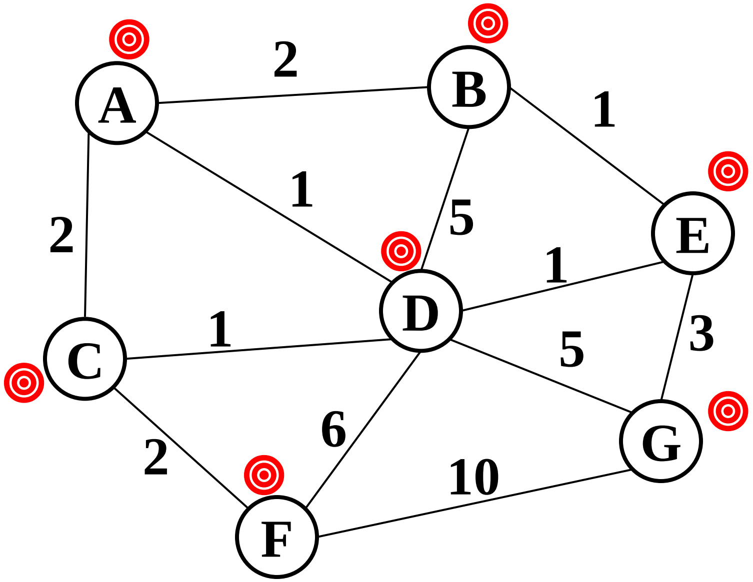
- But that is not what we want here
- Otherwise identical
- Feel free to look back and compare

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`,

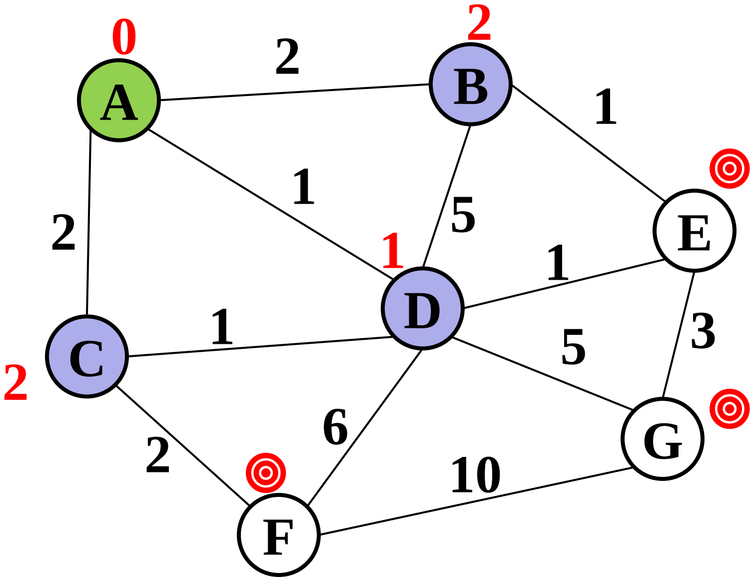
```
if(w < u.cost) {
  u.cost = w;
  u.prev = v;
}
```

Example



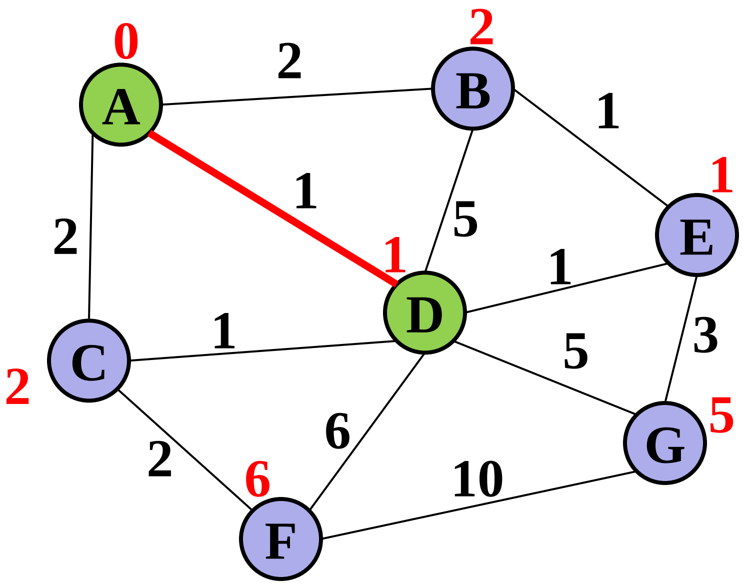
vertex	known?	cost	prev
A		??	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	

