EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Graphs

in the "real world"

- Networks are graphs
  - Information: WWW, citation, ...
  - Social: co-actor, dating, messenger, communities, ...
  - Technological: Internet, power grids, airline routes, …
  - Biological: Neural networks, food web, blood vessels, …
- Object hierarchies are graphs
- Circuit layouts are graphs
- Computer programs are graphs

Graph Traversals

- Traversals of graphs are also called *searches*
- We can use either breadth-first or depth-first traversals
  - Breadth-first requires a queue
  - Depth-first requires a stack
- We each case, we will have to track which vertices have been visited requiring $\Theta(|V|)$ memory
- The time complexity cannot be better than and should not be worse than $\Theta(|V| + |E|)$
  - Connected graphs simplify this to $\Theta(|E|)$
  - Worst case: $\Theta(|V|^2)$

Applications

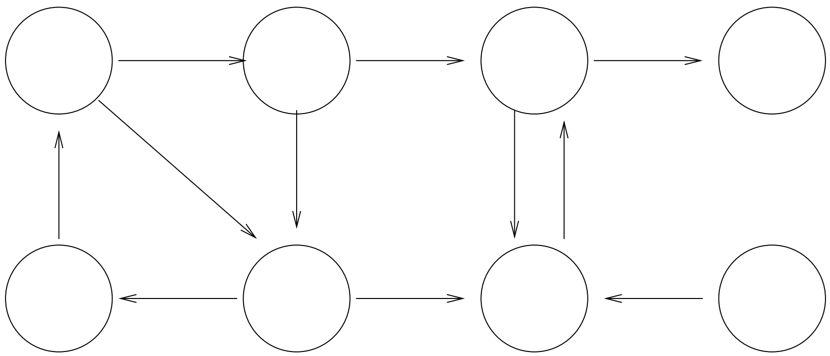
Applications of tree traversals include:

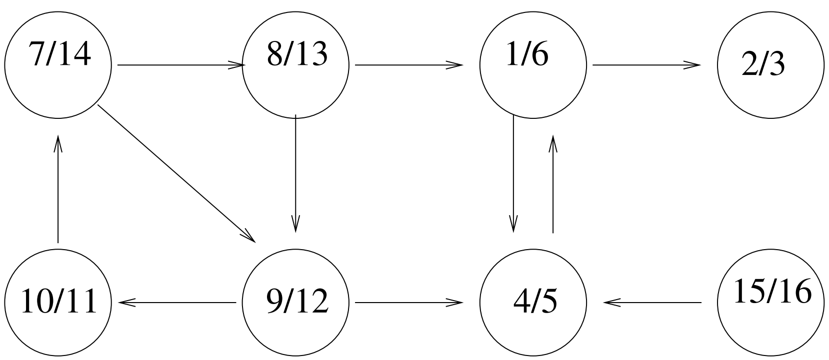
- Determining connectedness and finding connected sub-graphs
- Determining the path length from one vertex to all others
- Testing if a graph is bipartite
- Branch and bound search
- Topological Sort
- …

Strongly-Connected Components

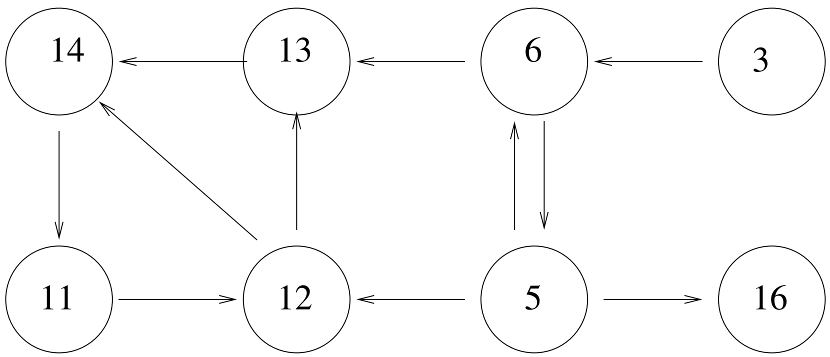
- Kosaraju's Algorithm
  - Perform DFS on graph G = (V, E),
    - Number vertices according to their finishing time in DFS of G
  - Perform DFS on Gr = (V,Er), where Er are reverse of edges in E, selecting nodes in decreasing order of finishing time in previous DFS
  - Strongly connected components = reachable trees obtained in last DFS

Example

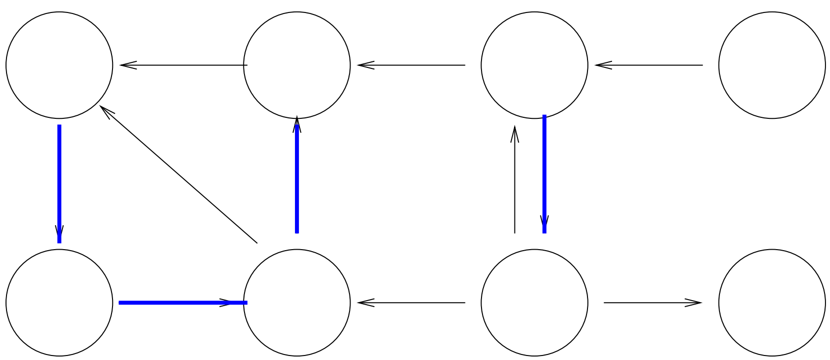




Example



Reverse graph with distance labels



Reverse graph reachable trees

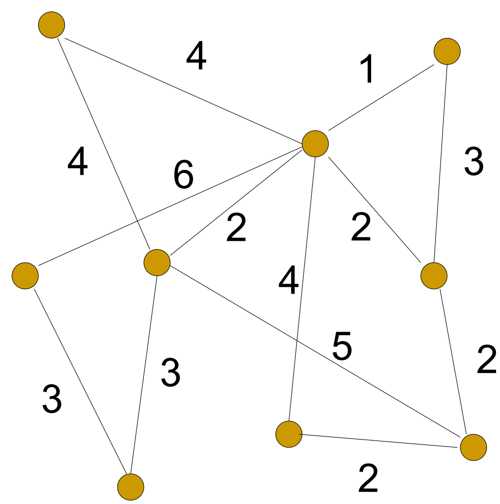
Strongly-Connected Components

- Correctness
  - If v and w are in a strongly-connected component
  - Then there is a path from v to w and a path from w to v
  - Therefore, there will also be a path between v and w in G and Gr

