Announcements

- HW4 due Friday

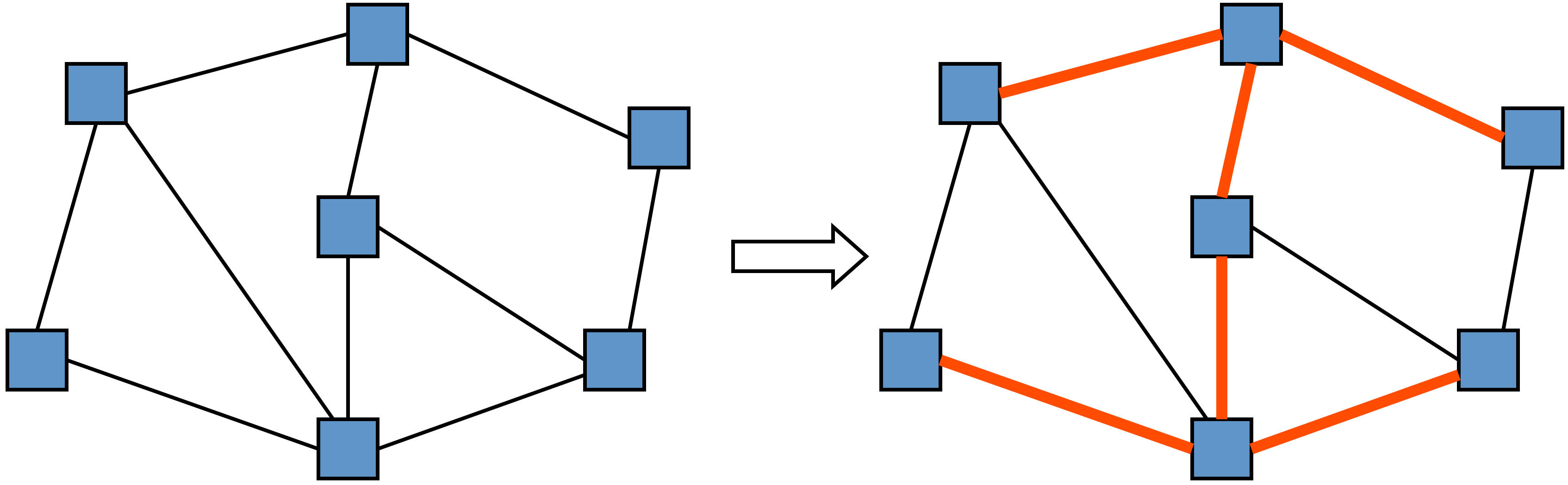
CSE373: Data Structures & Algorithms
Minimum Spanning Trees

Hunter Zahn

Summer 2016

Spanning Trees

- A simple problem: 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. 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
   – So **|E| >= |V|-1**

4. 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

- 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: 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:

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 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
    }
}
```

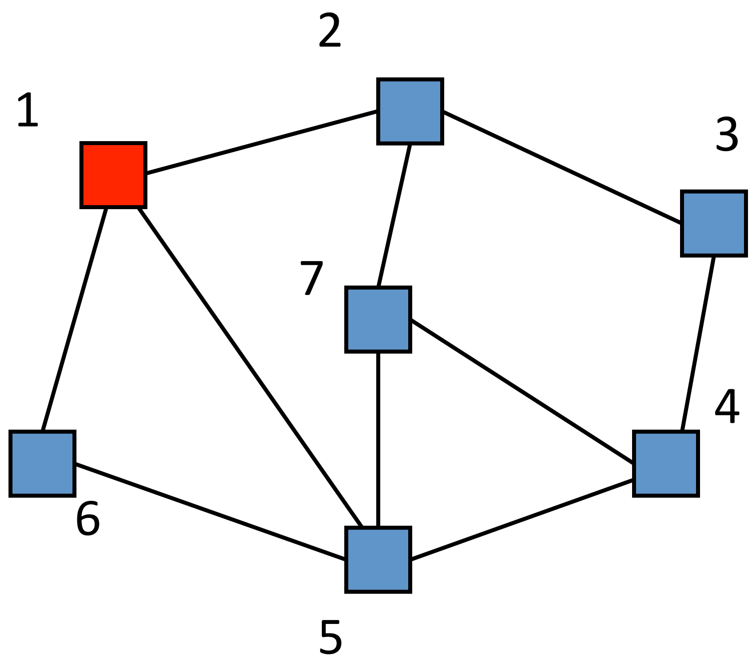
Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|E|)$

Example

Stack

f(1)



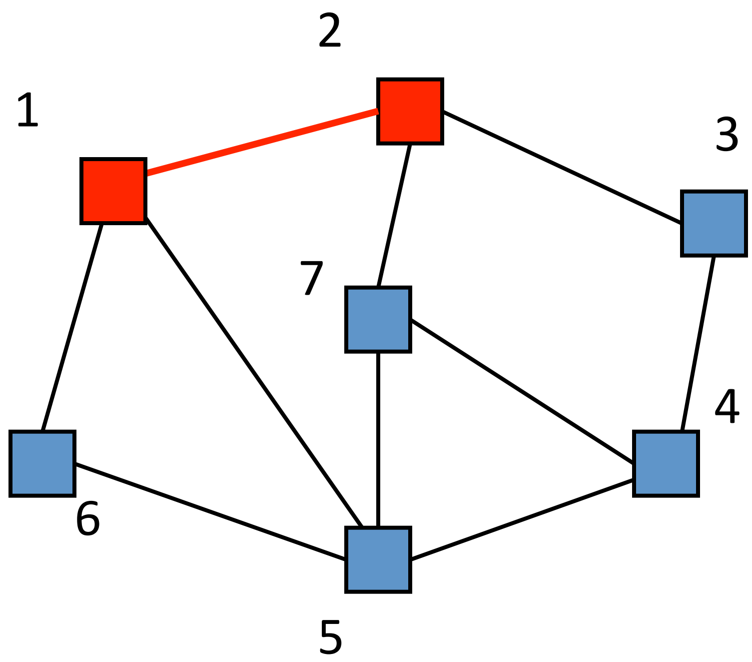
Output:

Example

Stack (bottom)

f(1)

f(2)



Output: (1,2)

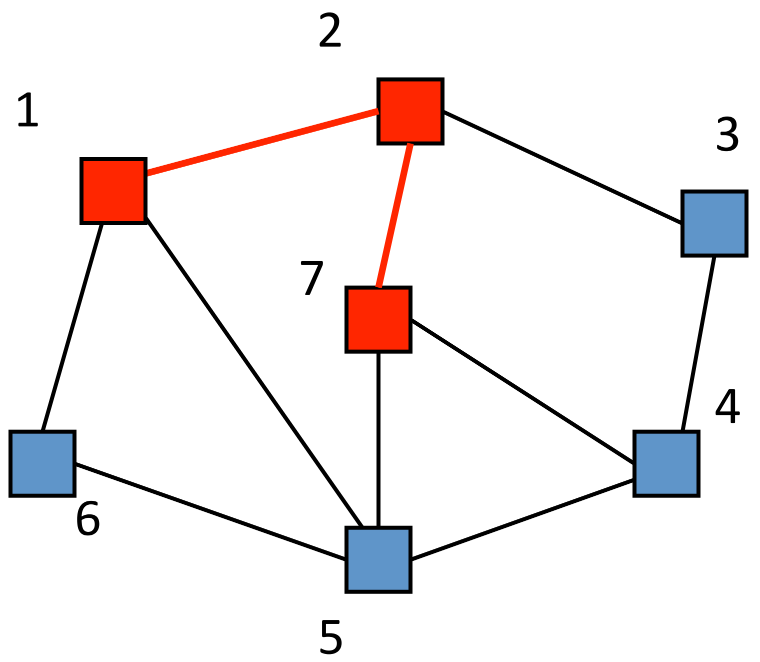
Example

Stack
(bottom)

f(1)

f(2)

f(7)



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

Example

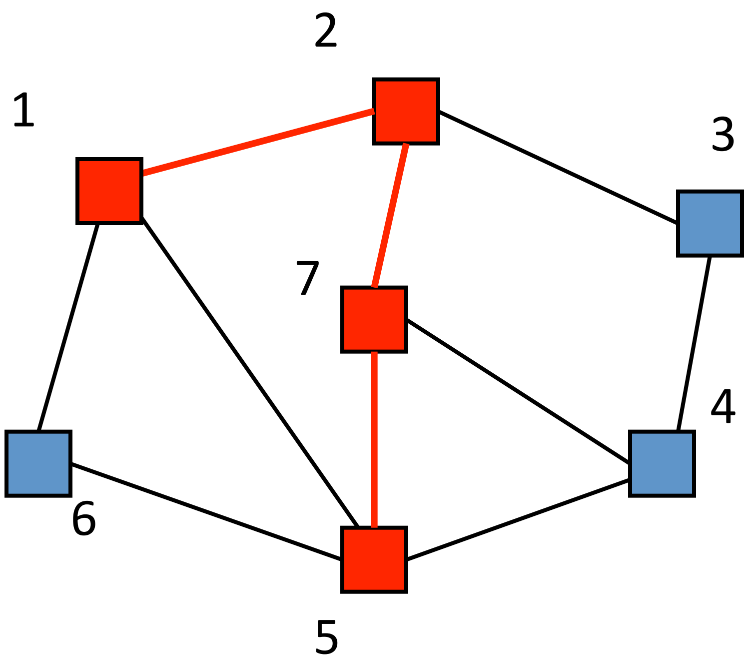