Example



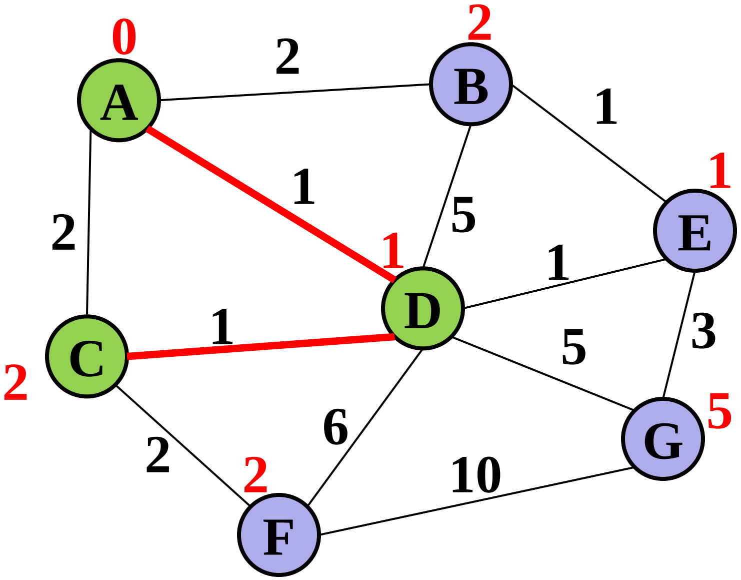
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		2	A
D		1	A
E		??	
F		??	
G		??	

Example



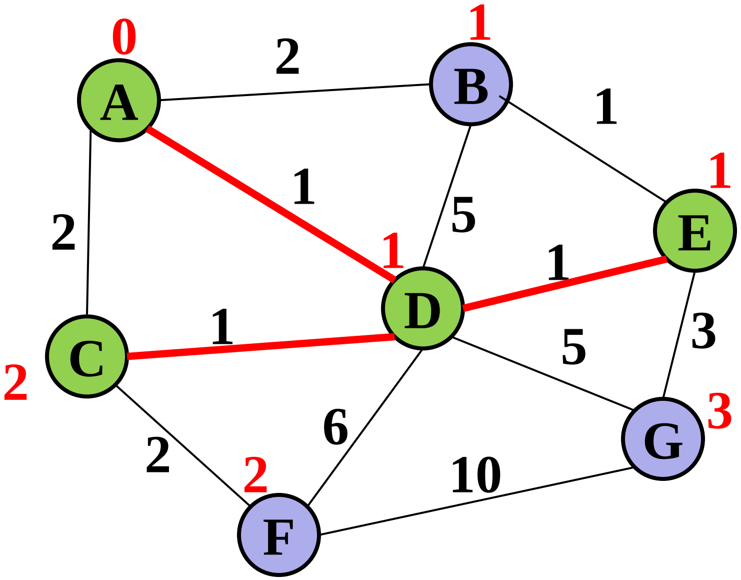
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

Example



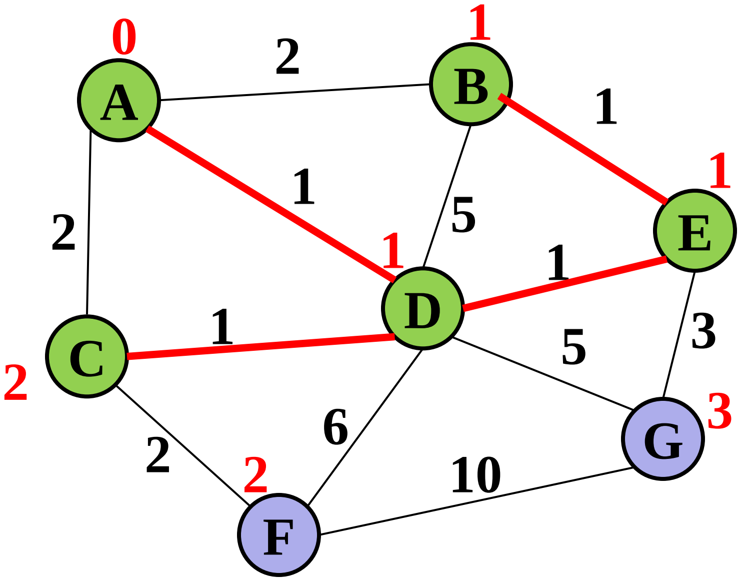
vertex	known?	cost	prev
A	Y	0	
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