- Running time
  - Two executions of DFS
  - $O(|E|+|V|)$

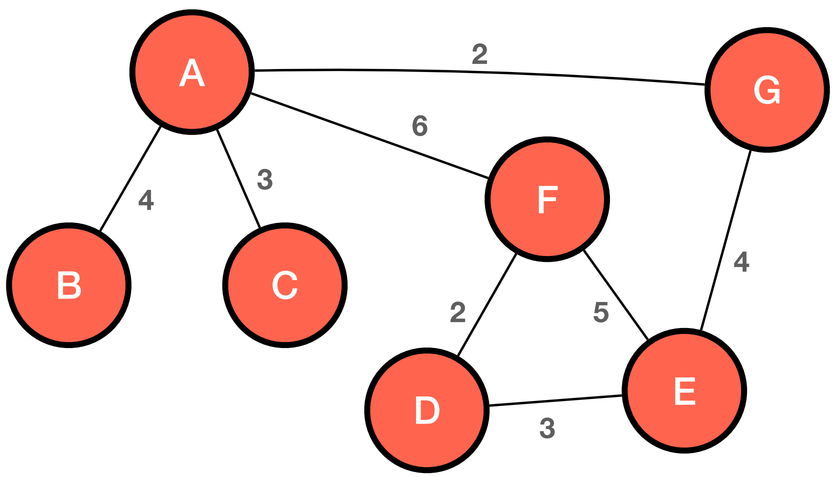
Weighted Graphs

- A weighted graph $G = (V, E)$ is a graph along with a weight function $w : E \to \Re$
- Weighted graphs can be directed or undirected



Spanning Trees

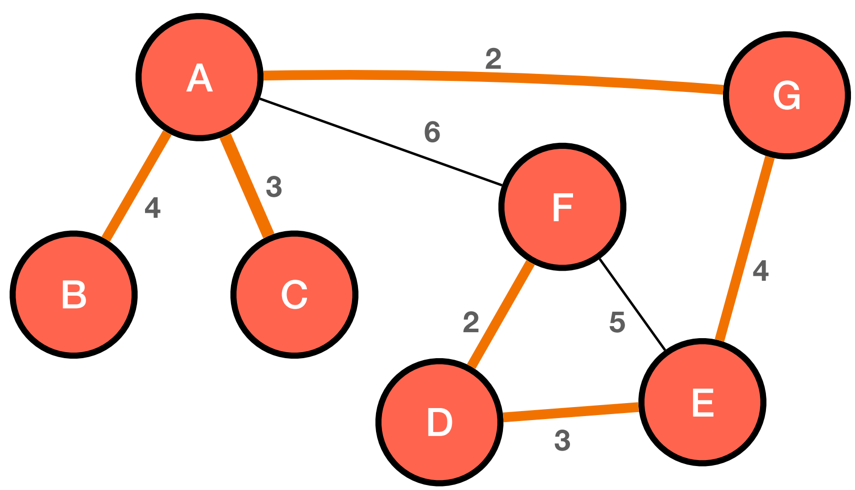
- A **spanning tree** of an undirected graph is
  - edge subset forming a tree that spans every vertex
  - #V - 1 edges



Minimum Spanning Trees

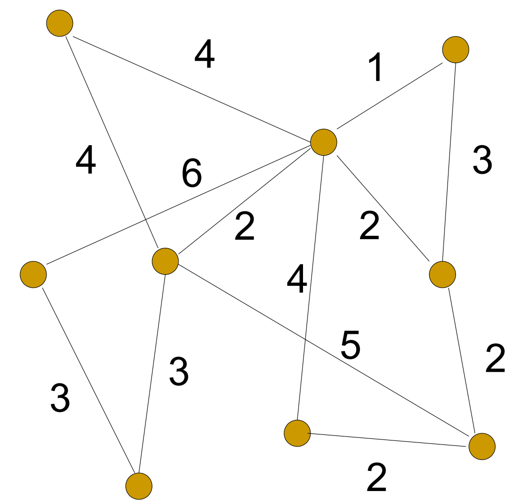
- A **minimum spanning tree** (MST) of an undirected weighted graph (V, E) with weights $w(\cdot)$:

Connected subgraph (V, E') which is a tree and for which $\sum_{\{i,j\} \in E'} w(i, j)$ is minimized



MST Problem

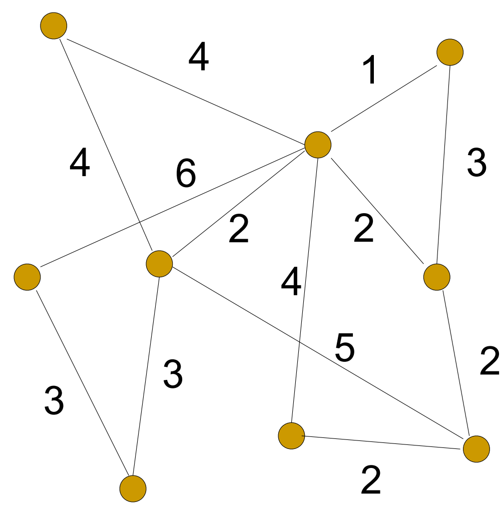
- Given a weighted graph G, we want a subgraph $G' = (V, E'),\ E' \subseteq E$, such that
  - all vertices are connected on G′
  - Subgraph $G' = (V, E'),\ E' \subseteq E$
  - total weight $\sum_{(x,y)\in E'} w(x,y)$ is minimized