Stack
(bottom)

f(1)

f(2)

f(7)

f(5)



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

Example

Stack
(bottom)

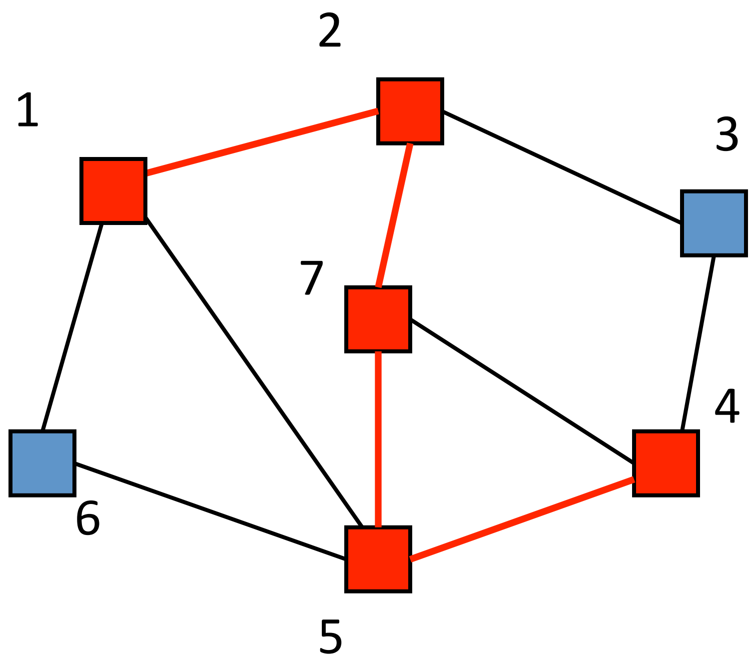
f(1)

f(2)

f(7)

f(5)

f(4)



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

Example

Stack
(bottom)

f(1)

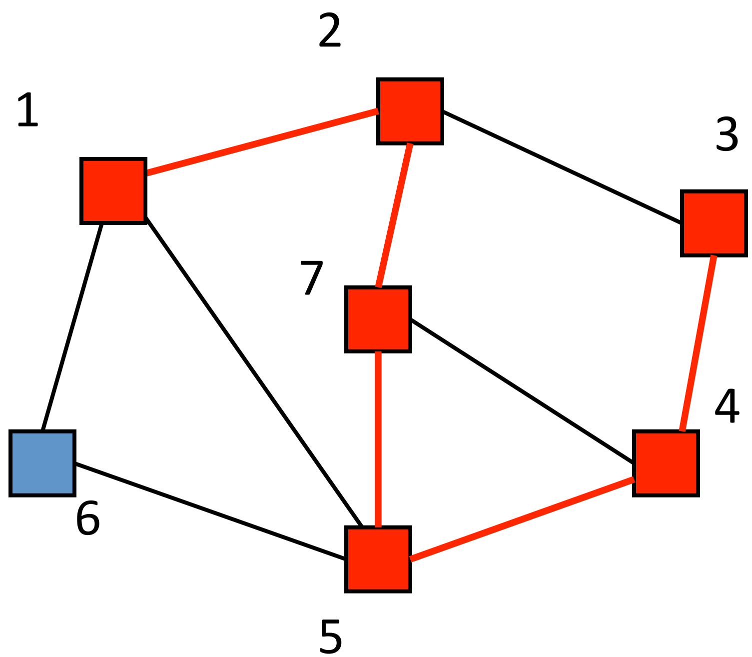
f(2)

f(7)

f(5)

f(4)

f(3)



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

Example

Stack
(bottom)

f(1)

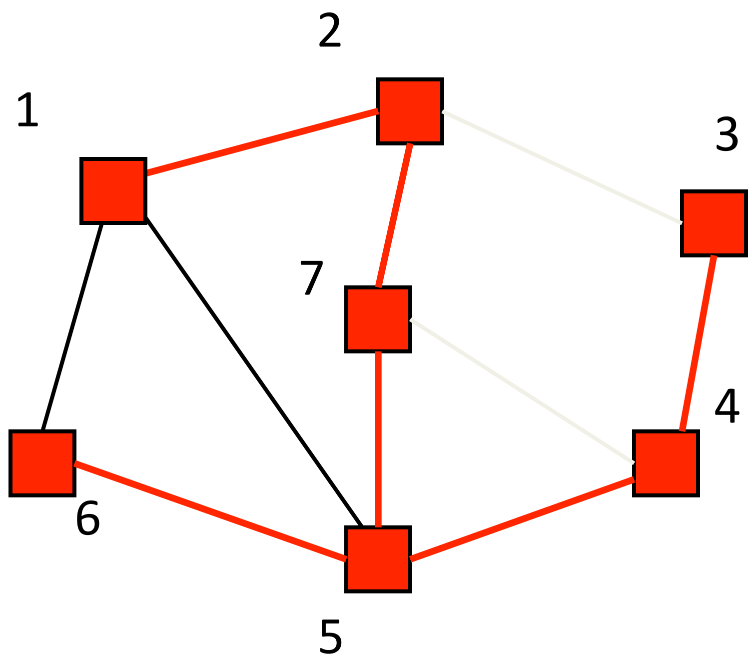
f(2)

f(7)

f(5)

f(4) f(6)

f(3)

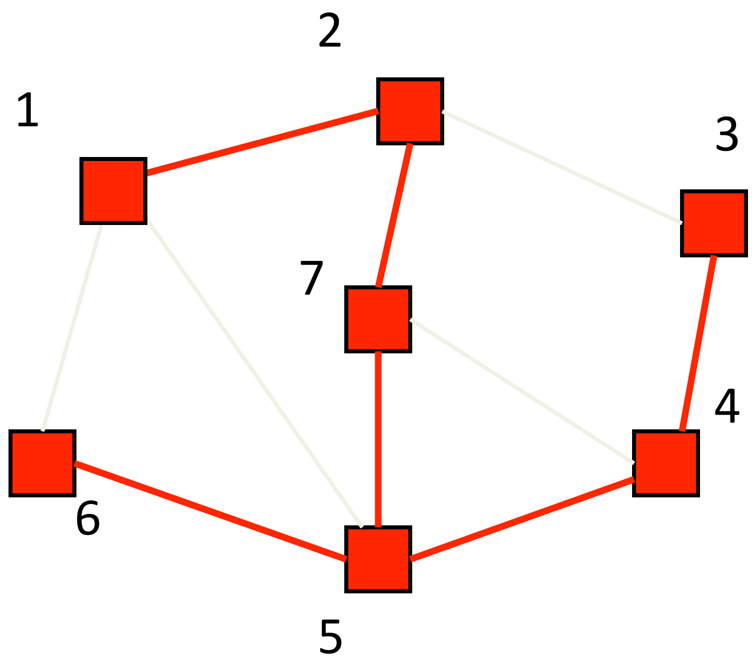


Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

Example

Stack
(bottom)

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
  - Else it would have created a cycle
- The graph is connected, so we reach all vertices

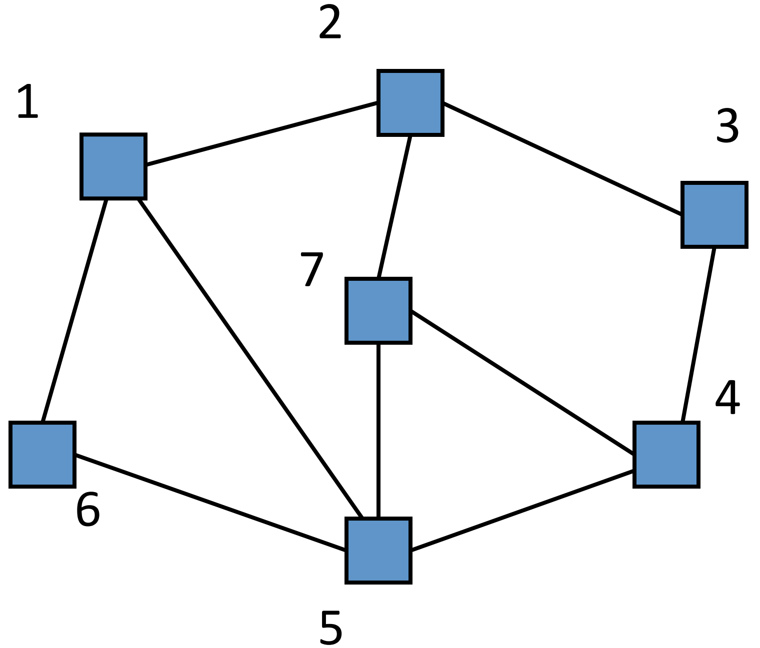
Efficiency:

- 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)

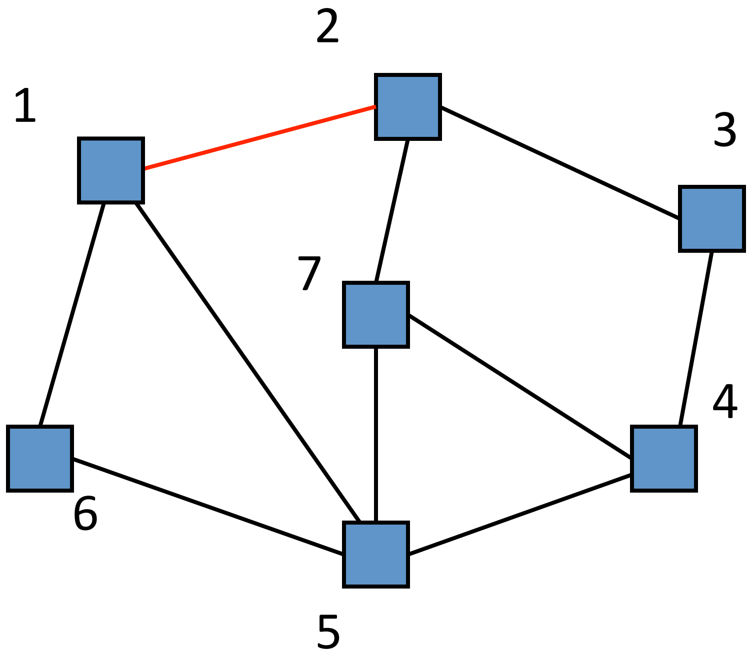


Output:

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)

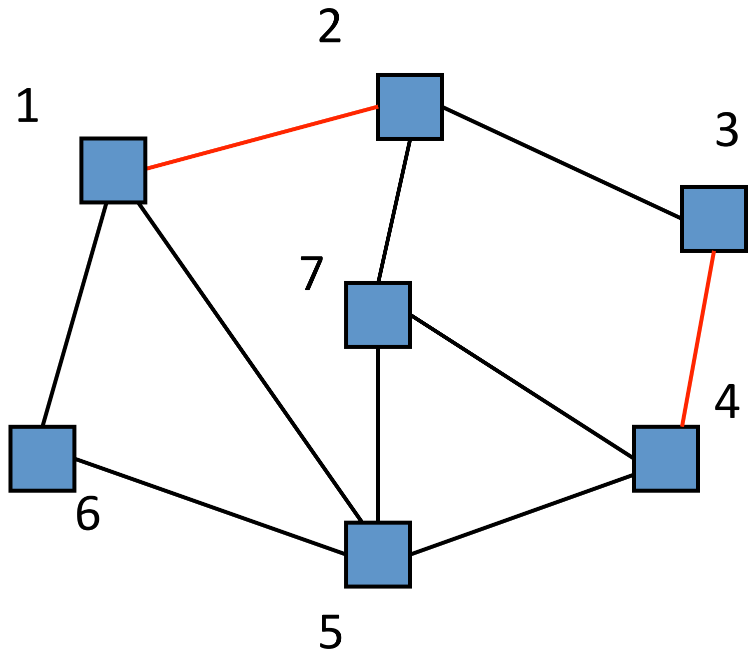