Example



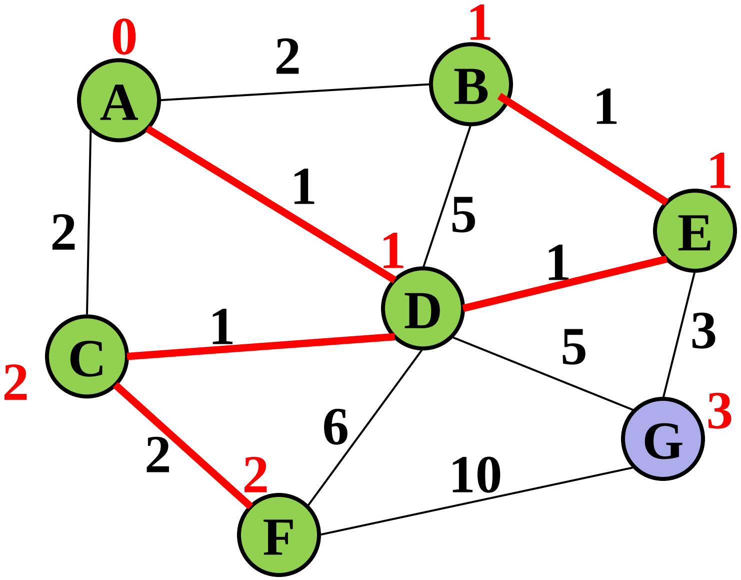
vertex	known?	cost	prev
A	Y	0	
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Example



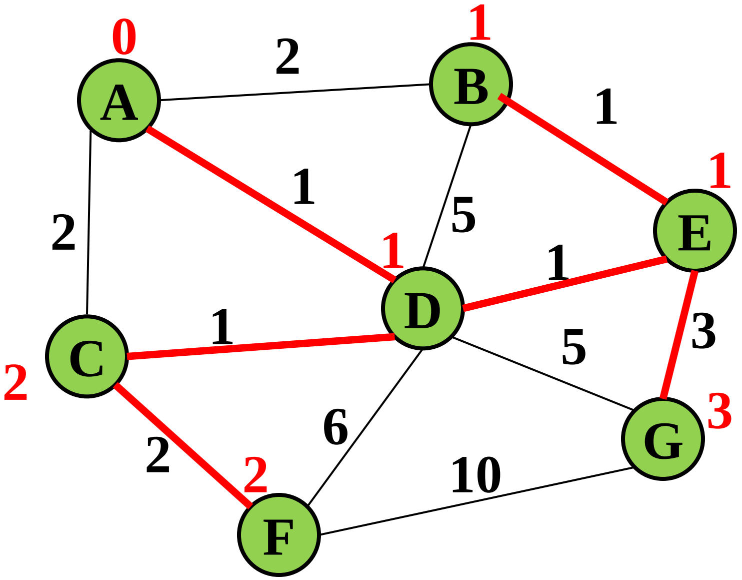
vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Example



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

Example



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

Analysis

- Correctness
  - Intuitively similar to Dijkstra

- Run-time
  - Same as Dijkstra
  - *O*(**|E|** **`log`** **|V|**) using a priority queue

Kruskal’s Algorithm

Idea: Grow a forest out of edges that do not grow a cycle,
just like for the spanning tree problem.