- Spanning tree: a tree containing all vertices in G
- Question: Find a spanning tree with minimum weight.
  - The problem is thus called Minimum Spanning Tree (MST)

MST: Problem and Motivation

- Suppose we have n computers, connected by wires as given in the graph
- Each wire has a renting cost
- We want to select some wires, such that all computers are connected (i.e. every two can communicate)
- Algorithmic question: How to select a subset of wires with the minimum renting cost?
- Answer to this graph?



Applications

- Networks
  - electric, computer, water, transportation
- Computer vision
  - Facial recognition
  - Handwriting recognition
  - Image segmentation
- Low-density parity check codes (LDPC)

Minimum Spanning Tree Algorithms

- Kruskal's algorithm (1956)

ON THE SHORTEST SPANNING SUBTREE OF A GRAPH AND THE TRAVELING SALESMAN PROBLEM

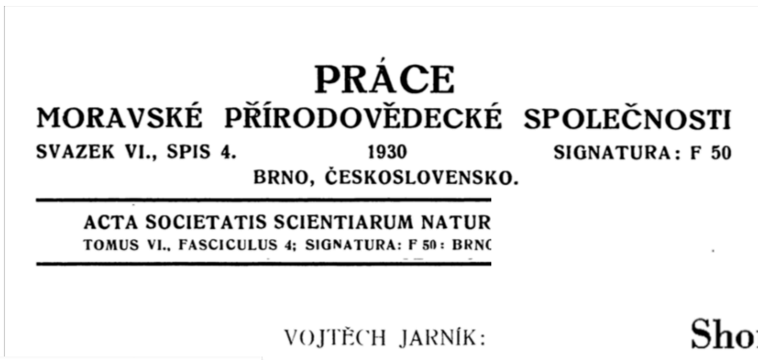
JOSEPH B. KRUSKAL, JR.

Several years ago a typewritten translation (of obscure origin) of [1] raised some interest. This paper is devoted to the following theorem: If a (finite) connected graph has a positive real number attached to each edge (the *length* of the edge), and if these lengths are all distinct, then among the spanning[1] trees (German: Gerüst) of the graph there is only one, the sum of whose edges is a minimum; that is, the shortest spanning tree of the graph is unique. (Actually in [1] this theorem is stated and proved in terms of the "matrix of lengths" of the graph, that is, the matrix $\|a_{ij}\|$ where a_{ij} is the length of the edge connecting vertices i and j. Of course, it is assumed that $a_{ij}=a_{ji}$ and that $a_{ii}=0$ for all i and j.)

The proof in [1] is based on a not unreasonable method of constructing a spanning subtree of minimum length. It is in this construction that the interest largely lies, for it is a solution to a problem (Problem 1 below) which on the surface is closely related to one version (Problem 2 below) of the well-known traveling salesman problem.

Minimum Spanning Tree Algorithms

- Prim-Jarnik Algorithm



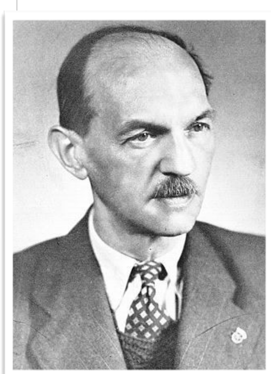
PRÁCE
MORAVSKÉ PŘÍRODOVĚDECKÉ SPOLEČNOSTI
SVAZEK VI., SPIS 4. 1930 SIGNATURA: F 50
BRNO, ČESKOSLOVENSKO.

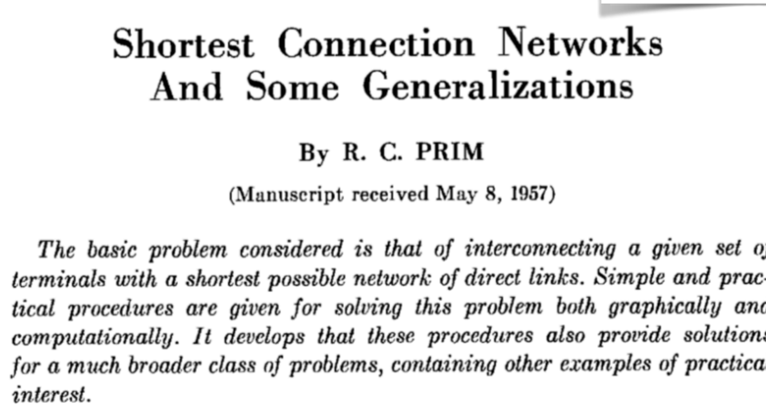
ACTA SOCIETATIS SCIENTIARUM NATUR
TOMUS VI., FASCICULUS 4; SIGNATURA: F 50: BRNC

VOJTĚCH JARNÍK:

problému mini

opisu panu O. BORŮVI





Shortest Connection Networks And Some Generalizations

By R. C. PRIM

(Manuscript received May 8, 1957)

The basic problem considered is that of interconnecting a given set of terminals with a shortest possible network of direct links. Simple and practical procedures are given for solving this problem both graphically and computationally. It develops that these procedures also provide solutions for a much broader class of problems, containing other examples of practical interest.