Output: (1,2)

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)

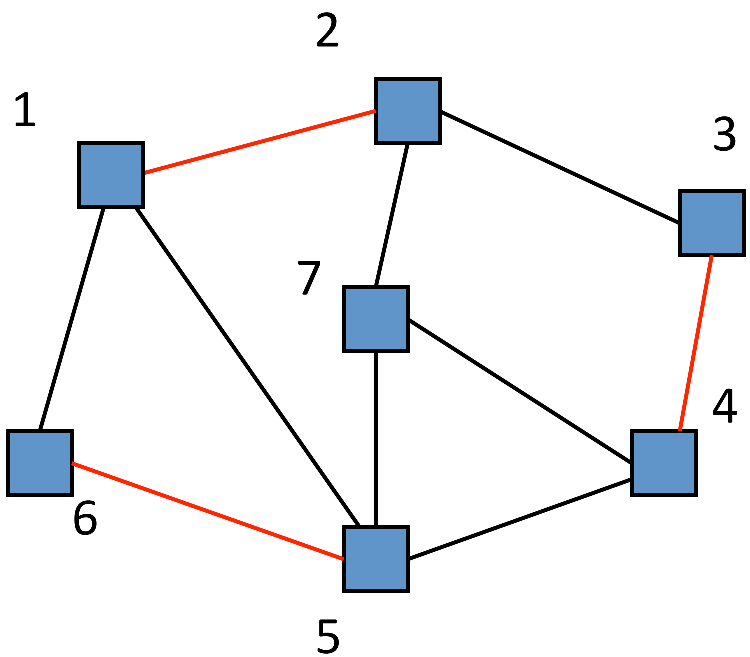


Output: (1,2), (3,4)

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)

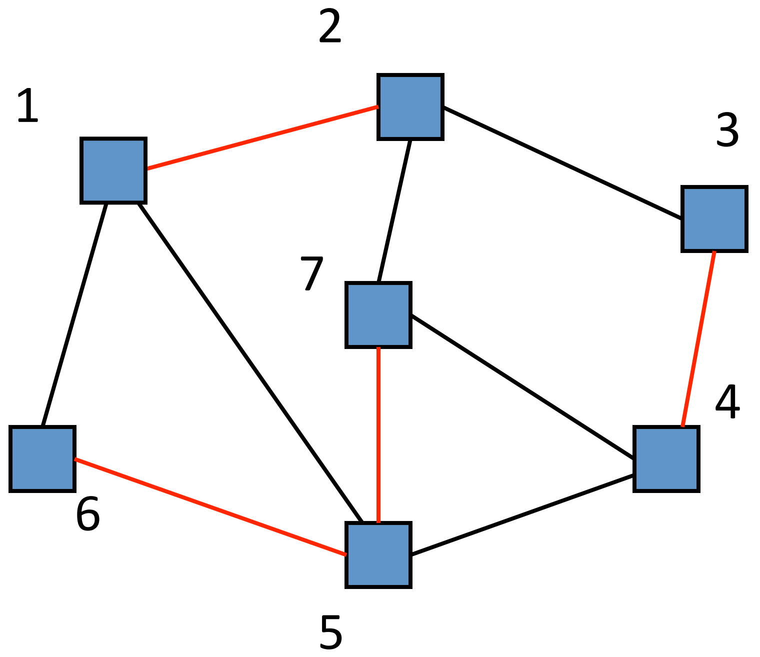


Output: (1,2), (3,4), (5,6),

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)

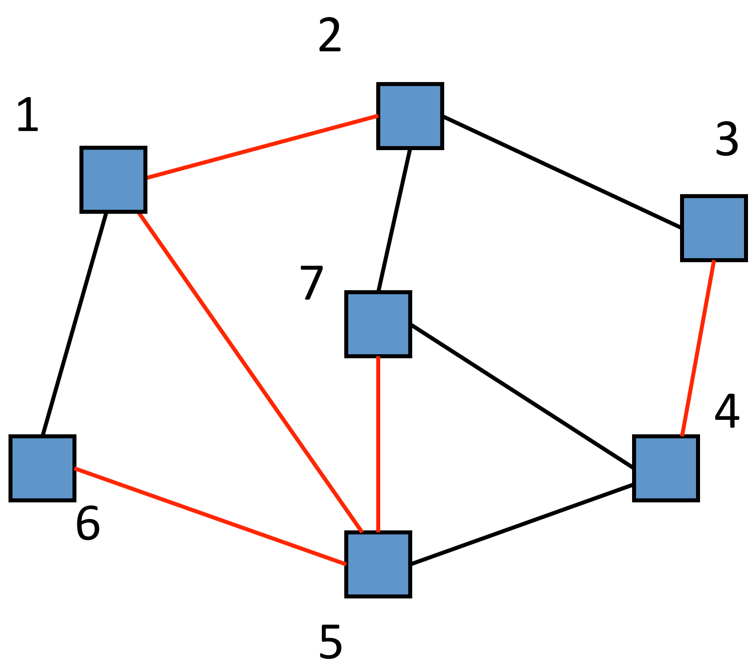


Output: (1,2), (3,4), (5,6), (5,7)

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)

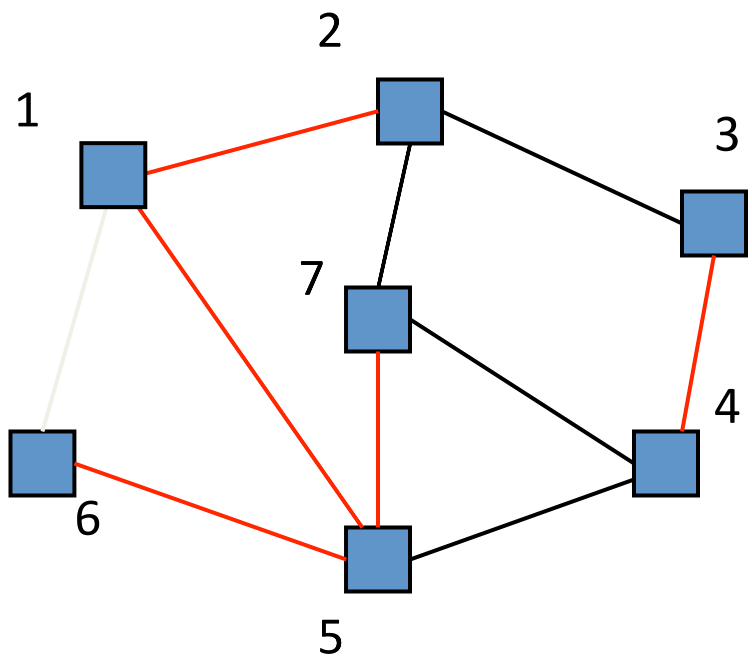


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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)

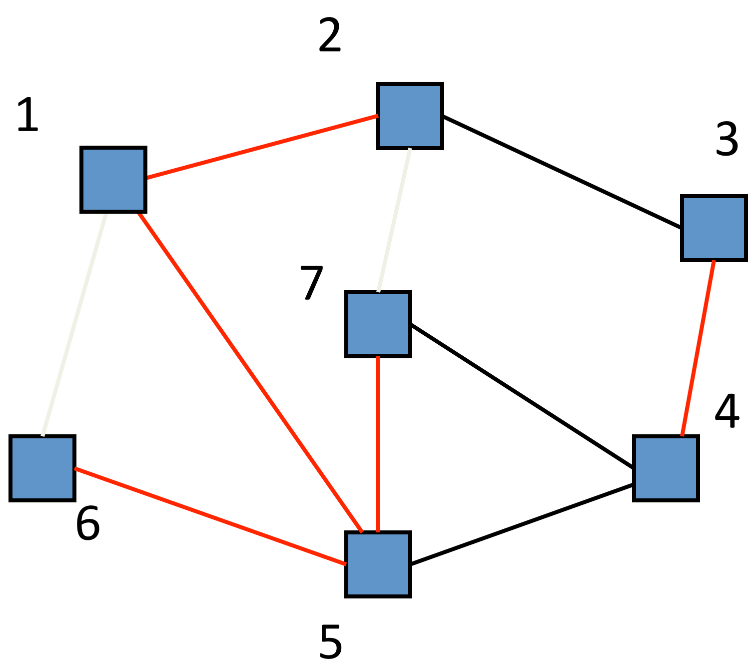


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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)

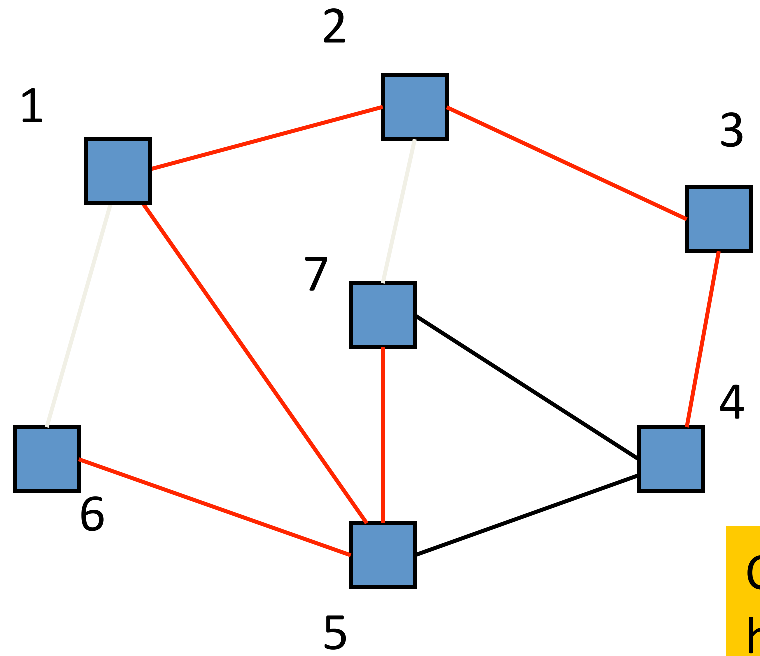


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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 we know: use union-find!
  - Initially, each item is in its own 1-element set
  - Union sets when we add an edge that connects them