- But now consider the edges in order by weight

So:

- Sort edges: $O(|E| \log |E|) = O(|E| \log |V|)$
- Iterate through edges using union-find for cycle detection $O(|E| \log |V|)$

Somewhat better:

- Floyd’s algorithm to build min-heap with edges $O(|E|)$
- Iterate through edges using union-find for cycle detection and `deleteMin` to get next edge $O(|E| \log |V|)$
- Not better worst-case asymptotically,
  but often stop long before considering all edges

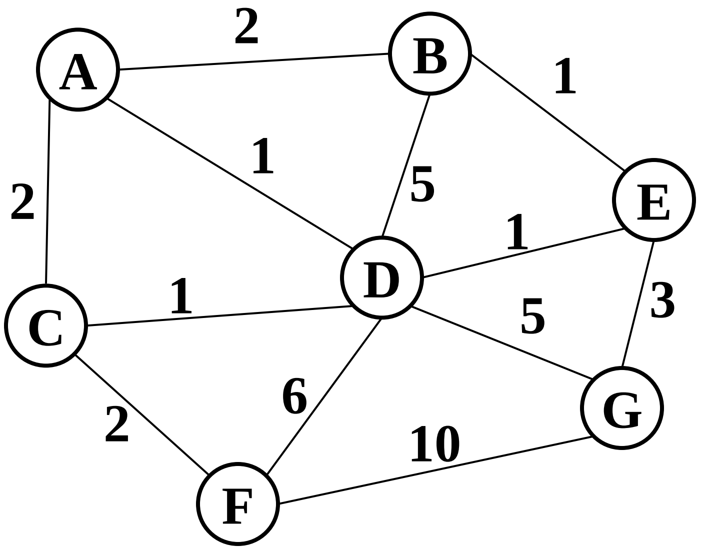
Pseudocode

```
1. Sort edges by weight (better: put in min-heap)
2. Each node in its own set
3. While output size < |V|-1
   – Consider next smallest edge (u,v)
   – if find(u,v) indicates u and v are in different sets
       • output (u,v)
       • union(u,v)
```

Recall invariant:

`u` and `v` in same set if and only if connected in output-so-far

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)

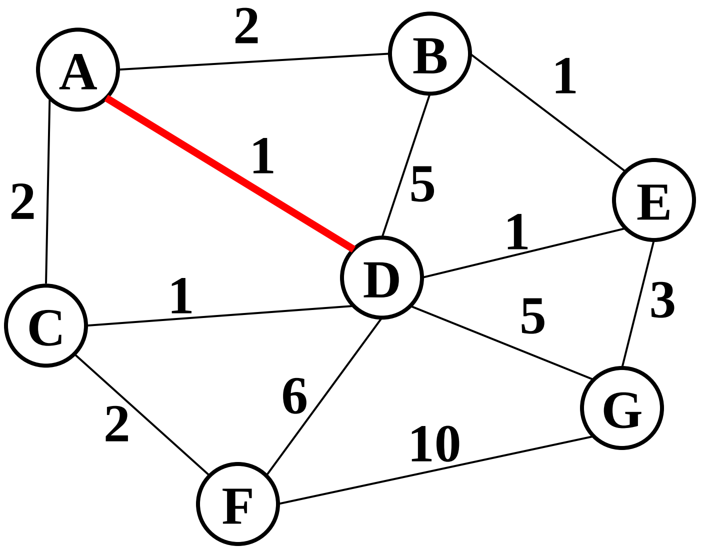
10: (F,G)

Sets: (A) (B) (C) (D) (E) (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)

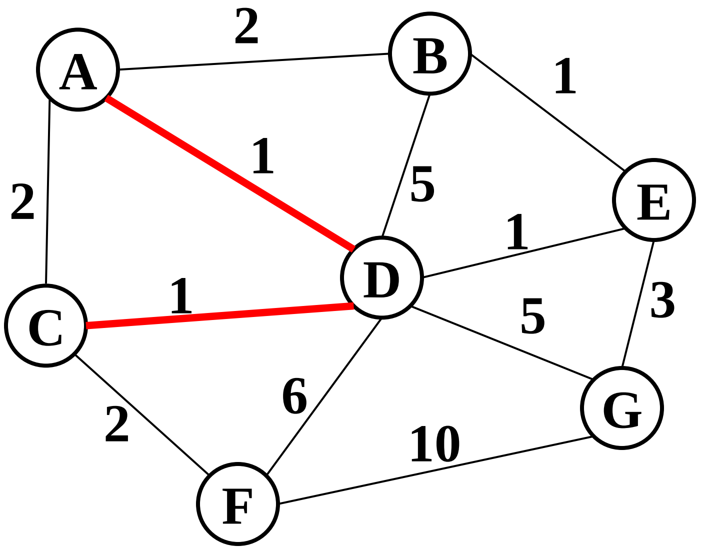
10: (F,G)