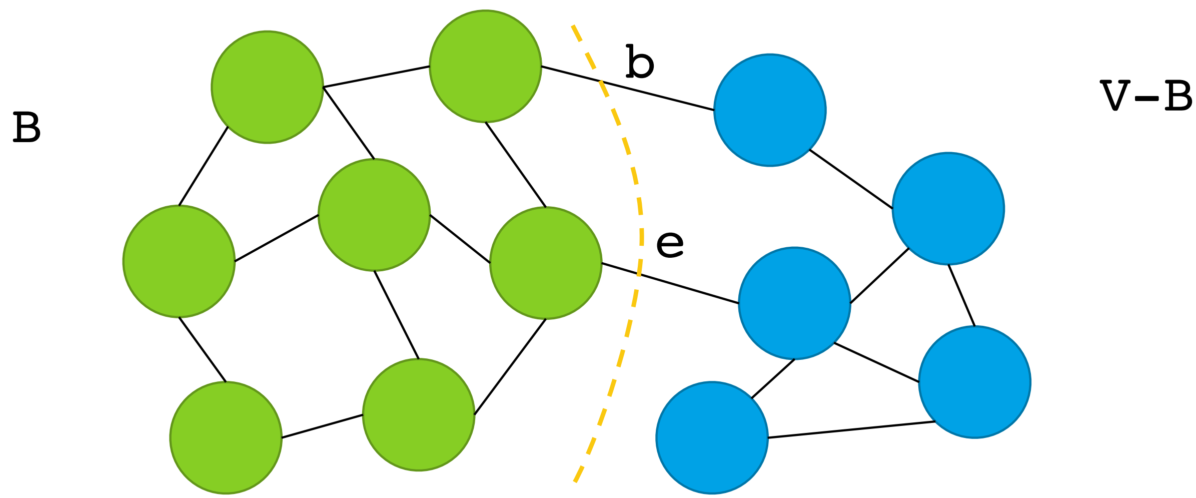
Preliminary ideas

- Build minimal spanning trees incrementally
- Show by induction that algorithm is correct at each step

- Concept: A set of edges $T \subset E$ is **promising** if it is a subset of a a minimal spanning tree (V, E')
    - Let $B = \{v \in V : v \in e$ for some $e \in T\}$
    - Then, (B, T) is a subgraph of (V, E) which is a tree

Graph Cuts

- A cut is any partition of the vertices into two groups, $\boldsymbol{B}$ and $\boldsymbol{V-B}$



    - with edges **b** and **e** joining the partitions

Useful Property

- **Lemma**: Let $T \subset E$, and $B \subset V, B \neq V$. Assume T is a promising set of edges for the MST problem in (V,E), and no edge in T leaves B (no edge across cut (B, V-B)). Let e^* be the smallest weight edge in E such that $e^* = \{i, j\}, i \in B, j \notin B$. Then, $T' = T \cup \{e^*\}$ is a promising set.
- Allows us to grow a promising set! (Induction)

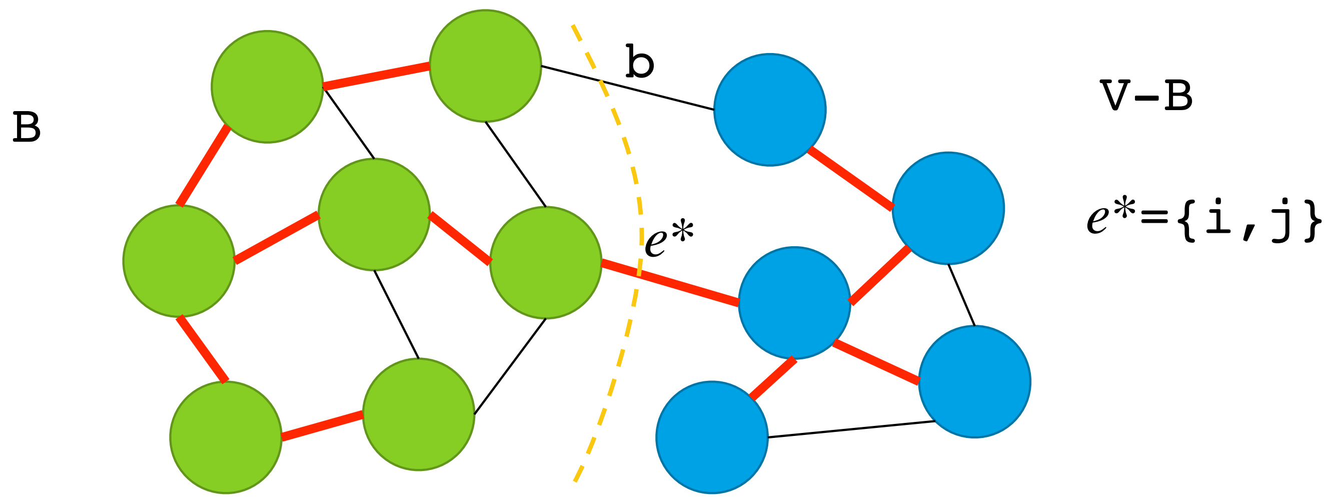
Proof.

(B, V-B) form a partition of V: a cut

Edge $e^* = \{i, j\}, i \in B, j \notin B$ has the smallest weight among all edges that cross the cut.