  - Stop when we have one set

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)`** equals **`find(v)`**, then do not add the edge
  - Else add the edge and **`union(find(u),find(v))`**
  - $O(|E|)$ operations that are almost $O(1)$ amortized

Summary So Far

The spanning-tree problem

- Add nodes to partial tree approach is $O(|E|)$
- Add acyclic edges approach is *almost* $O(|E|)$
  - Using union-find "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 2nd 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"

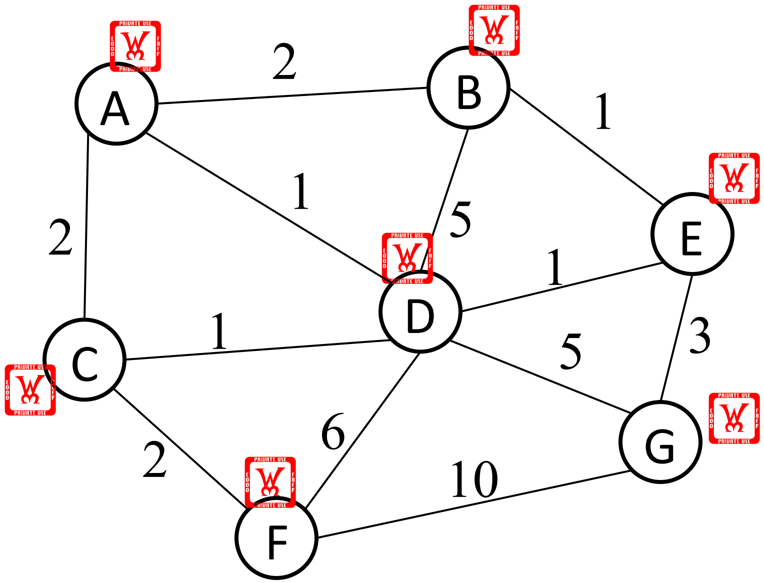
- That is not what we want here
- Otherwise identical (!)

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**,

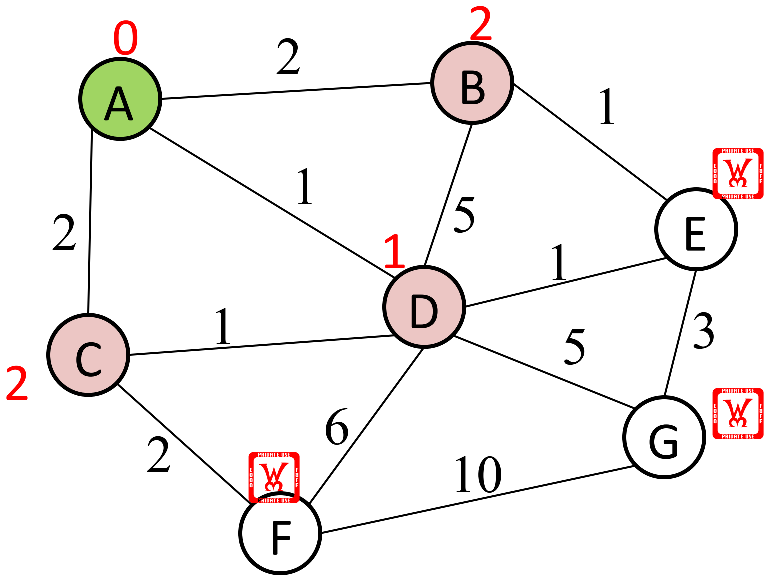
```
if(w < u.cost) {
  u.cost = w;
  u.prev = v;
}
```

Example



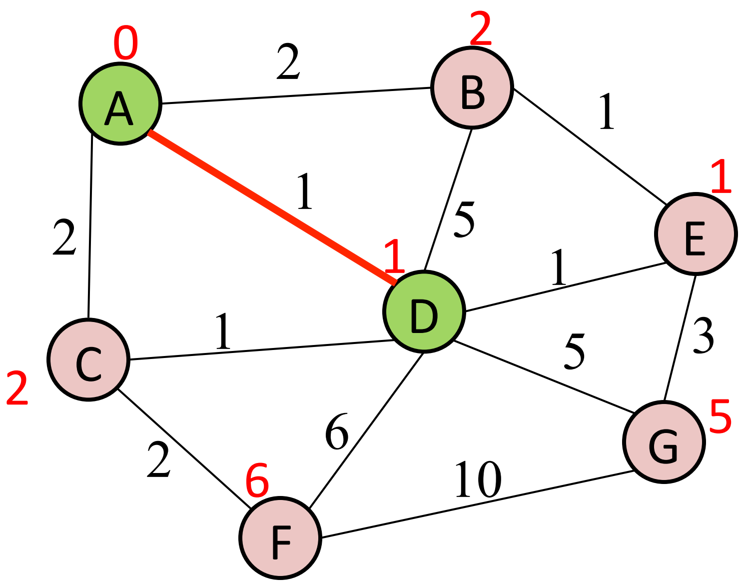
vertex	known?	cost	prev
A		??	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	

Example



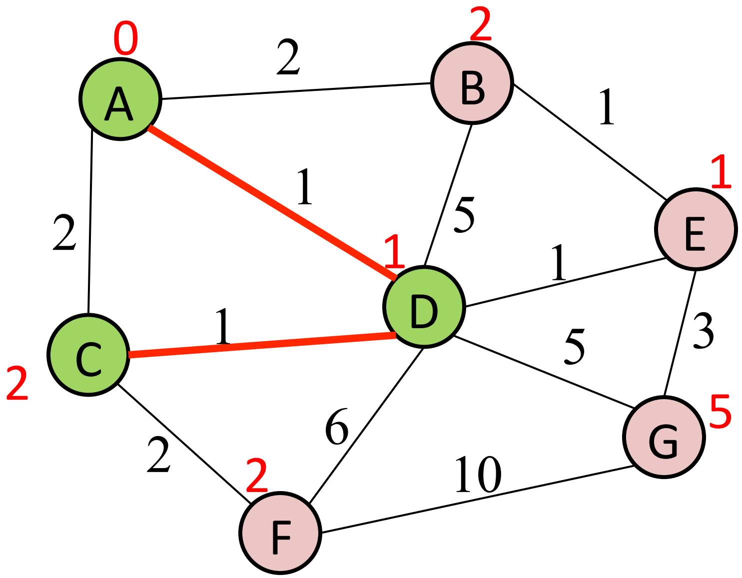
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		2	A
D		1	A
E		??	
