Algorithms & Data Structures

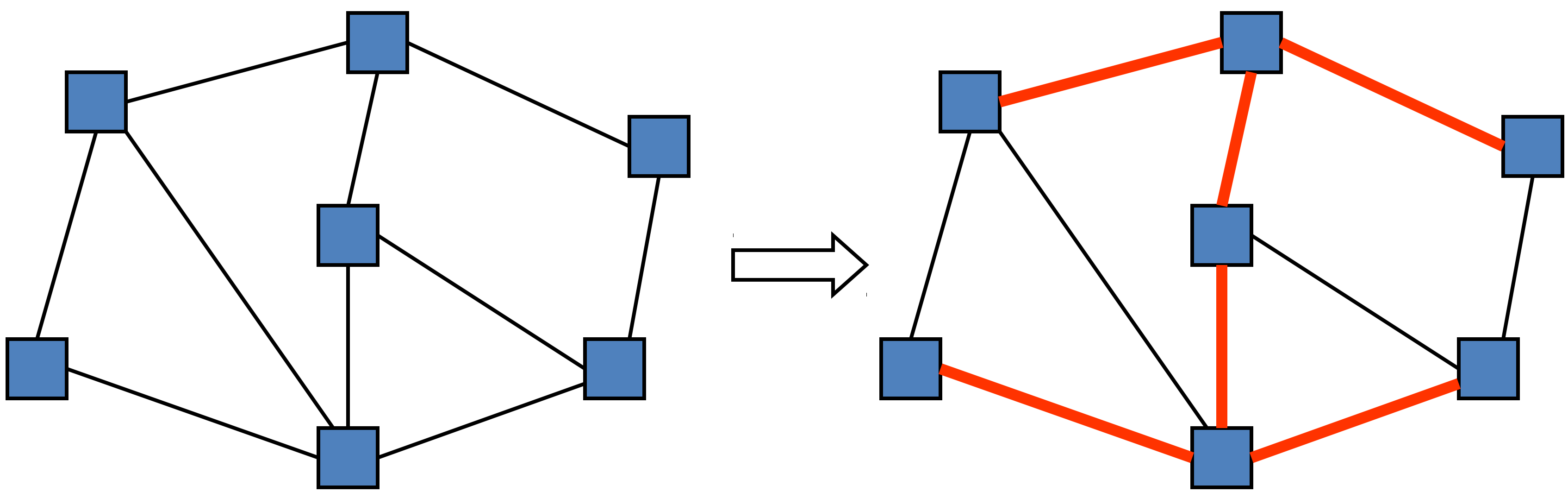
Lesson 13: Minimum Spanning Trees

Marc Gaetano

Edition 2017-2018

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
}
```

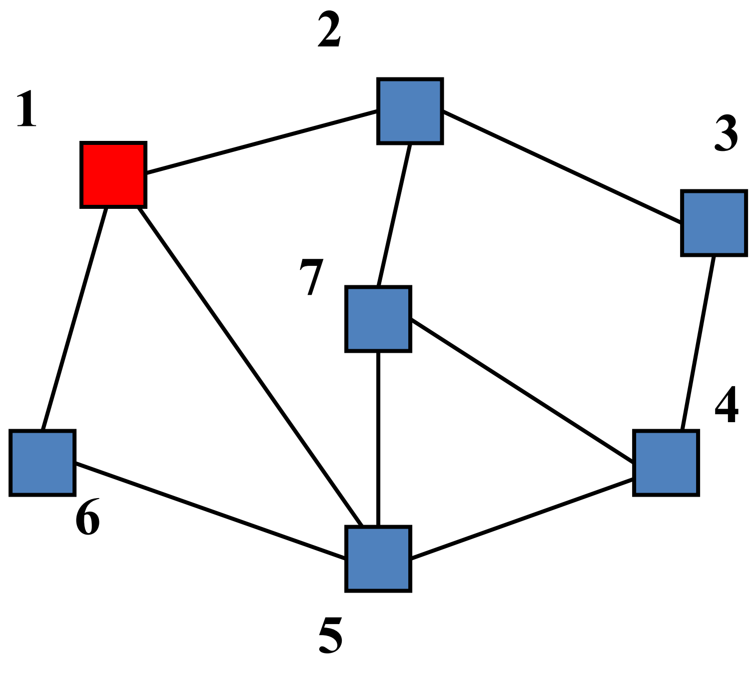
Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

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

Example

Stack

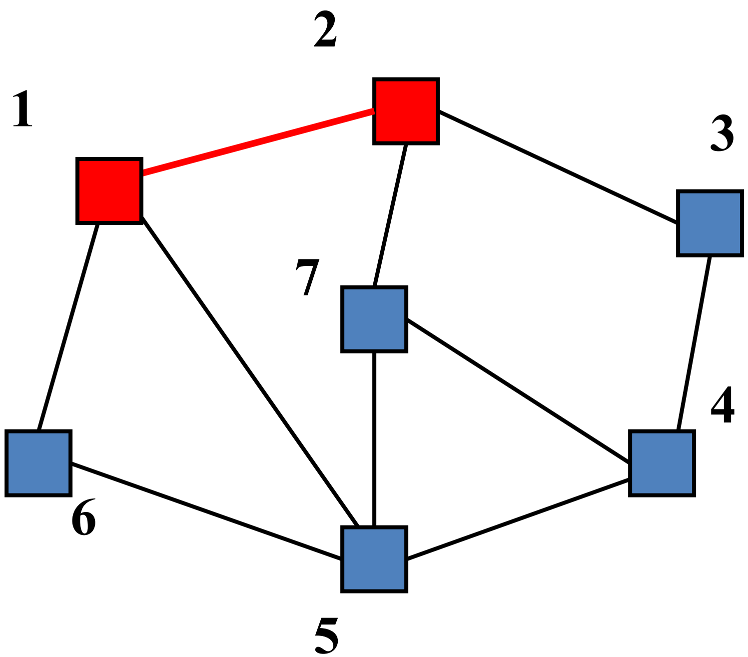
f(1)



Output:

Example

Stack (bottom)
f(1)
f(2)



Output: (1,2)

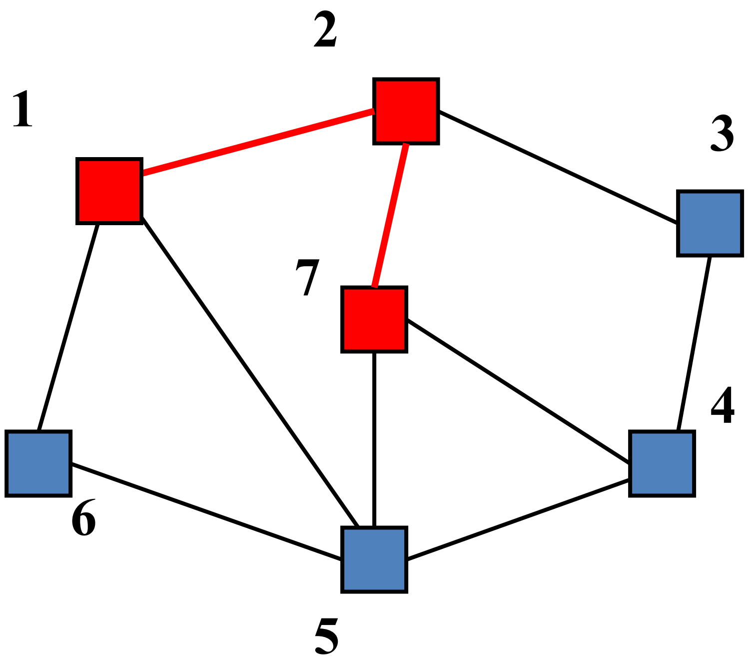
Example

Stack (bottom)

f(1)

f(2)

f(7)



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

Example

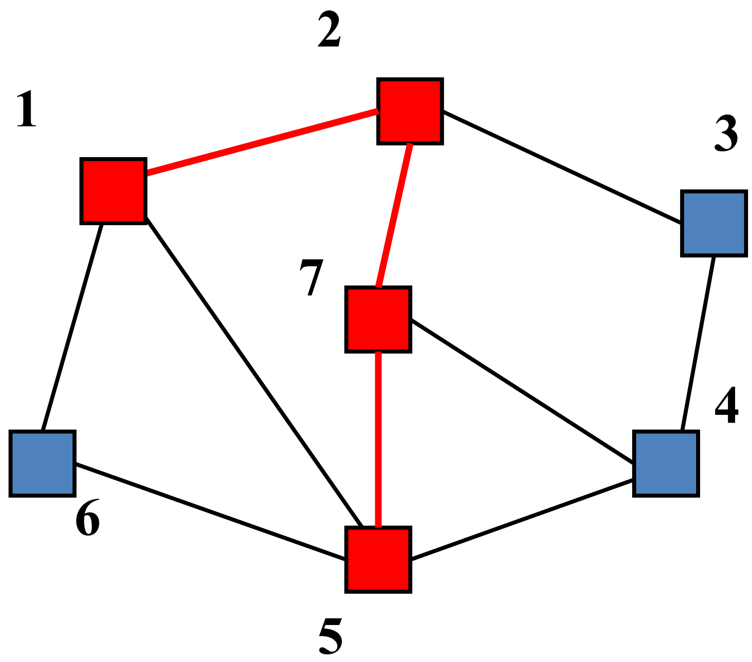
Stack (bottom)

f(1)

f(2)

f(7)

f(5)



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

Example

Stack (bottom)

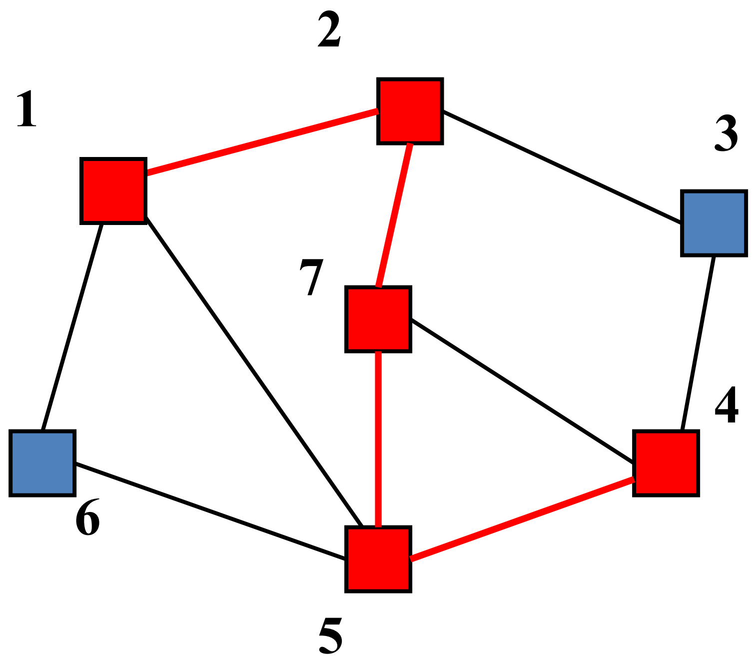
f(1)

f(2)

f(7)

f(5)

f(4)



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

Example

Stack (bottom)

f(1)