Case 1: MST includes e^*

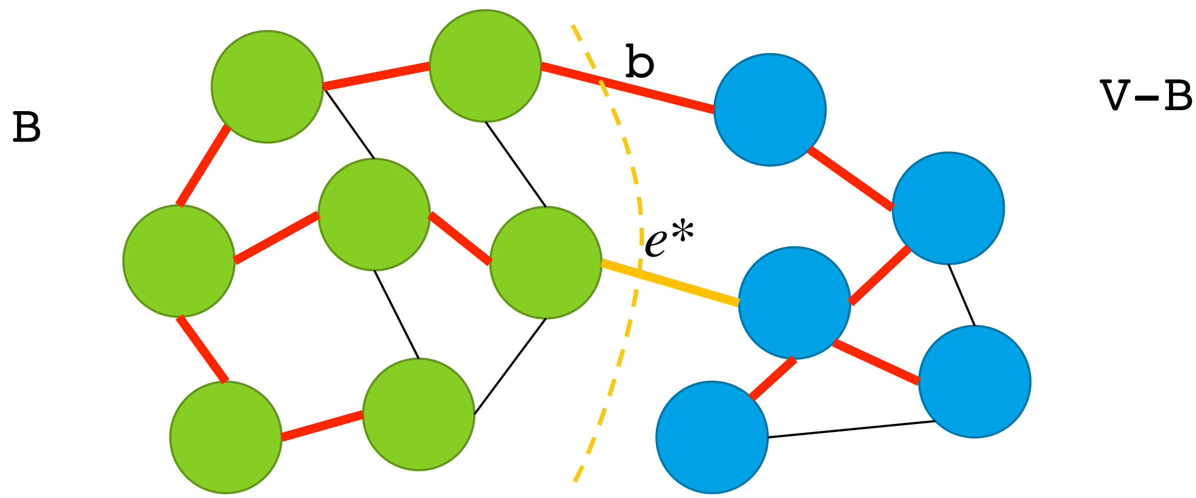
- Let MST (G, E') where $T \subset E'$ be such that $e^* \in E'$
- Simple: then $T \cup \{e^*\} \subset E'$ is promising



-

Case2: MST does not includes e

- Let MST (G, E') where $T \subset E'$ be such that $e^* \notin E'$
- Hard: Look at cycle including e. Swap b for e^* in E', total weight must not increase, hence have MST $(V, E' \cup \{e^*\} - \{b\})$



Proof of Correctness

- **Lemma**: Proof (cont)

Since T is promising, let MT = minimum spanning tree $(V, E'), T \subset E'$.

If $e^* \in E'$, then $T' = T \cup \{e^*\} \subset E'$ has no cycles, and is thus a promising set of edges, proving the theorem.

If $\{i, j\} \notin E'$, then $E' \cup \{e^*\}$ has one cycle that includes e^*, as it has #V edges and a tree must have at most #V-1 edges

In that cycle, there exists edge b that leaves B, hence $b \notin T$

Note $w(b) \geq w(e^*)$ by how we selected e^*

Note: $E' \cup \{e^*\} - \{b\}$ leaves graph connected, and has number of edges = #V-1, and its total weight is no greater than the weight of E'

Hence, (V, $E' \cup \{e^*\} - \{b\}$) is also MST, and $T' = T \cup \{e^*\} \subset E' \cup \{e^*\} - \{b\}$ so T' is a promising subset

Kruskal's Algorithm

- Sort edges by weight in ascending order
- Start with empty set T (note: it is promising)
- For each edge e in sorted list
  - If adding edge e to T does not create cycle in $(V, T \cup e)$
  - …add it to MST: $T = T \cup \{e\}$
  - Claim: T is now promising set with one more edge
- Stop when you have $\#V - 1$ edges in T

Proof of Correctness

- At any stage in algorithm, T is a forest (no cycles can be created)
- At any stage in algorithm, T is a promising set
  - True initially when $T = \varnothing$
  - If at a stage in algorithm, vertex $e*$ is the lowest weight edge remaining, and it does not form a cycle with T, then:
    - Let {a,b} = $e*$; let B be vertices in T connected to a
    - Then, no edge in T leaves B and $e*$ is minimum weight edge among edges that leave B
    - By lemma, $T \cup \{e*\}$ is promising set
- Thus, algorithm converges to MST

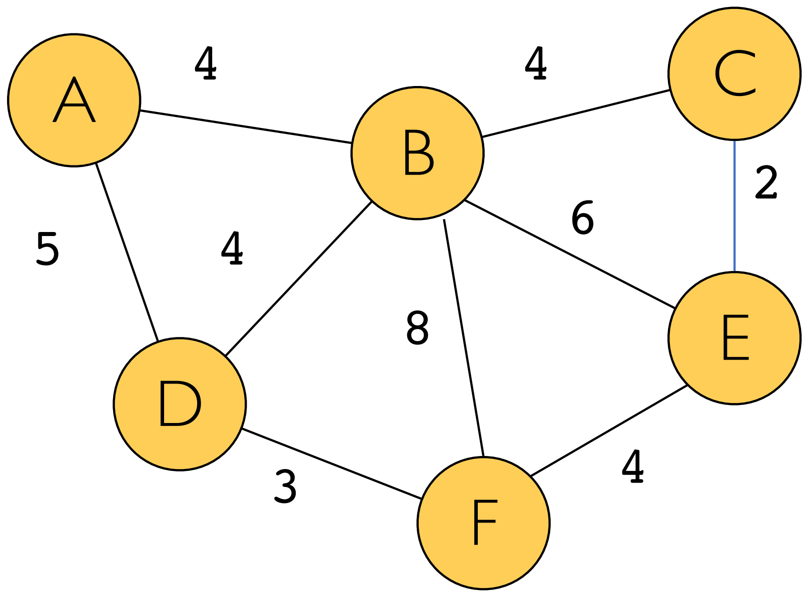
Kruskal

- How can we tell if adding edge will create cycle?
- Start by giving each vertex its own "cloud", which consists of all connected vertices in current T (Disjoint Sets)
- If both ends of lowest-cost edge are in same cloud
  - we know that adding the edge will create a cycle!
- When edge is added to MST
  - merge clouds of the endpoints