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

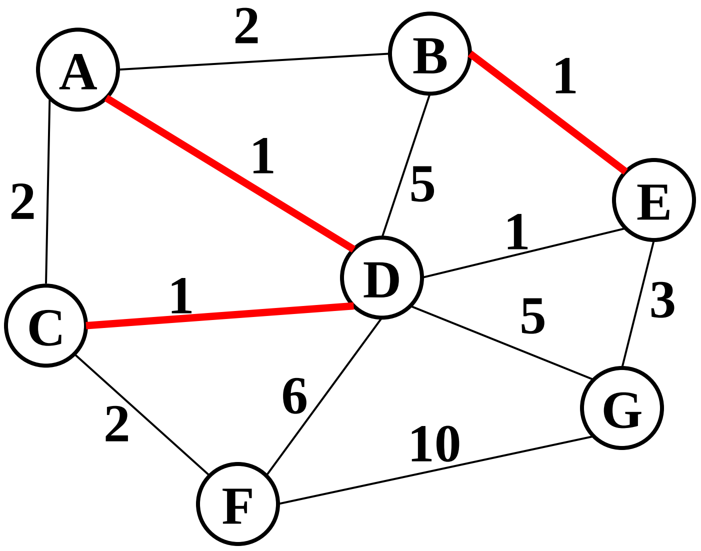
10: (F,G)

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

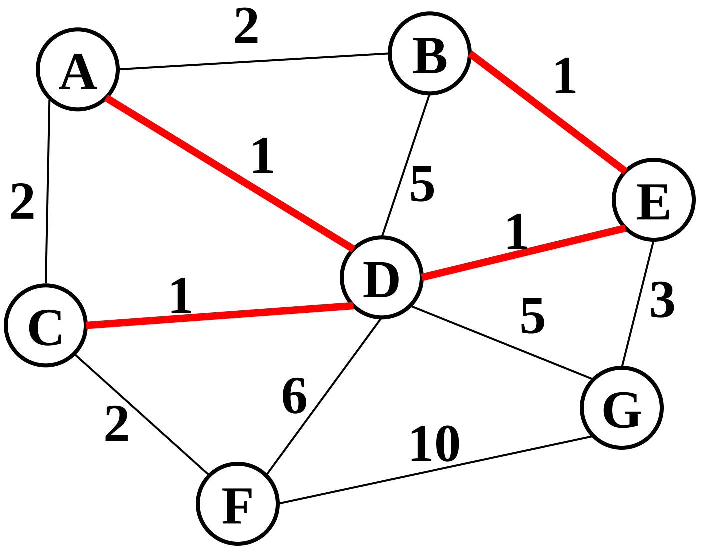
10: (F,G)

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

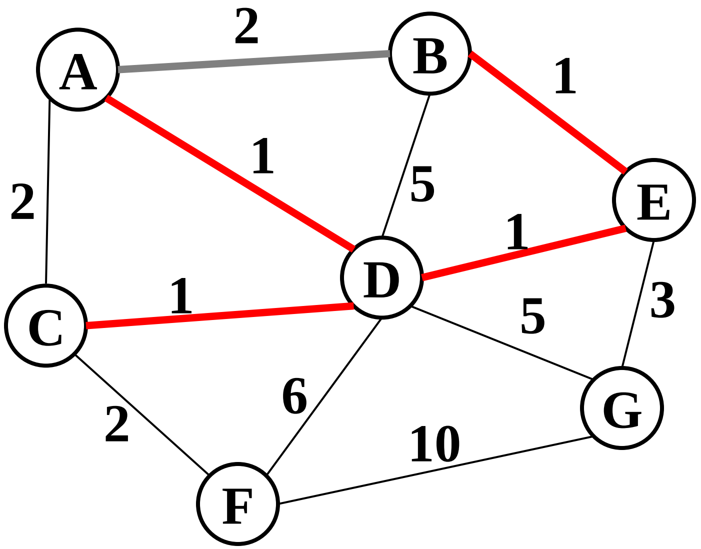
10: (F,G)