F		??	
G		??	

Example



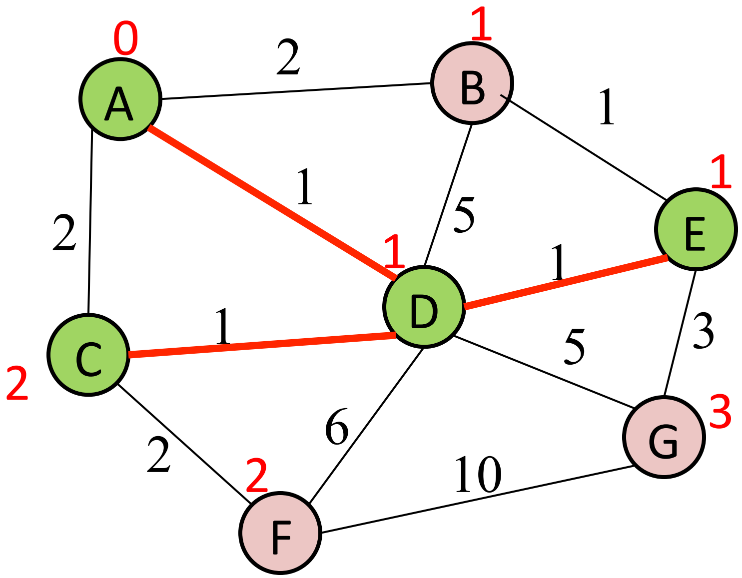
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

Example



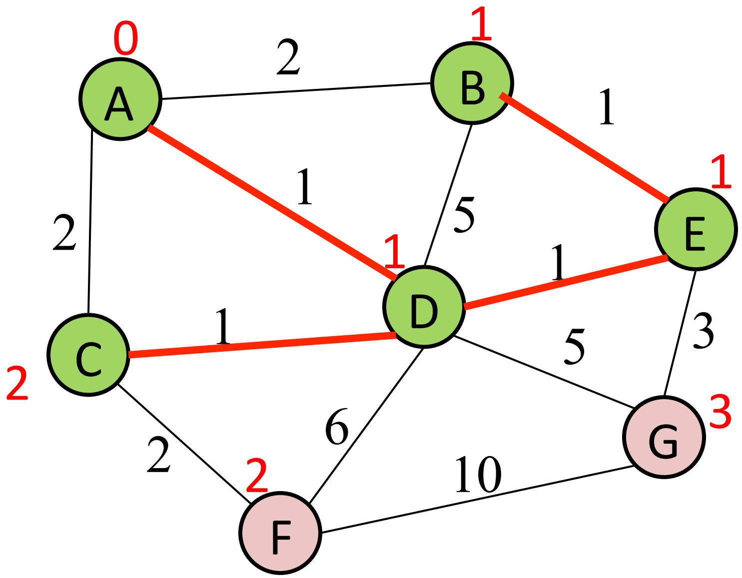
vertex	known?	cost	prev
A	Y	0	
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

Example



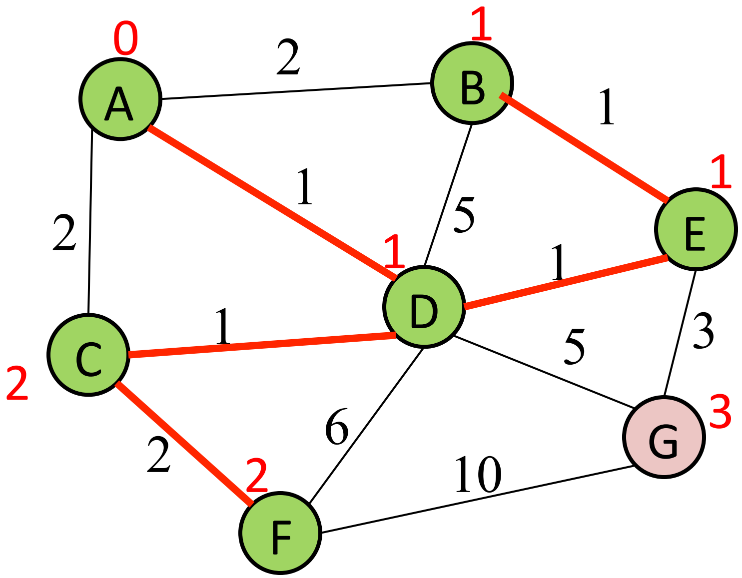
vertex	known?	cost	prev
A	Y	0	
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Example



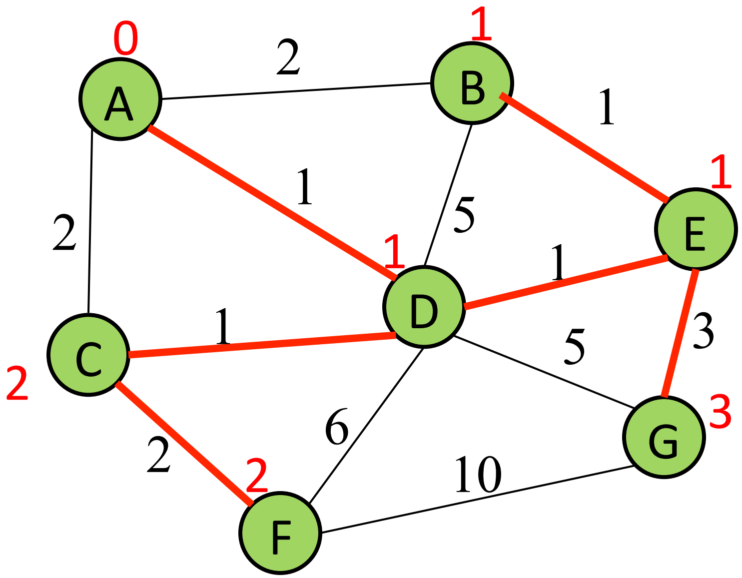
vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

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 ??