Kruskal Pseudo-Code

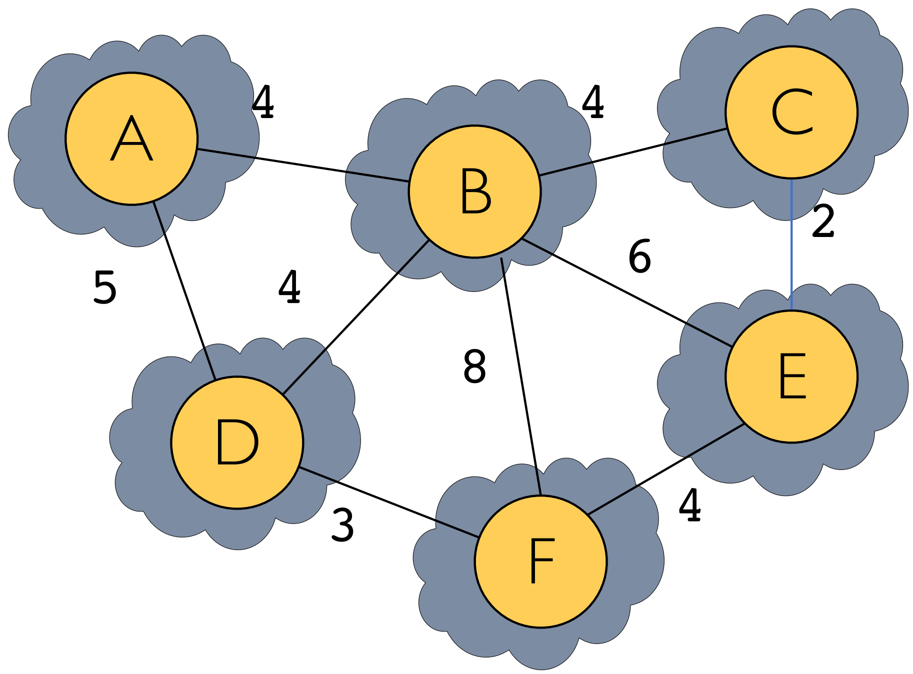
```
function kruskal(G):
   // Input: undirected, weighted graph G
   // Output: list of edges in MST
   for vertices v in G:
      makeCloud(v) // put every vertex into it own set
   MST = []
   Sort all edges
   for all edges (u,v) in G sorted by weight:
      if u and v are not in same cloud:
         add (u,v) to MST
         if size(MST) = |V| - 1:
            break
         merge clouds containing u and v
   return MST
```

Example



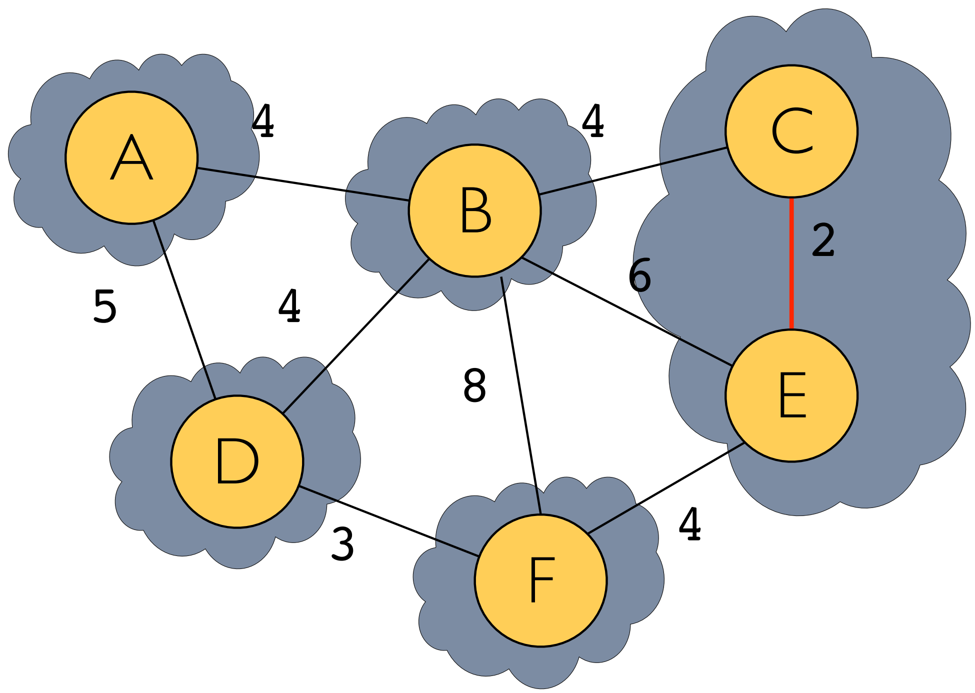

```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



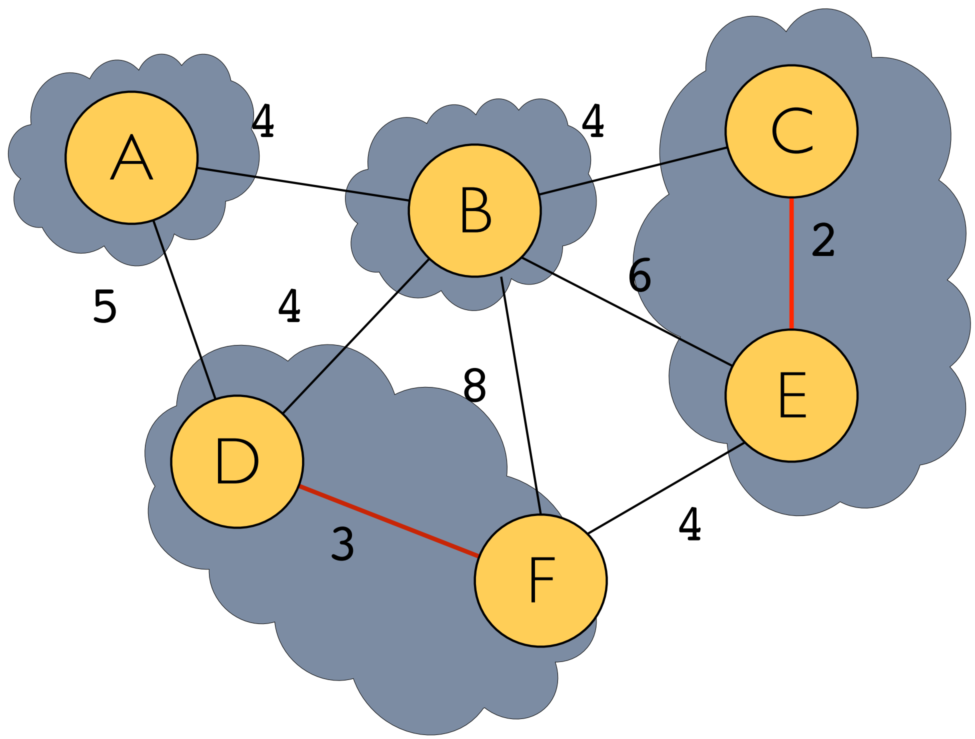