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

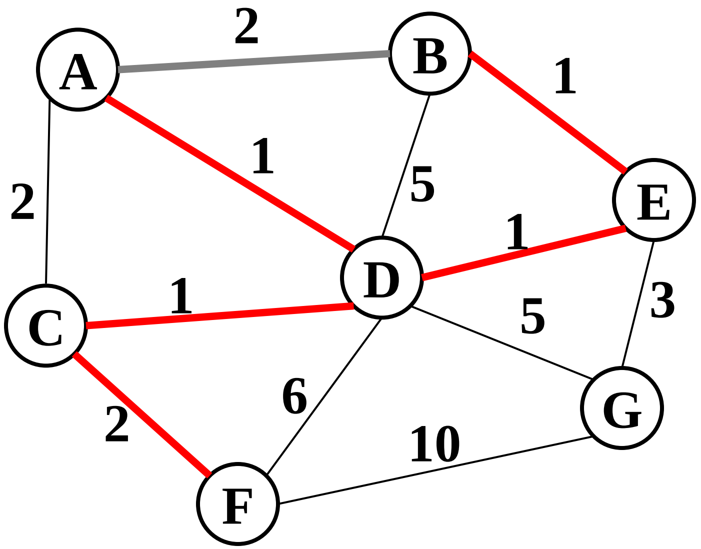
10: (F,G)

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

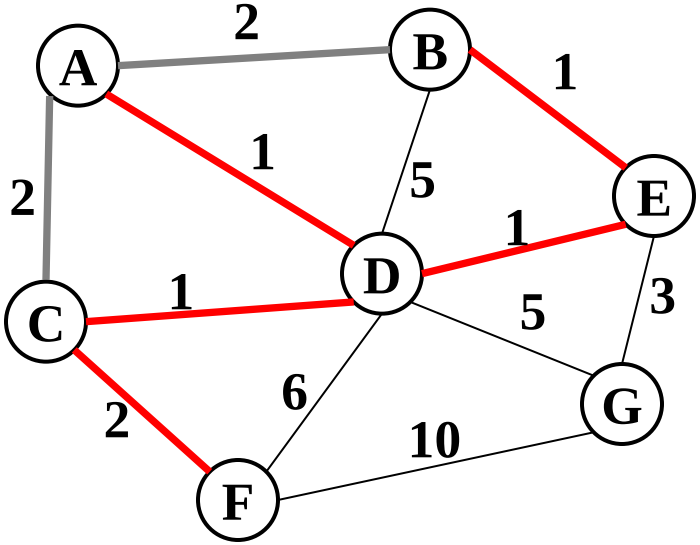
10: (F,G)

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

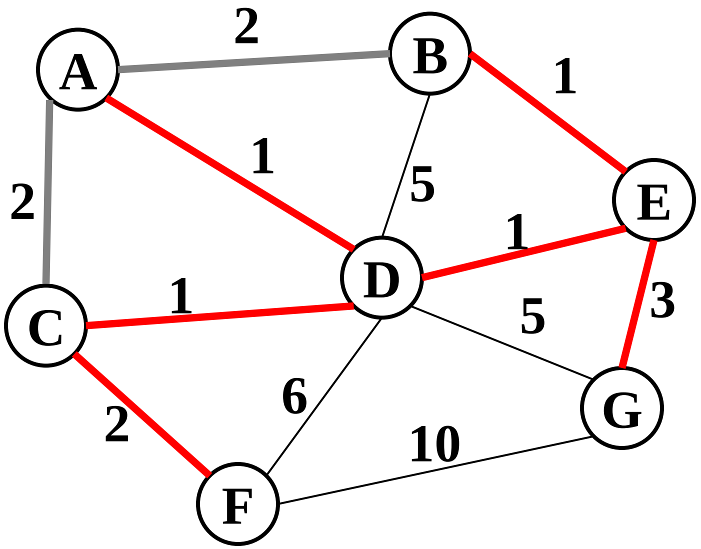
10: (F,G)

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

Sets: (A, B, C, D, E, 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

Analysis

- Correctness
  - That it is a spanning tree
    - When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
    - The graph is connected, we consider all edges
  - That it is minimum
    - By induction
    - At every step, the output is a subset of a minimum tree

- Run-time
  - $O(\mathbf{|E|}\ \mathtt{log}\ \mathbf{|V|})$