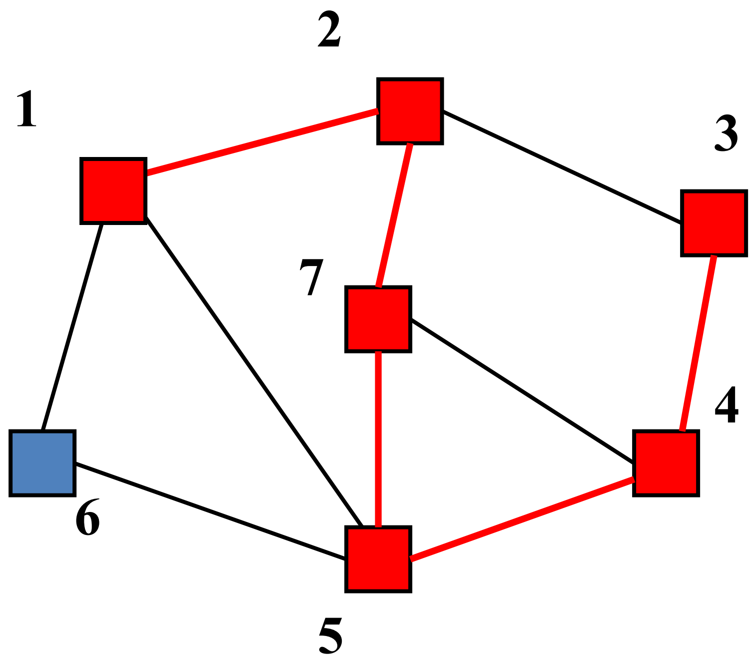
f(2)

f(7)

f(5)

f(4)

f(3)



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

Example

Stack (bottom)

f(1)

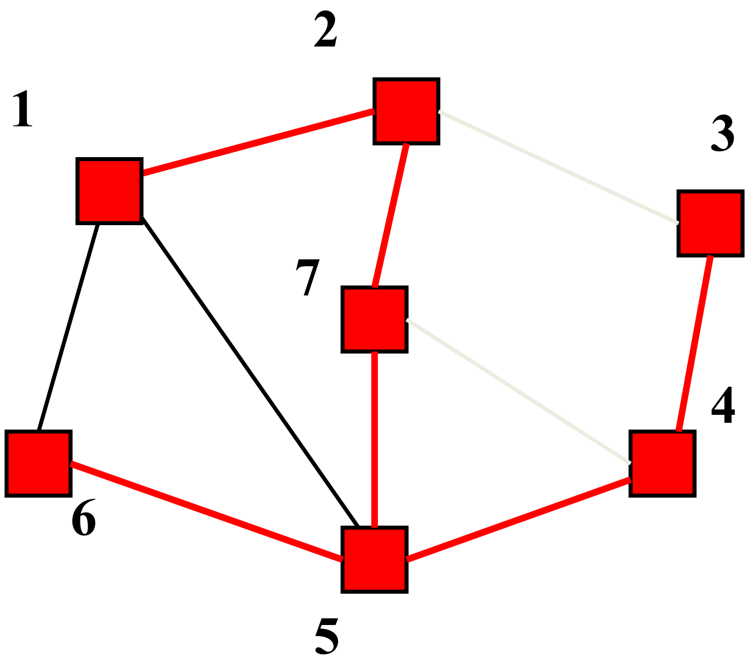
f(2)

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)

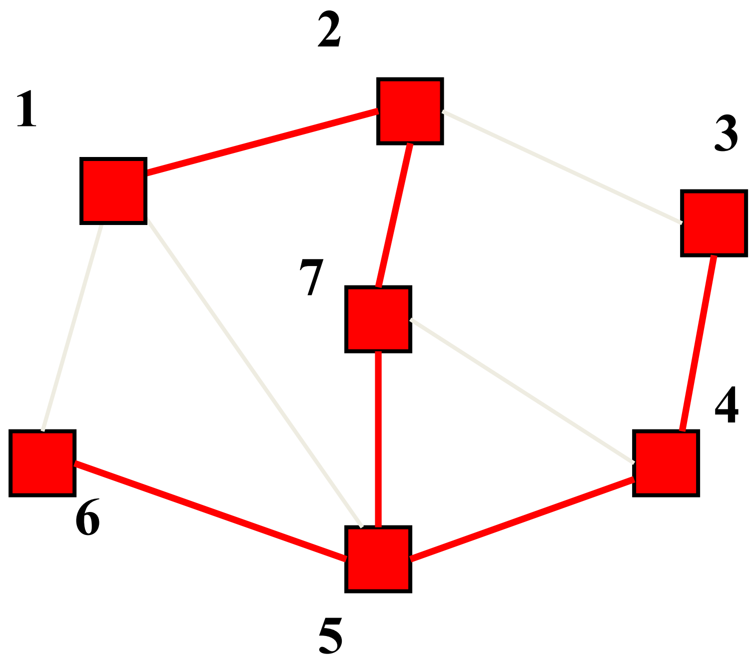
f(2)

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

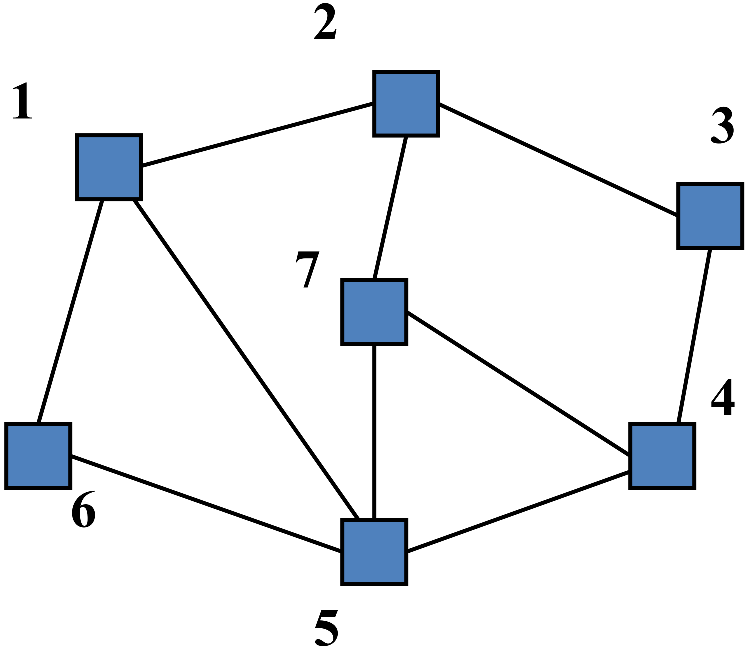
Efficiency:

- 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)

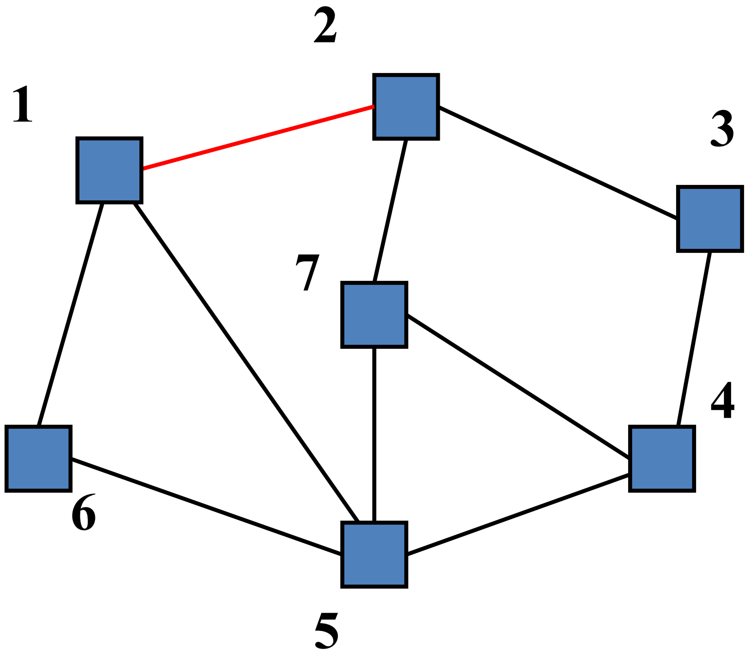


Output:

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)

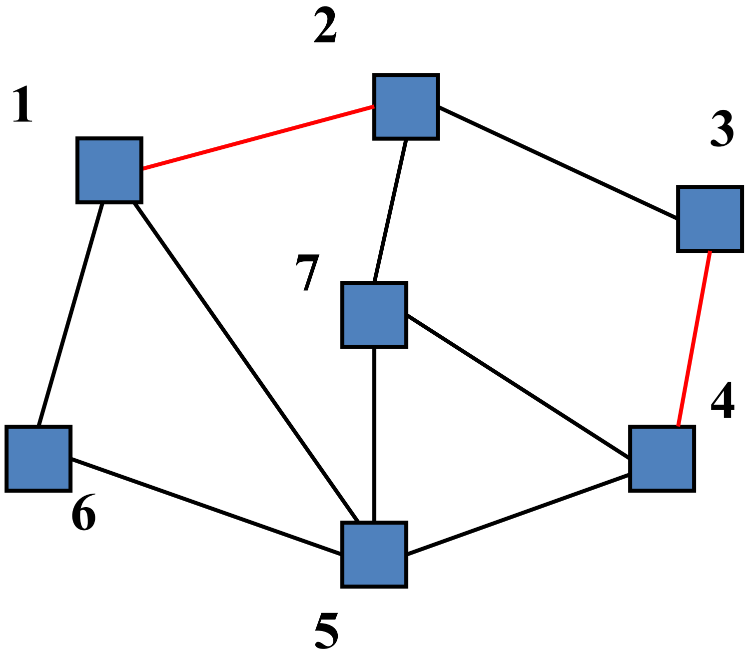


Output: (1,2)

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)

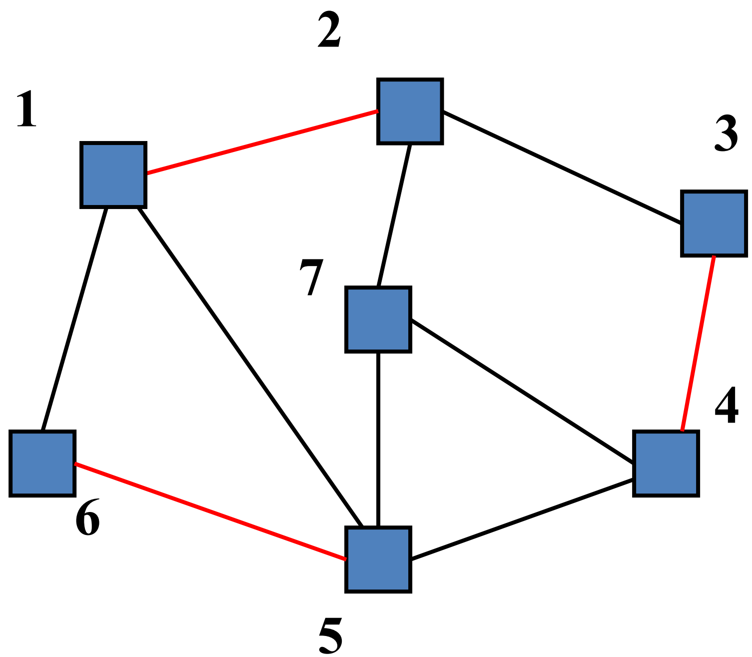


Output: (1,2), (3,4)

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)

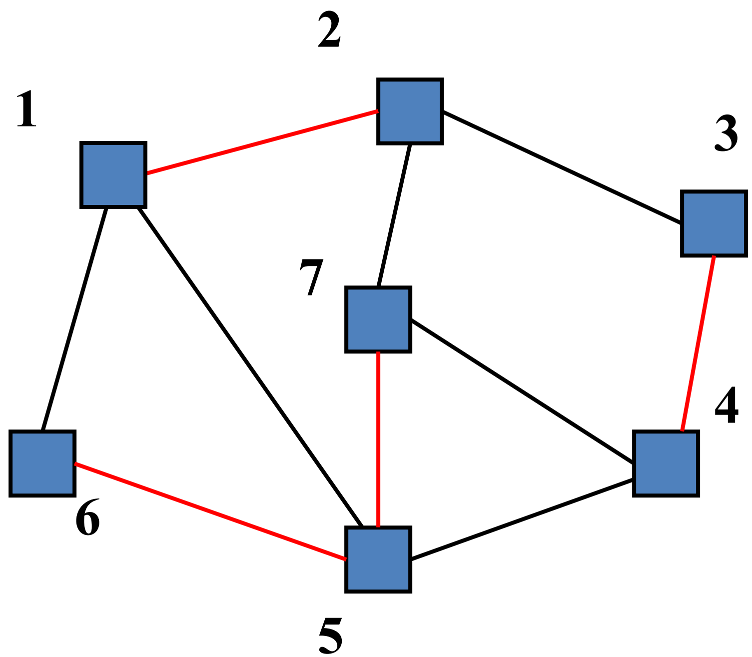


Output: (1,2), (3,4), (5,6),

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)

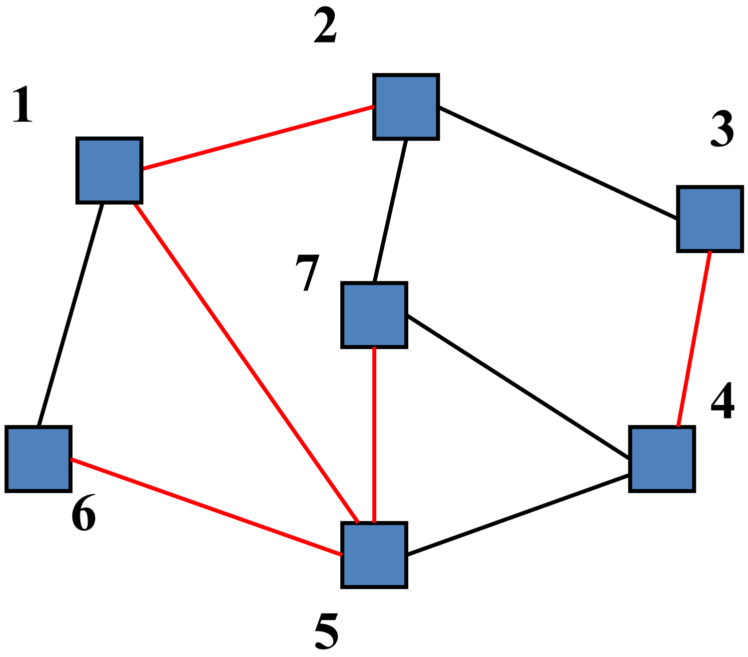