```
edges = [(C,E),(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



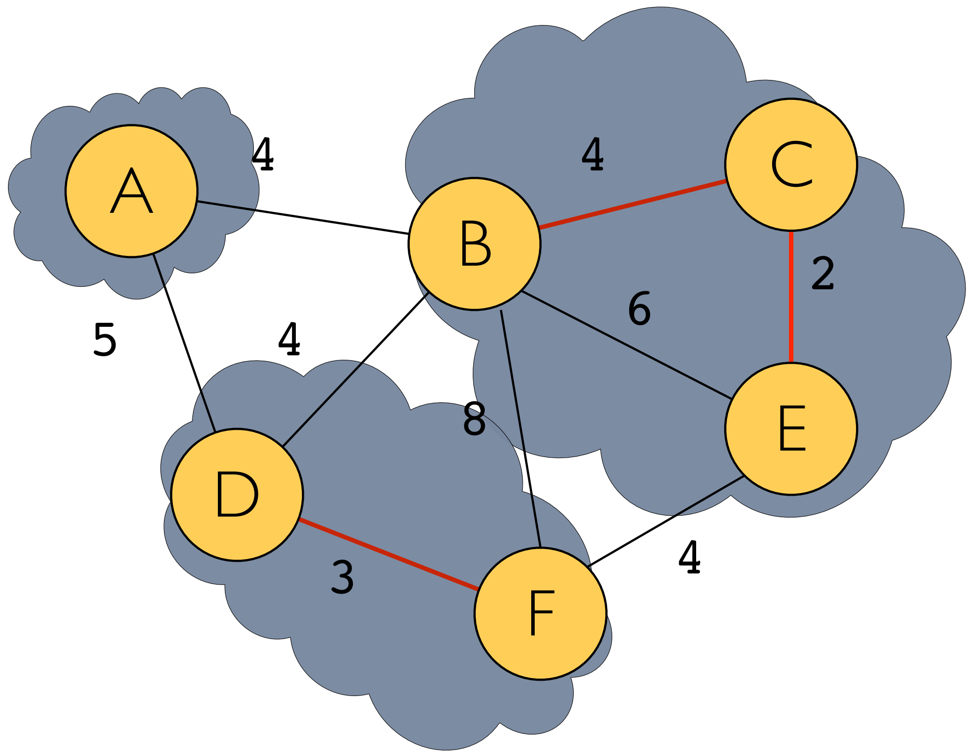

```
edges = [(D,F),(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



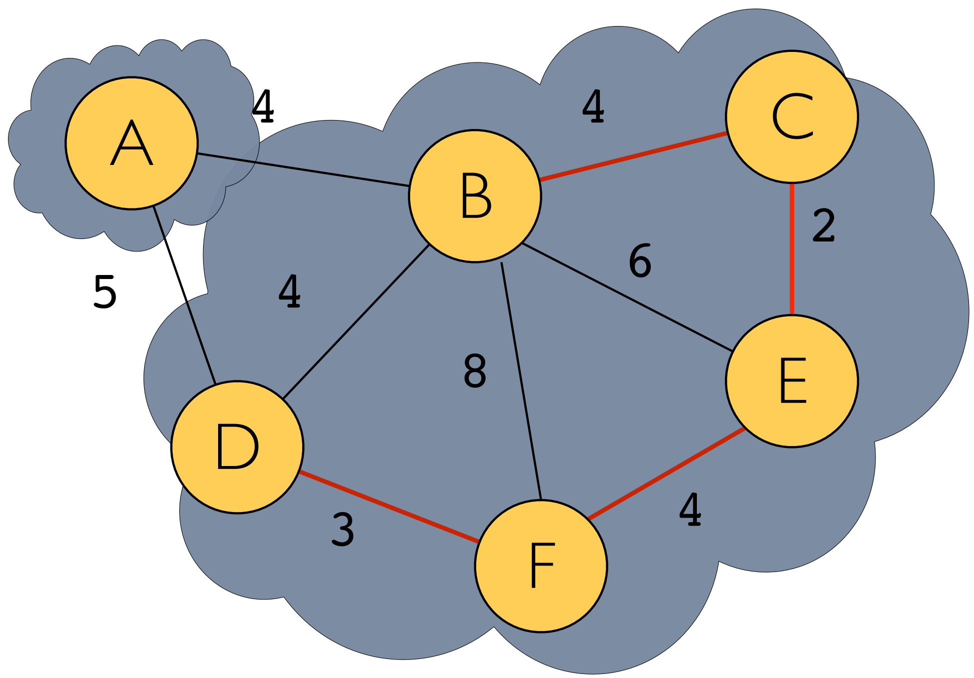