  - A bit tricky
  - Intuitively similar to Dijkstra

- Run-time
  - Same as Dijkstra
  - O(`|E|log|V|`) using a priority queue
    - Costs/priorities are just edge-costs, not path-costs

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|`**) (next course topic)
- Iterate through edges using union-find for cycle detection almost O(**`|E|`**)

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|E|`**)
- Not better *worst-case* asymptotically, but often stop long before considering all edges

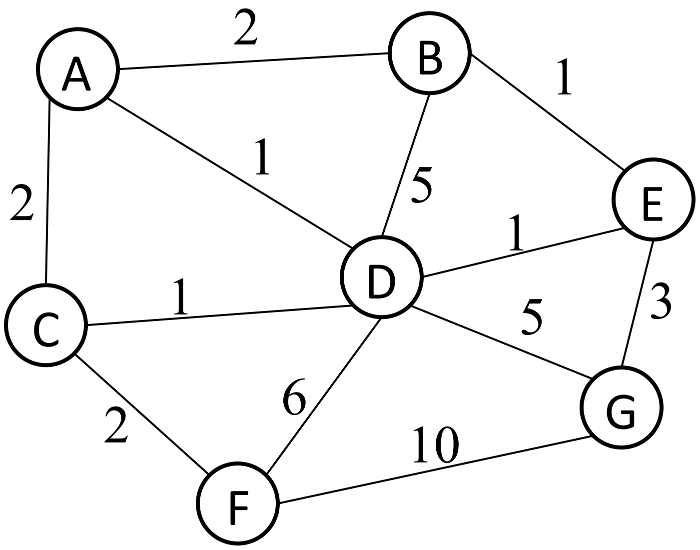
Pseudocode

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(find(u),find(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)

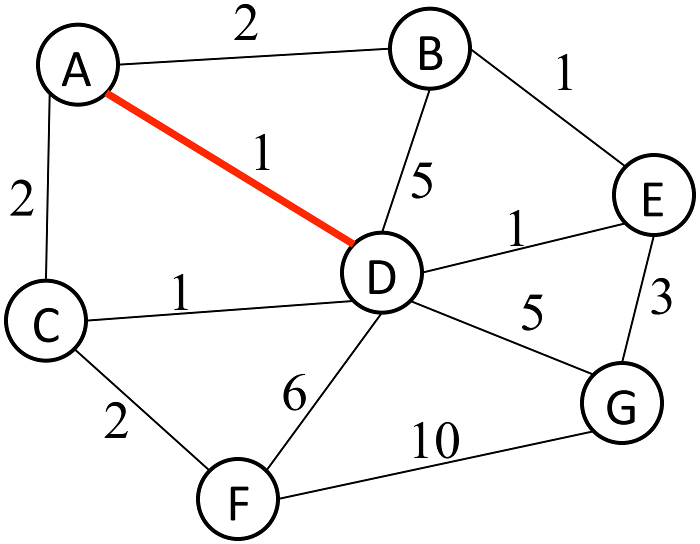
6: (D,F)

10: (F,G)

Output:

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

Example



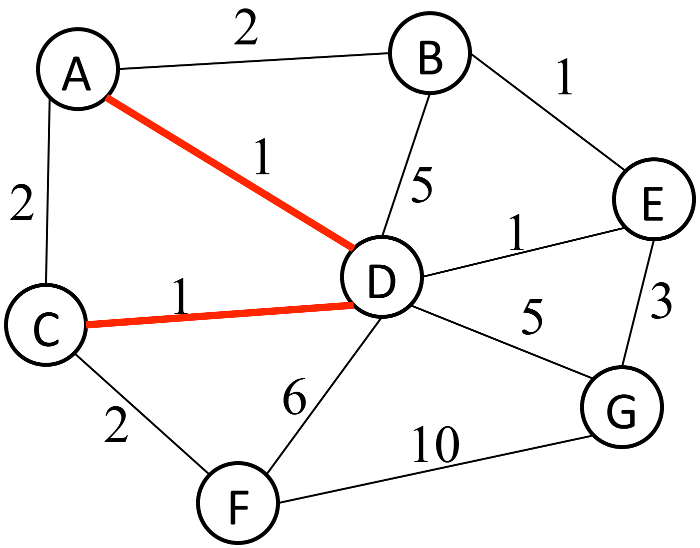
Edges in sorted order:

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)

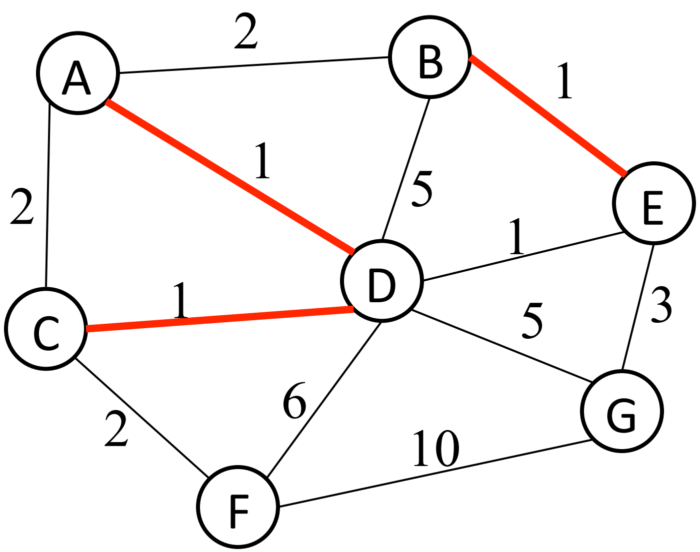
6: (D,F)

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)

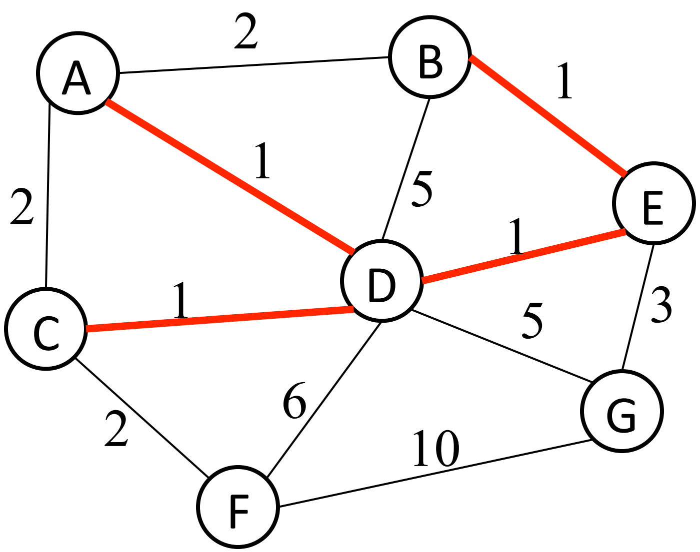