Output: (1,2), (3,4), (5,6), (5,7)

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)

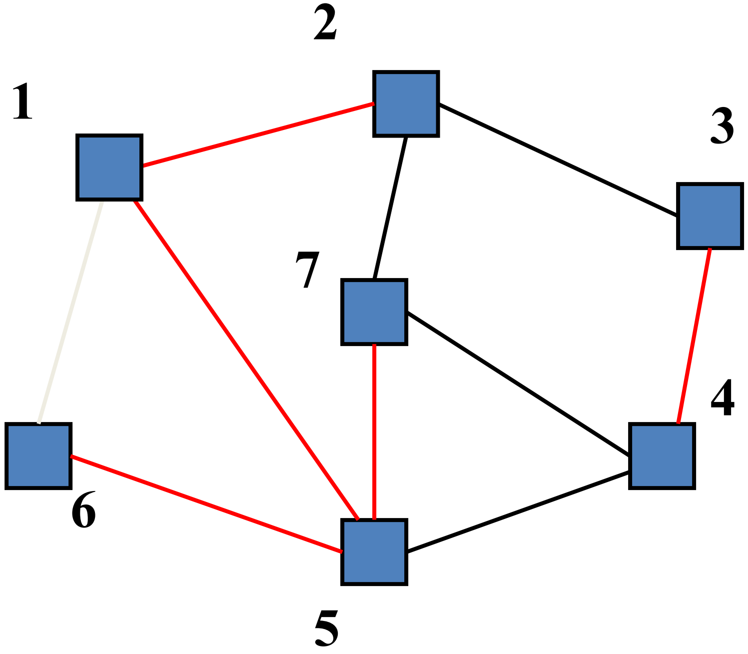


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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)

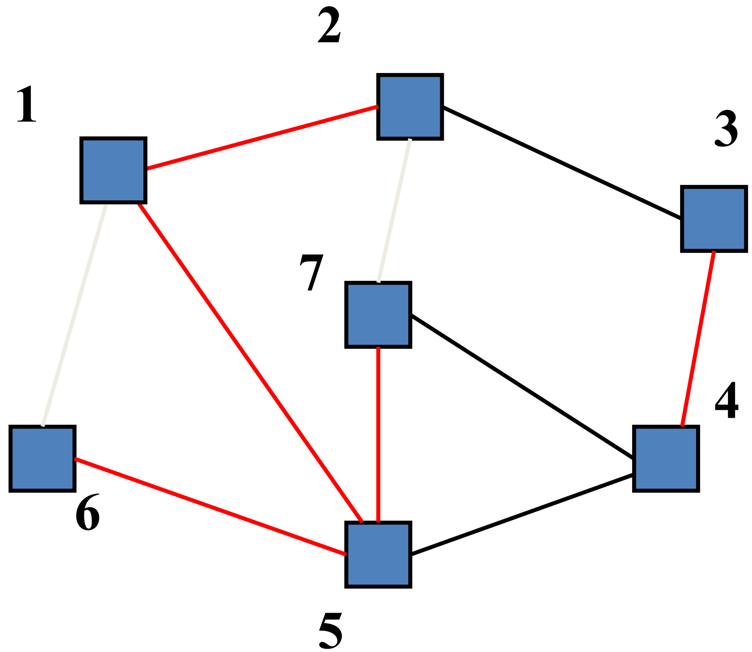


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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)

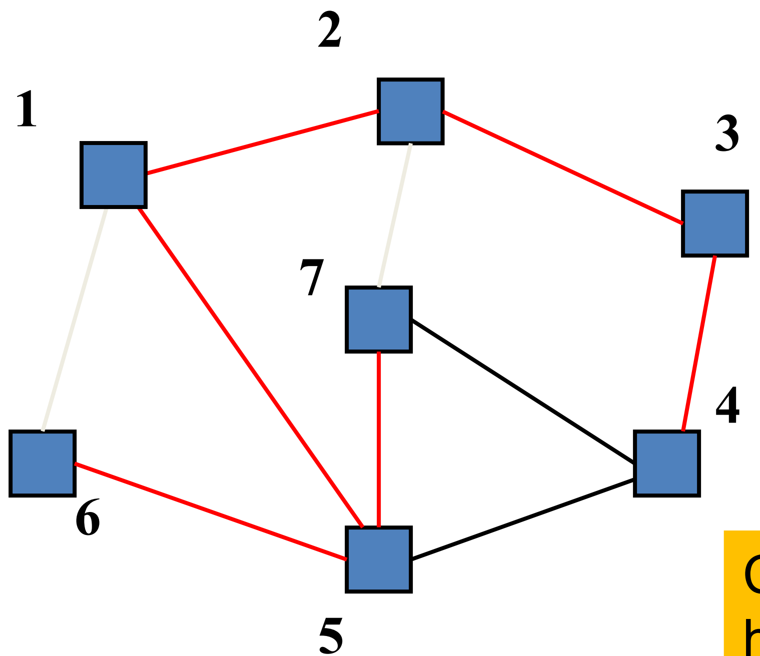


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

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 $O(|\mathbf{V}|)$ because we may need to traverse all edges already in the output

- So overall algorithm would be $O(|\mathbf{V}||\mathbf{E}|)$

- But there is a faster way: 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
  - Explain in next lesson

Summary So Far

The spanning-tree problem

- Add nodes to partial tree approach is $O(|\mathbf{E}|)$
- Add acyclic edges approach is *almost* $O(|\mathbf{E}|)$
  - 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 $O(|\mathbf{E}| \log |\mathbf{V}|)$

Getting to the Point

Algorithm #1

Prim's Algorithm for Minimum Spanning Tree

is

Exactly our 1st approach to spanning tree

but process crossing edges in cost order

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

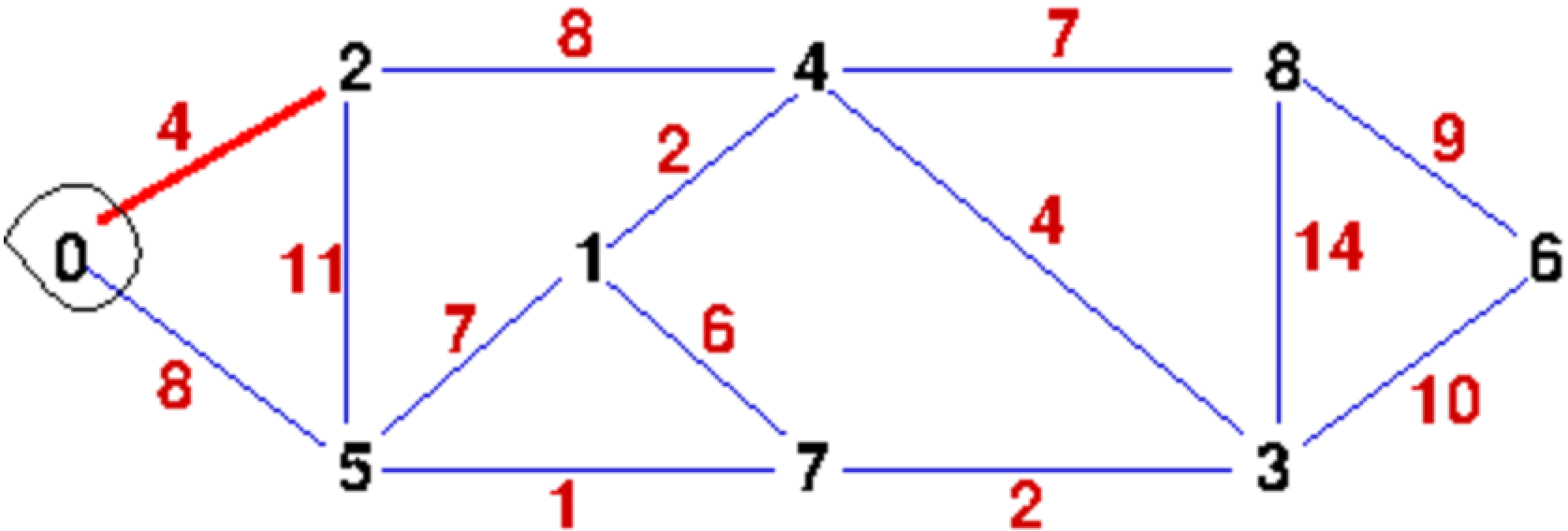
Idea:

- grow the MST starting with no edge
- mark vertices connected to the MST
- add an edge to the MST by picking the smallest weight edge among the *crossing* edges (*crossing edge*: edge with a vertex marked and a vertex not marked

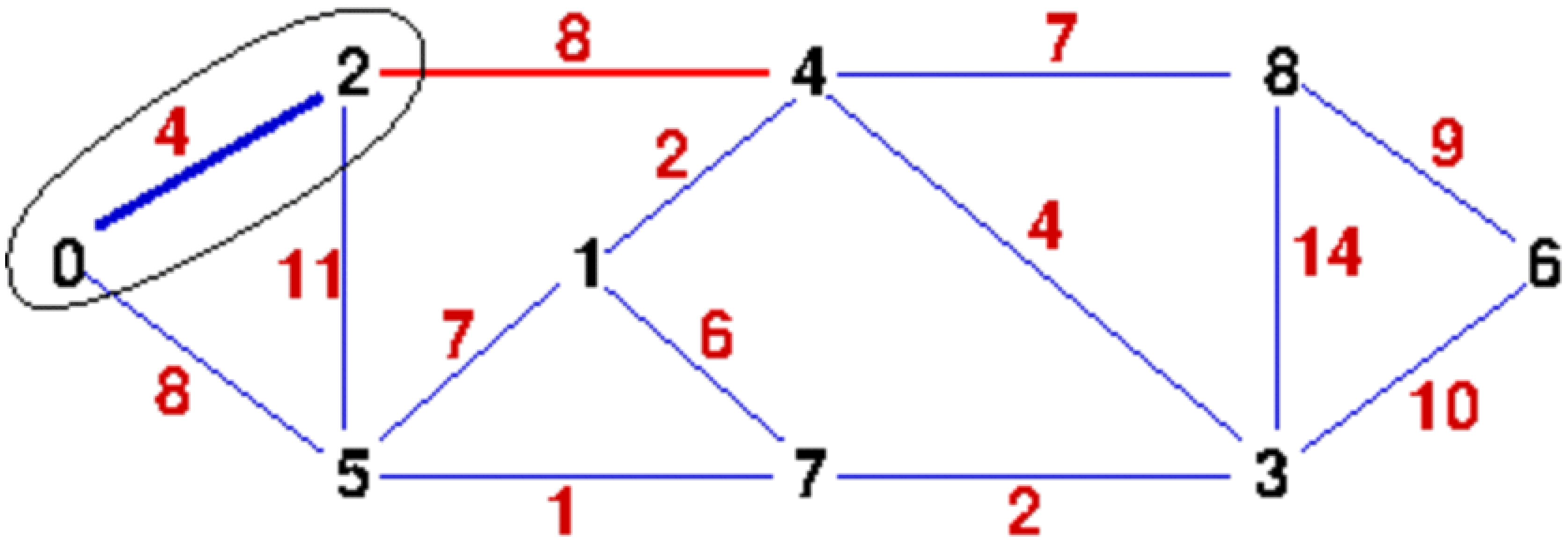
Algorithm pseudo-code

1. Set all vertices as *unmarked*
2. Set `S = { }`, the set of edges of the MST
3. Set `C = { }`, the set of crossing edge (`(u,v)` is a crossing edge iff u is marked and `v` is unmarked)
4. Choose any node `u`
   a) Mark u
   b) For each edge `e = (u,v)`, () add `e` to `C`