6: (D,F)

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)

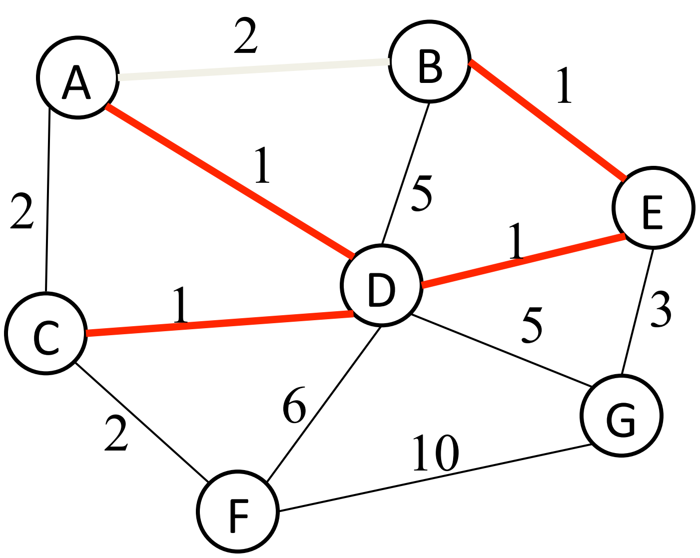
6: (D,F)

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)

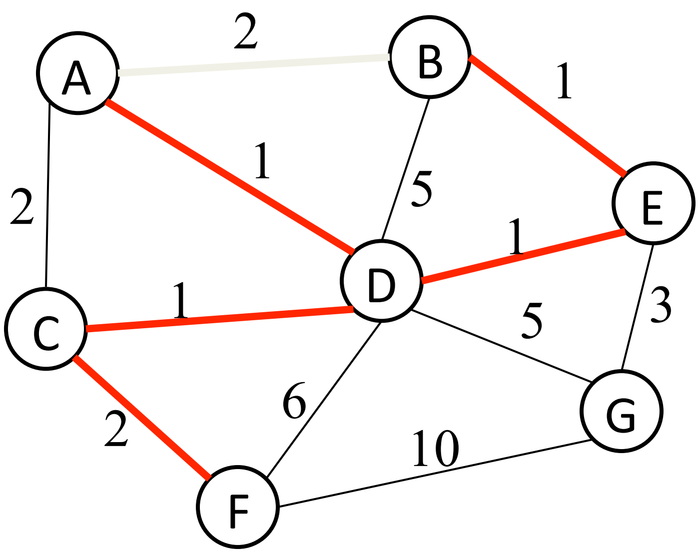
6: (D,F)

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)

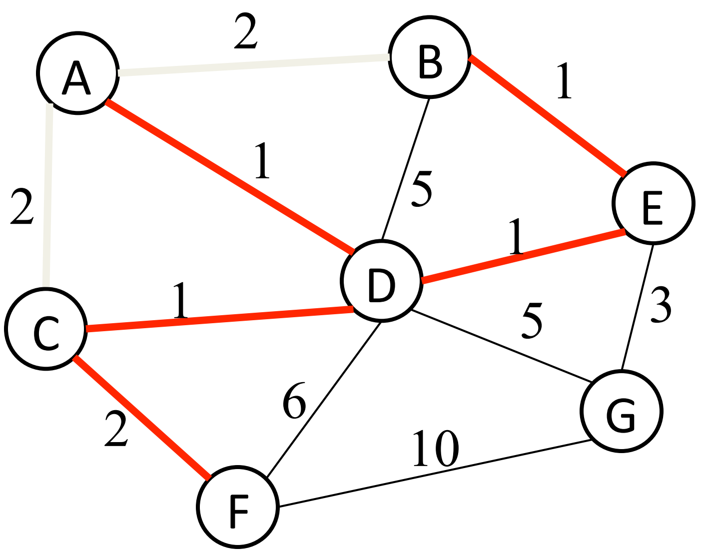
6: (D,F)

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)

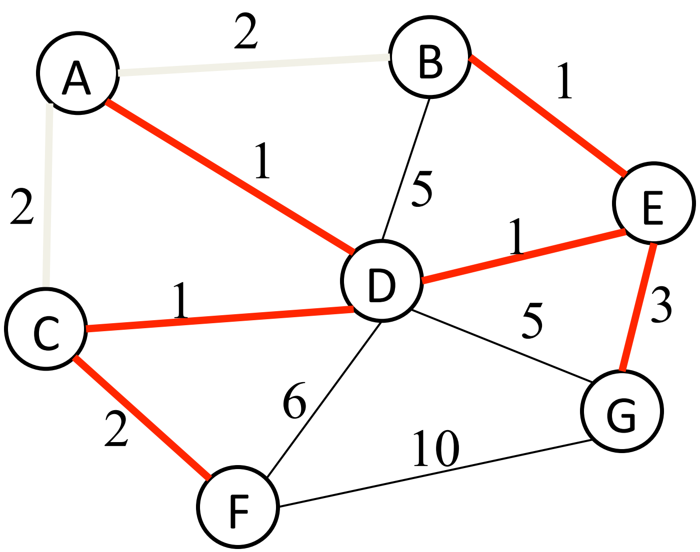
6: (D,F)

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: Correctness

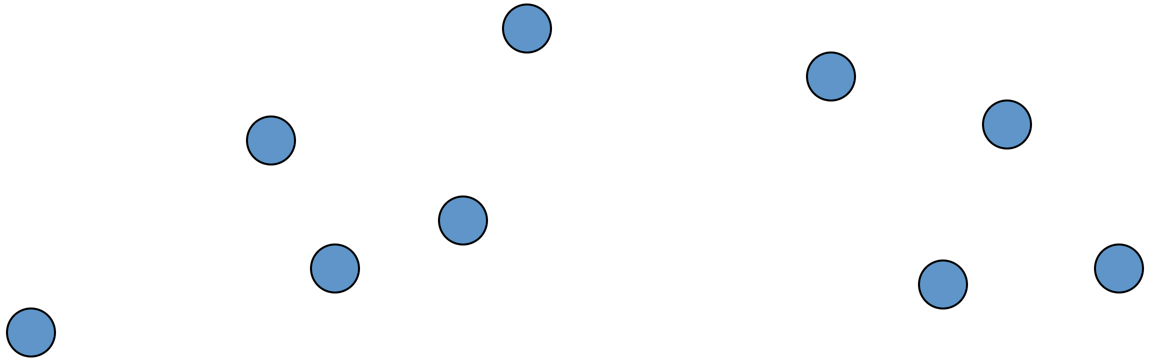
It clearly generates a spanning tree. Call it T_K.

Suppose T_K is *not* minimum:

- Pick another spanning tree T_{min} with *lower cost* than T_K
- Pick the smallest edge $e_1=(u,v)$ in T_K that is not in T_{min}
- T_{min} already has a path p in T_{min} from u to v
  - $\Rightarrow$ Adding e_1 to T_{min} will create a cycle in T_{min}
- Pick an edge e_2 in p that Kruskal's algorithm considered *after* adding e_1 (must exist: u and v unconnected when e_1 considered)
  - $\Rightarrow$ $cost(e_2) \geq cost(e_1)$