5. While there are unmarked vertices in the graph
   a) Select the crossing edge `e = (a,b)` with lowest cost
   b) Add `e` to S
   c) Mark b
   d) For each edge `e' = (b,c)` (`c` **not** marked), label e' "crossing"

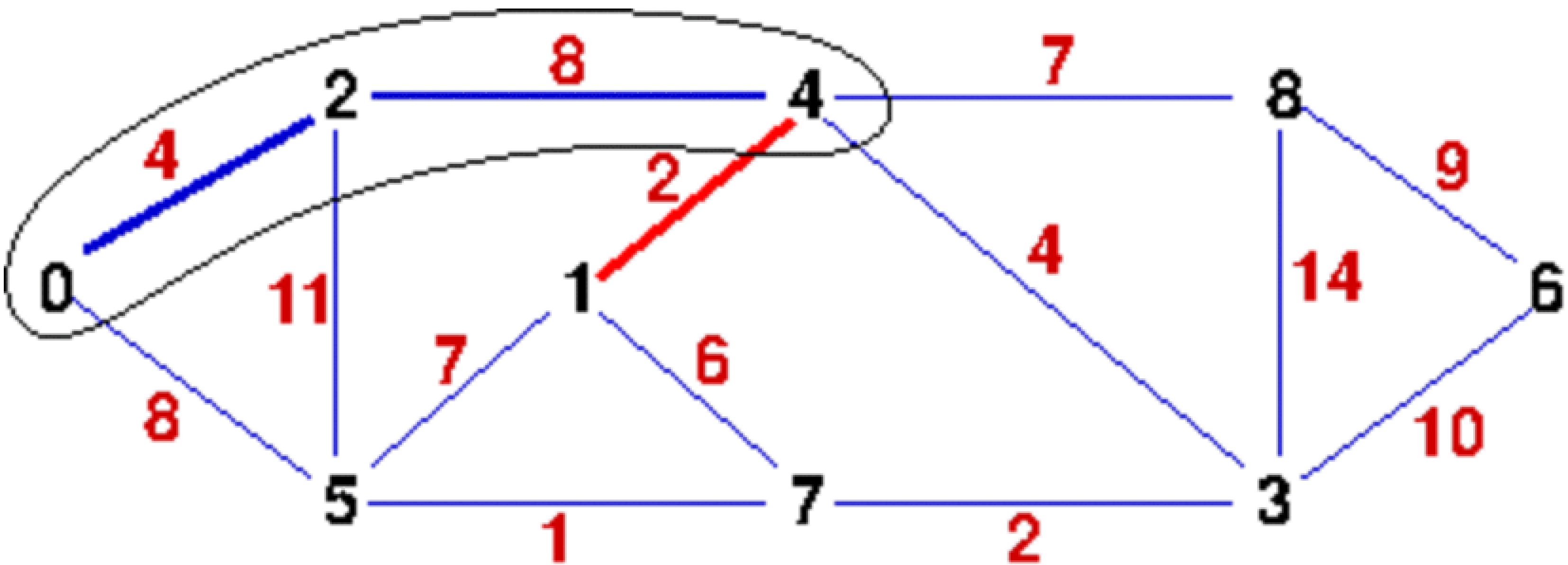
Prim example



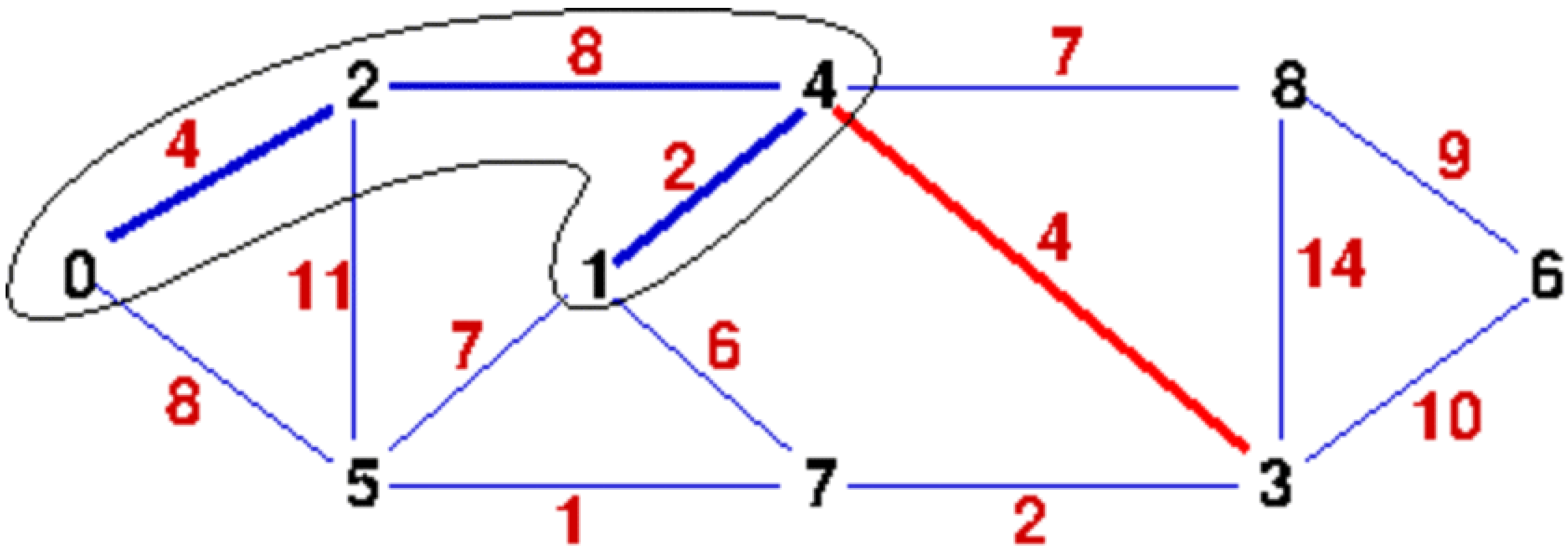
Prim example



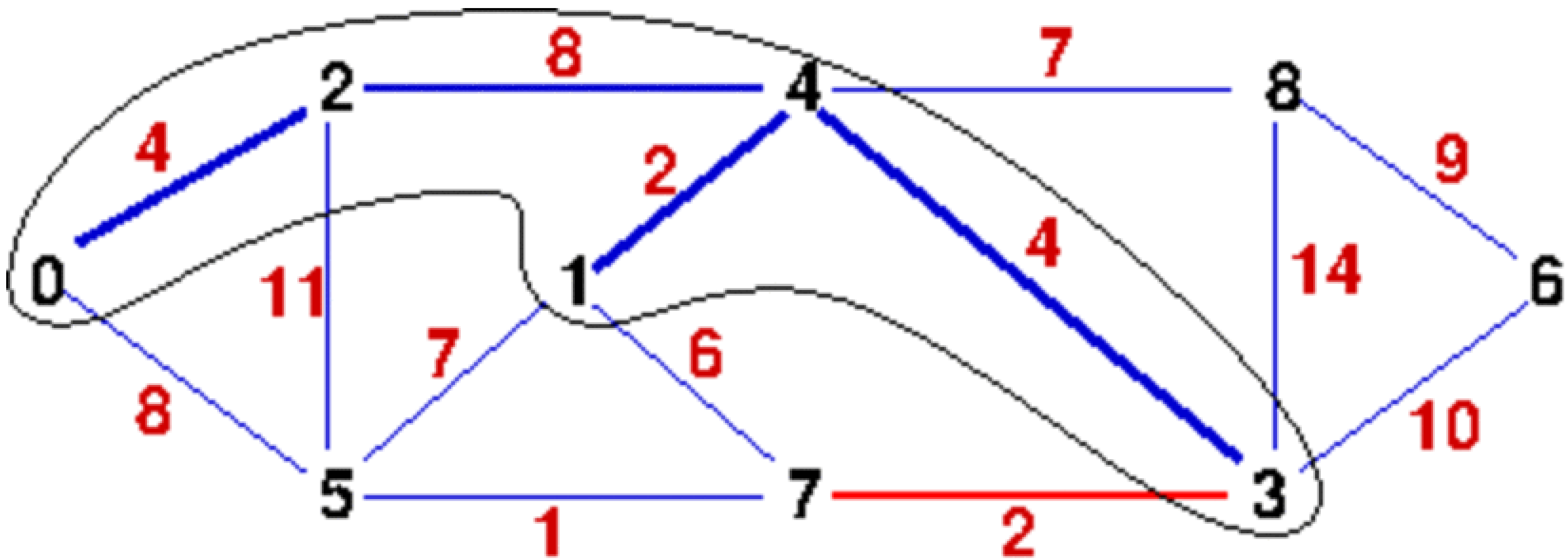
Prim example



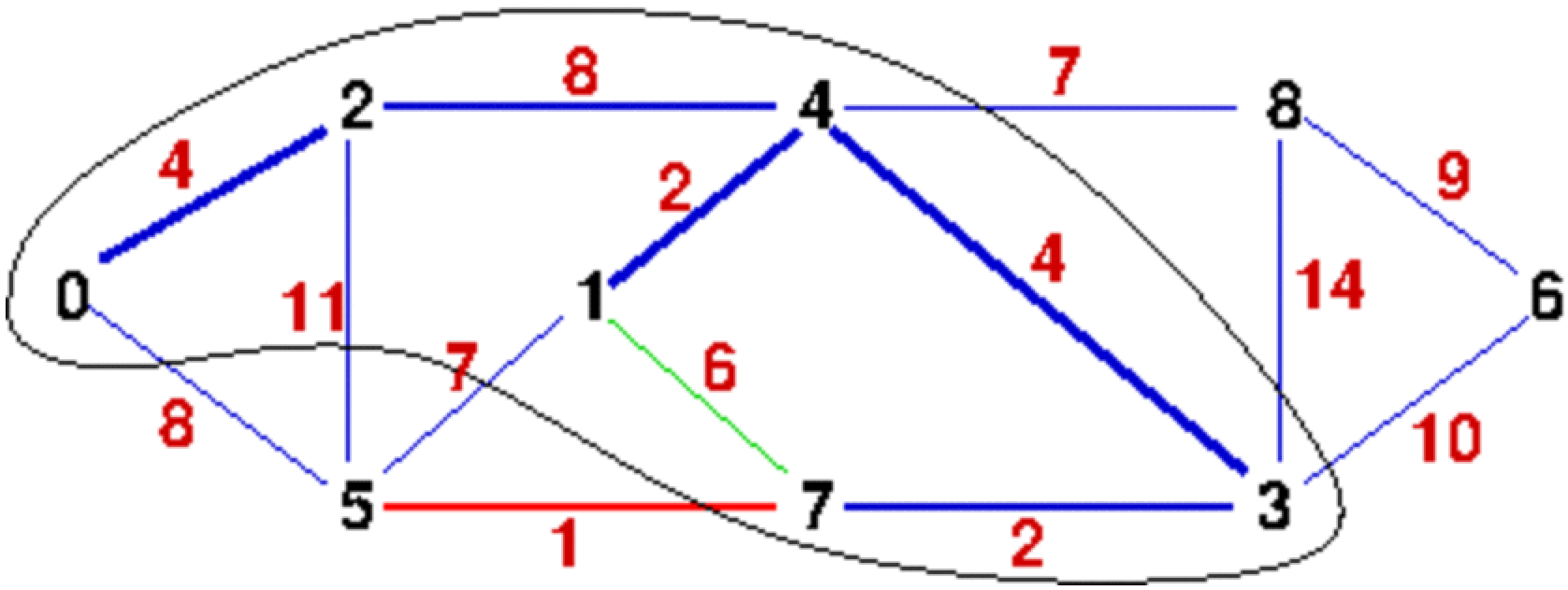
Prim example



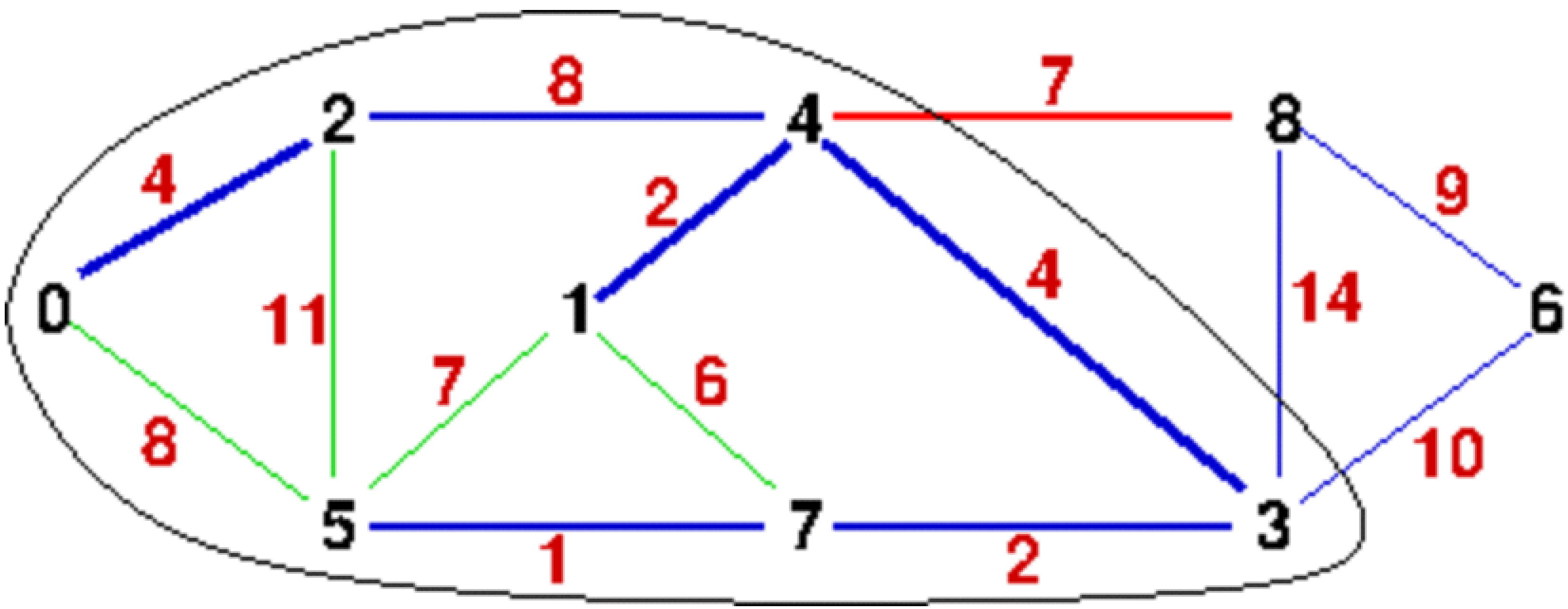
Prim example



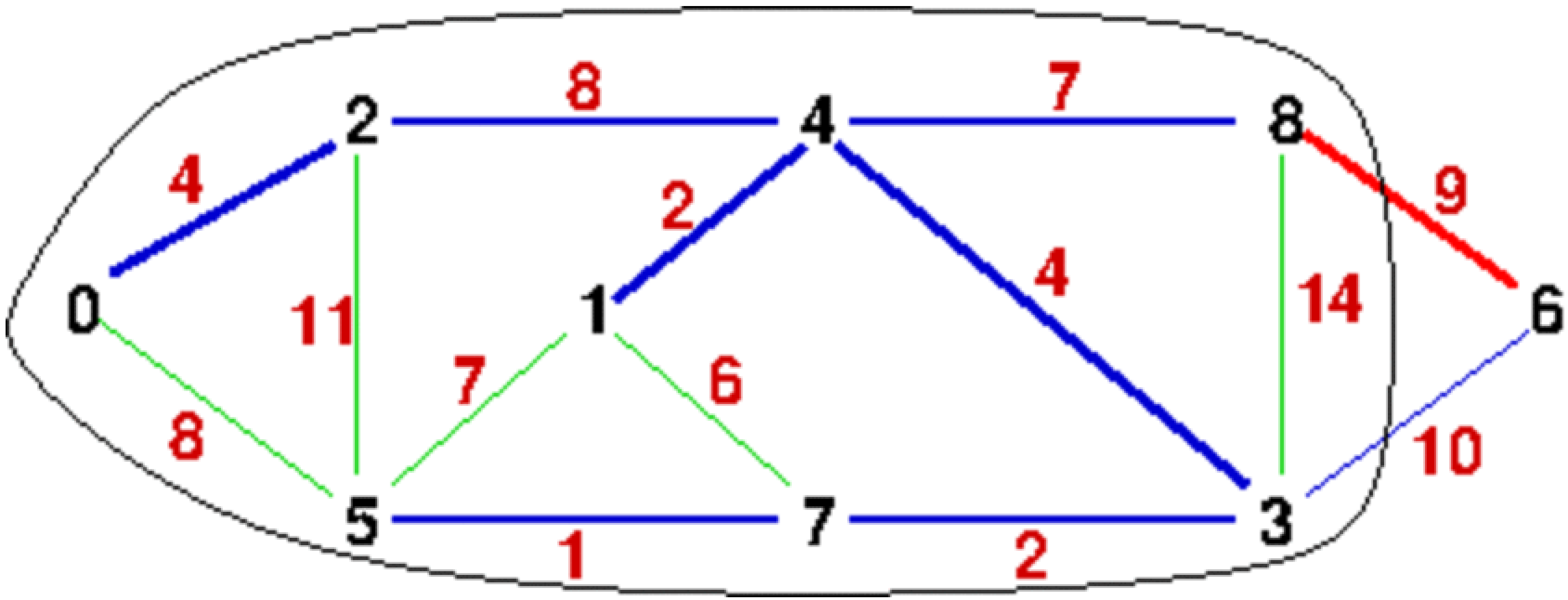
Prim example



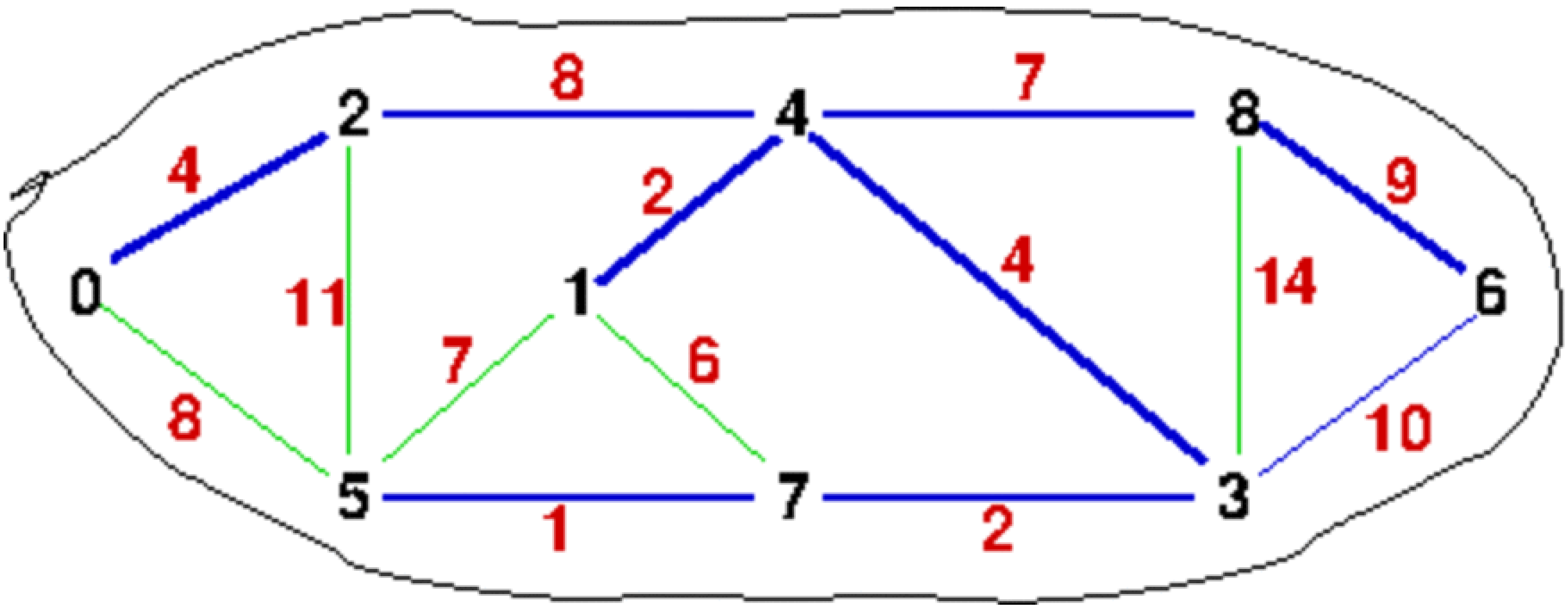
Prim example



Prim example



Prim example



Prim's analysis

- Correctness
  - Invariant: `S` is a MST of the subgraph induced by the ***marked*** vertices
  - Variant: `|U|` decrease down to `0` (`U` is the set of *unmarked* vertices)

- Run-time complexity
  - Sort the set of edges (*O*(|E| log |E|)) and pick each edge in increasing order of weight (*O*(|E|)) = *O*(|E| log |E|)
  - Somehow (non asymptotically) better: *O*(|E|log|E|) using a heap to store the crossing edges

Kruskal's algorithm idea

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 increasing weight

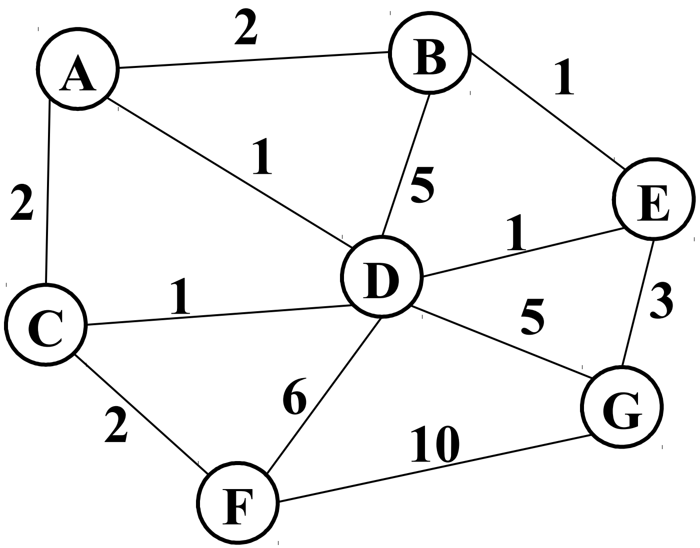
So:

- Sort edges: $O(|\mathbf{E}|\log|\mathbf{E}|)$
- Iterate through edges: $O(|\mathbf{E}|)$
- Use union-find for cycle detection: $O(|\mathbf{E}|)$ (next lesson)

Kruskal's pseudocode

1. Set `S = { }`, the set of edges of the MST
2. Sort edges by weight
3. Put each vertex in its own set
4. While the number of sets > 1
   - pick next smallest edge `e = (u,v)`
   - if `u` and `v` are in in different sets S_1 and S_2
     - add `e` to `S`
     - merge S_1 and S_2

Kruskal's 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)

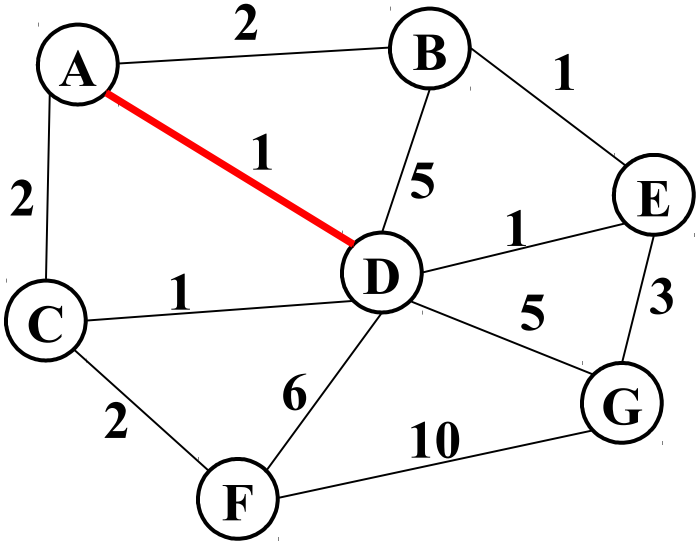
6: (D,F)

10: (F,G)

MST: { }

Sets: {A} {B} {C} {D} {E} {F} {G}

Kruskal's 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)