  - $\Rightarrow$ can replace e_2 with e_1 in T_{min} without increasing cost!
- Keep doing this until T_{min} is identical to T_K
  - $\Rightarrow$ T_K must also be minimal – contradiction!

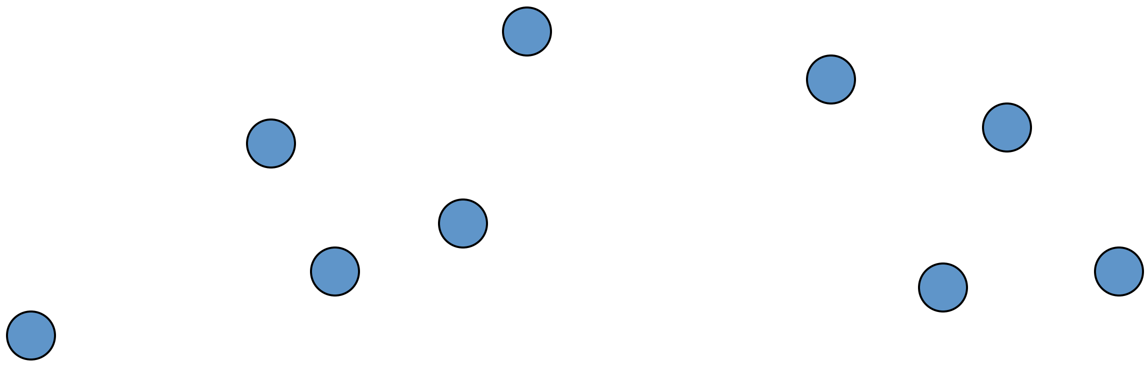
MST Application: Clustering

- Given a collection of points in an r-dimensional space, and an integer K, divide the points into K sets that are closest together

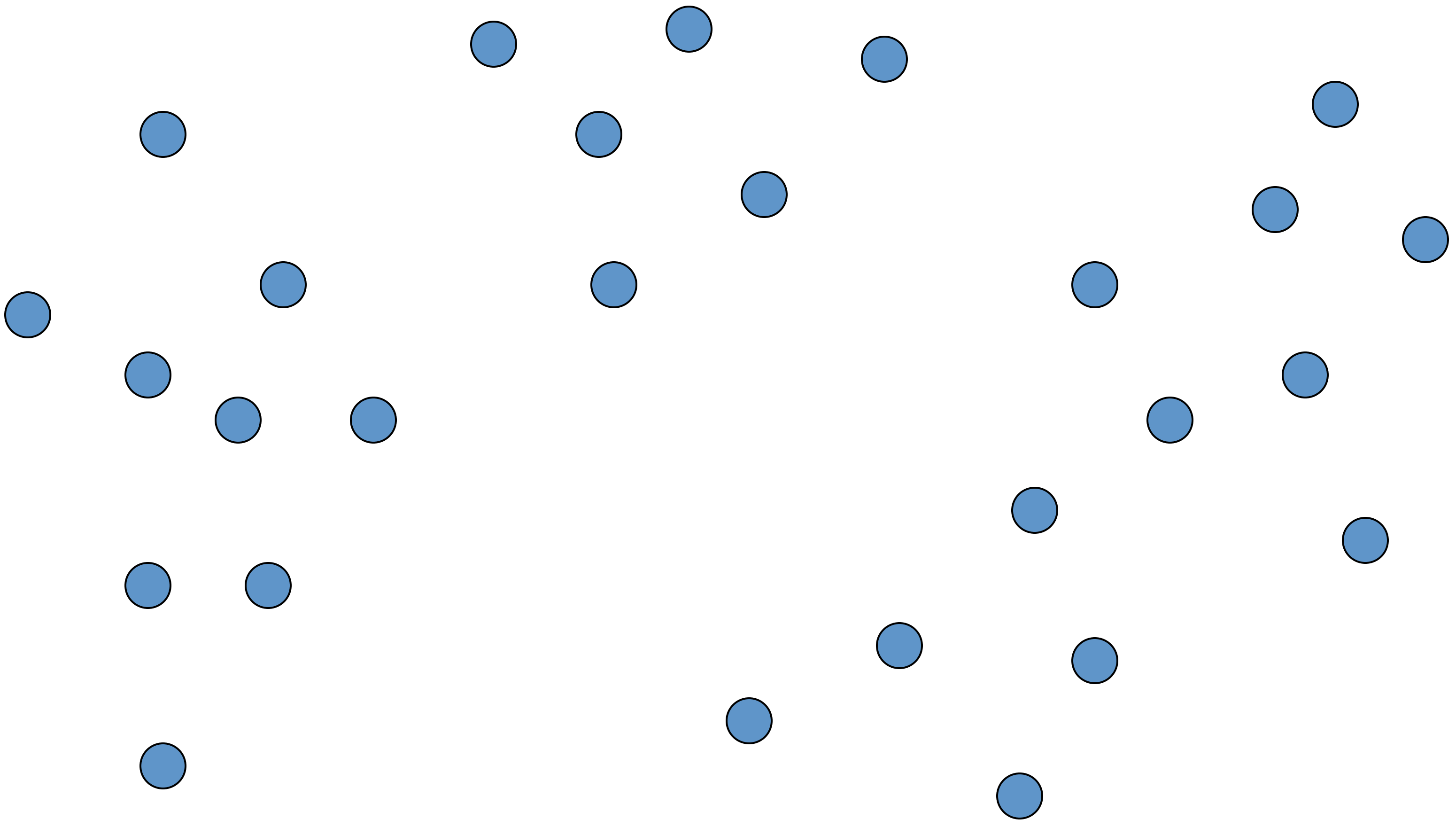


Distance clustering

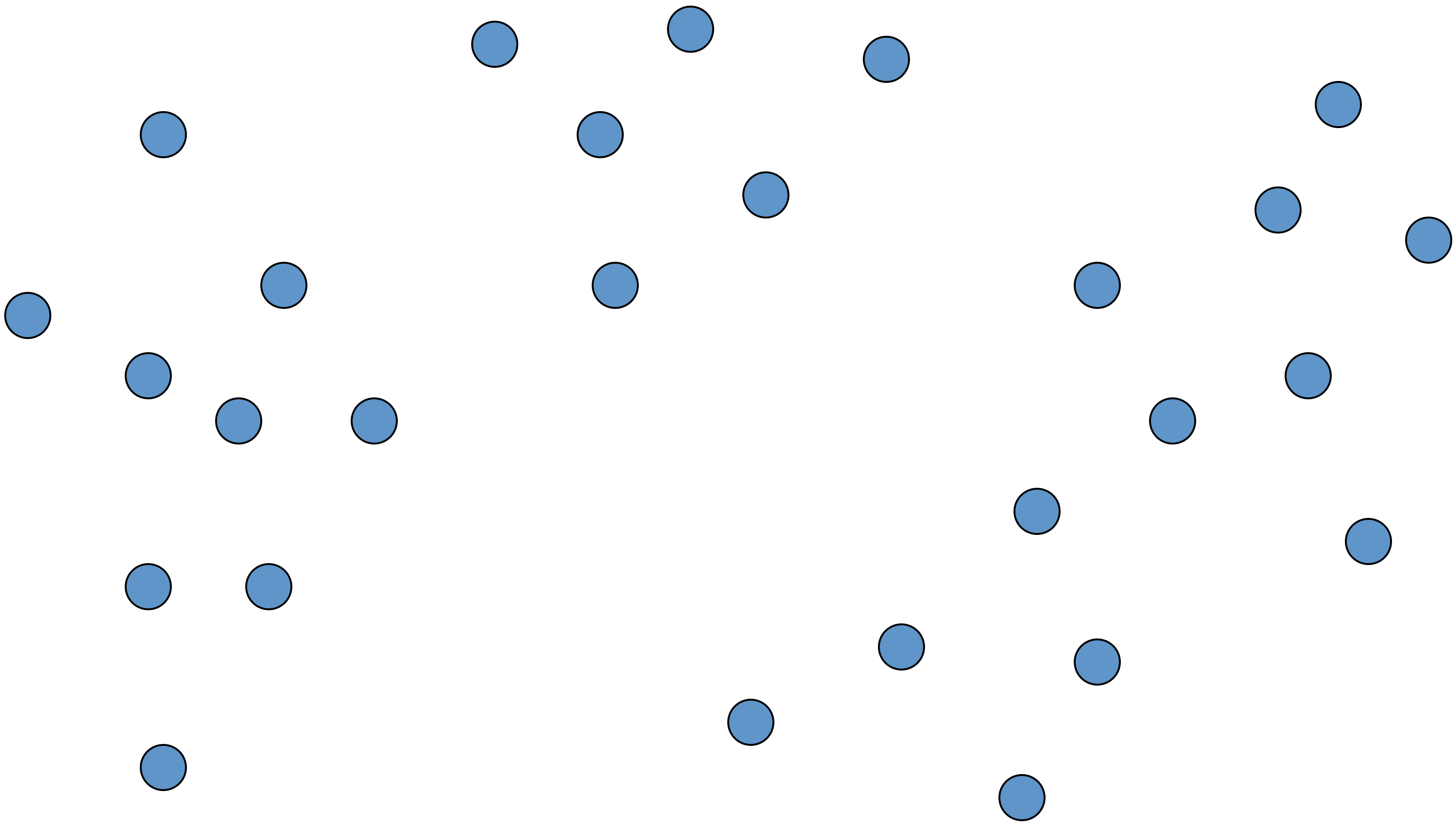
- Divide the data set into K subsets to maximize the distance between any pair of sets
  - $dist(S_1, S_2) = \min \{dist(x, y) \mid x \text{ in } S_1, y \text{ in } S_2\}$



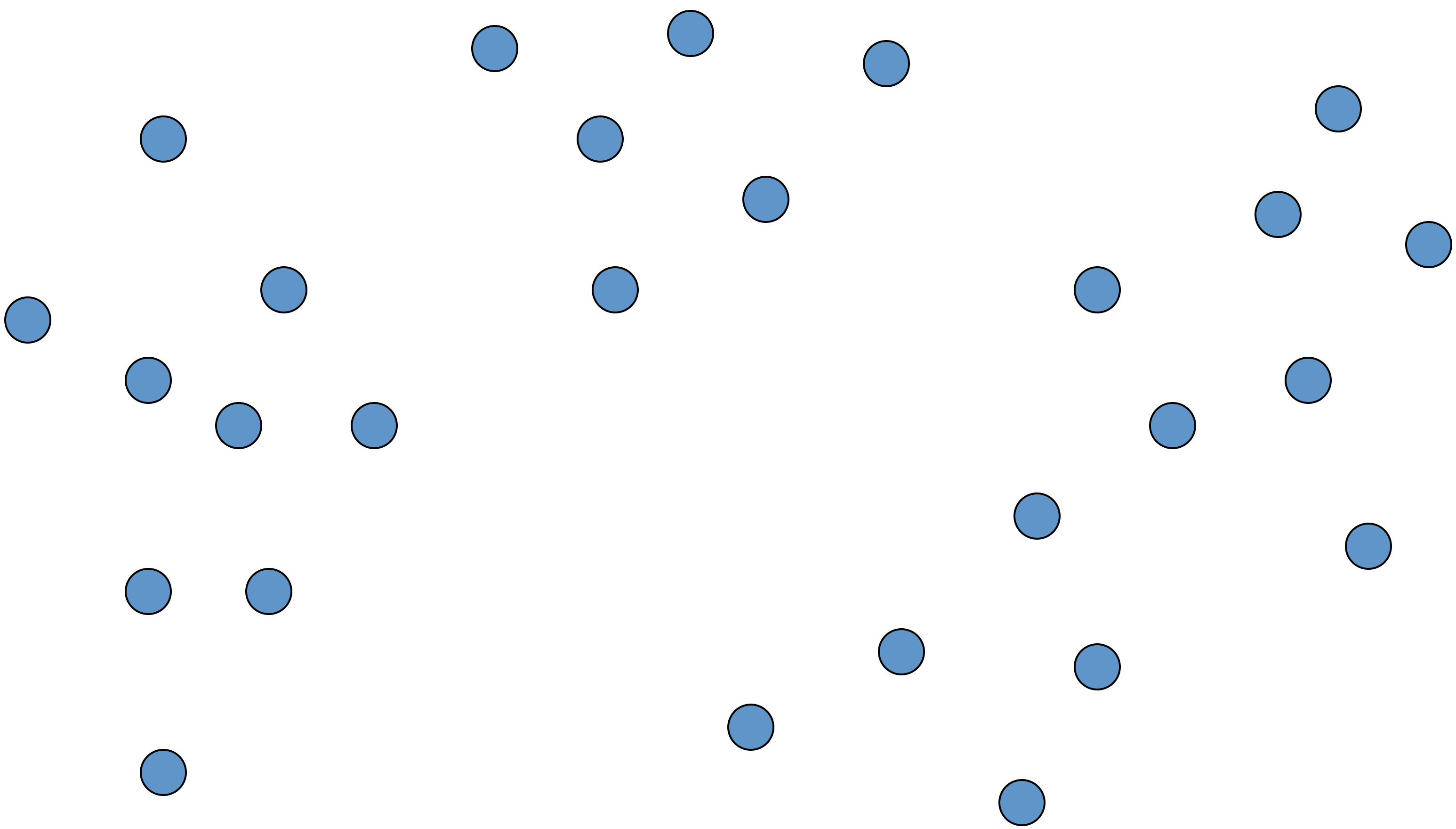
Divide into 2 clusters



Divide into 3 clusters



Divide into 4 clusters



Distance Clustering Algorithm

Let $C = \{\{v_1\}, \{v_2\}, . . ., \{v_n\}\}$; $T = \{ \}$

while $|C| > K$

Let $e = (u, v)$ with u in C_i and v in C_j be the minimum cost edge joining distinct sets in C

Replace C_i and C_j by $C_i \cup C_j$

K-clustering