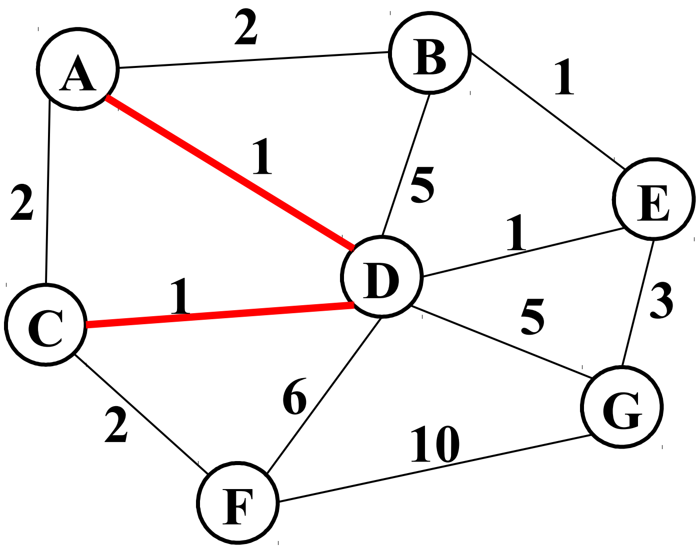
6: (D,F)

10: (F,G)

MST: { (A,D) }

Sets: {A, D} {B} {C} {E} {F} {G}

Kruskal's 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)

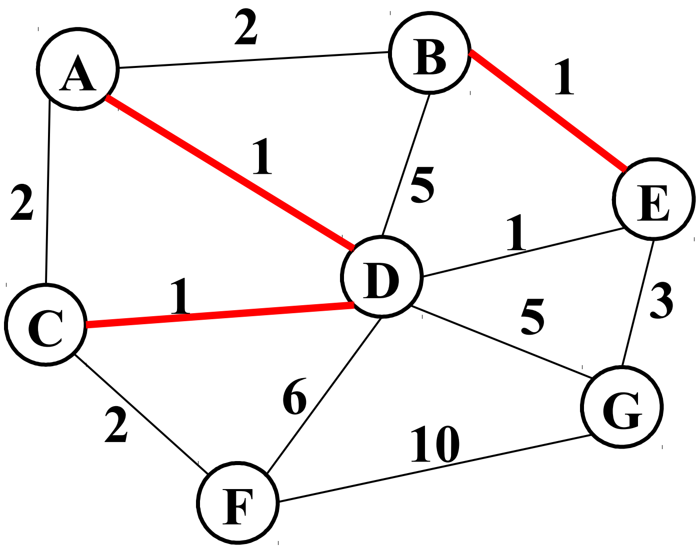
6: (D,F)

10: (F,G)

MST: { (A,D), (C,D) }

Sets: {A, D, C} {B} {E} {F} {G}

Kruskal's 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)

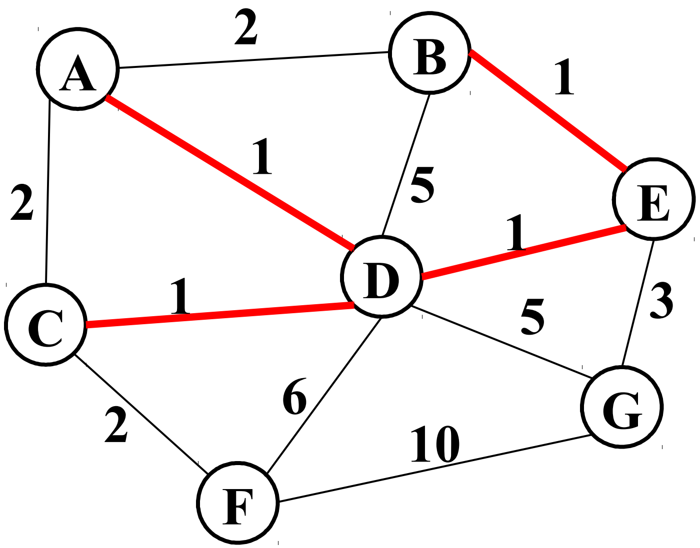
6: (D,F)

10: (F,G)

MST: { (A,D), (C,D), (B,E) }

Sets: {A, D, C} {B, E} {F} {G}

Kruskal's 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)

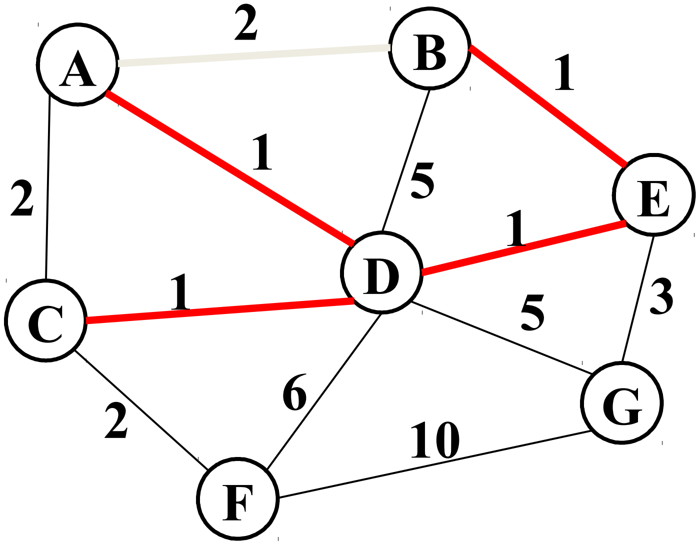
6: (D,F)

10: (F,G)

MST: { (A,D), (C,D), (B,E), (D,E) }

Sets: {A, D, C, B, E} {F} {G}

Kruskal's 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)

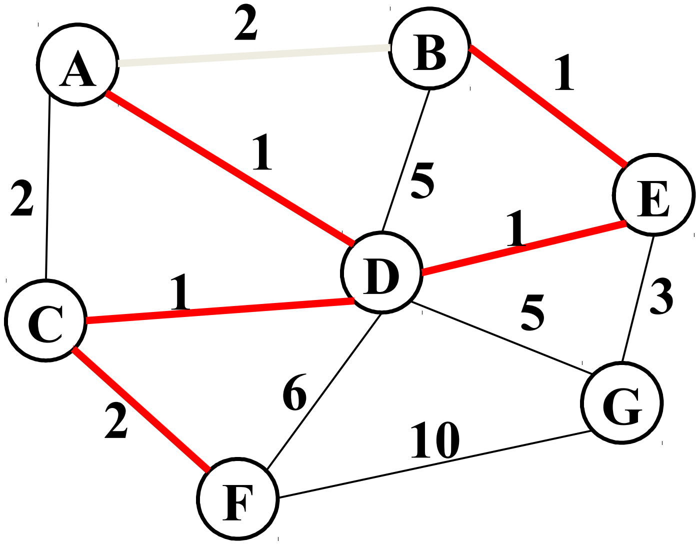
6: (D,F)

10: (F,G)

MST: { (A,D), (C,D), (B,E), (D,E) }

Sets: {A, D, C, B, E} {F} {G}

Kruskal's 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)

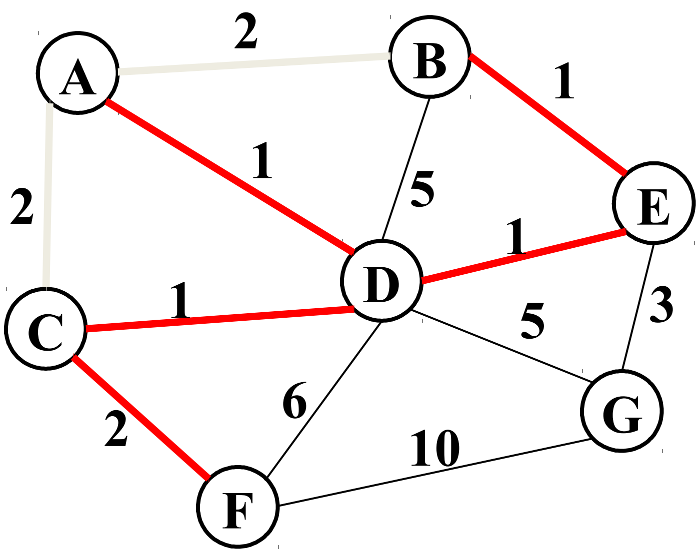
6: (D,F)

10: (F,G)

MST: { (A,D), (C,D), (B,E), (D,E), (C,F) }

Sets: {A, D, C, B, E, F} {G}

Kruskal's 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)

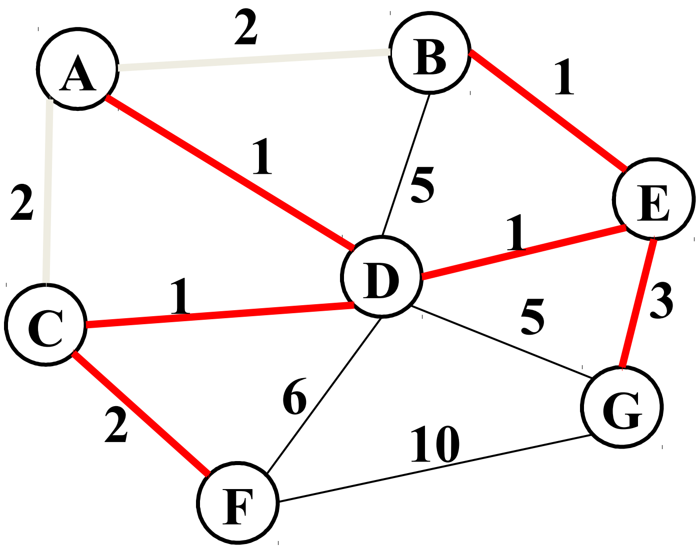
6: (D,F)

10: (F,G)

MST: { (A,D), (C,D), (B,E), (D,E), (C,F) }

Sets: {A, D, C, B, E, F} {G}

Kruskal's 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)

MST: { (A,D), (C,D), (B,E), (D,E), (C,F), (E,G) }

Sets: {A, D, C, B, E, F, G}

Kruskal's analysis

Correctness:

- invariant: S is a MST of the sub-graph induced by the set of sets of vertices
- variant: either the number of sets or the number of non chosen edges is decreasing

Runtime complexity:

- Floyd's algorithm to build min-heap with edges *O*(**|E|**)
- Iterate through edges using **`deleteMin`** to get next edge: *O*(**|E|** **`log`** **|E|**)
- Use union-find to manage the set of sets of vertices: *O*(**|E|**)
- often stop long before considering all edges