```
edges = [(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example



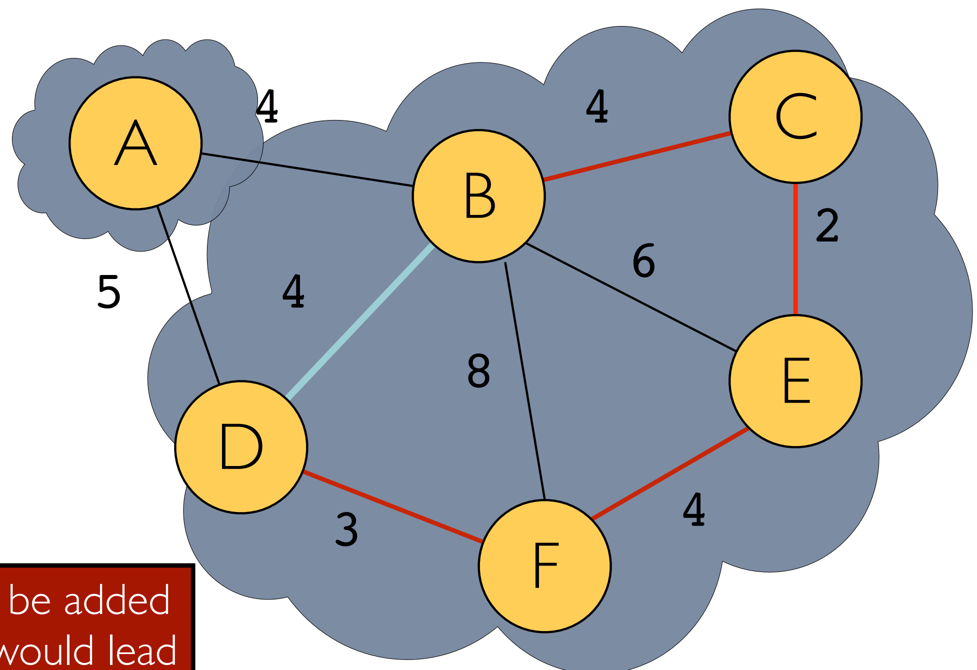

```
edges = [(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
```

Example




```
edges = [(B,D),(A,B),(A,D),(B,E),(B,F)]
```

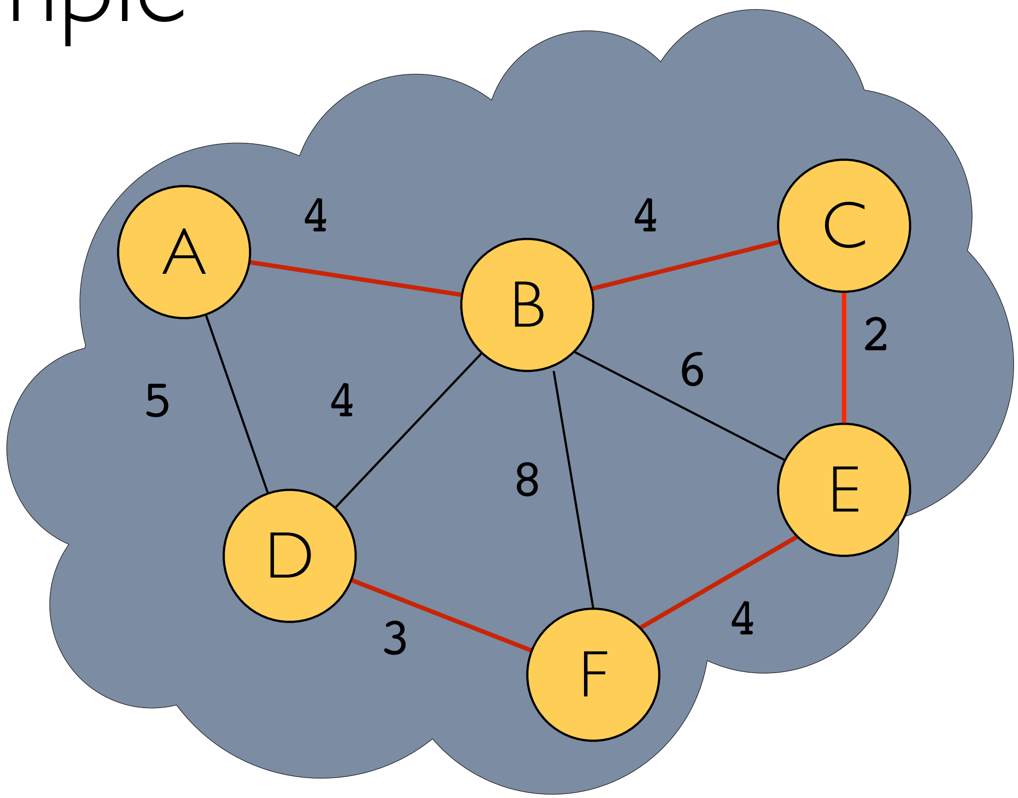
Example



BD cannot be added because it would lead to a cycle

```
edges = [(A,B),(A,D),(B,E),(B,F)]
```

Example




```
edges = [(A,D),(B,E),(B,F)]
```

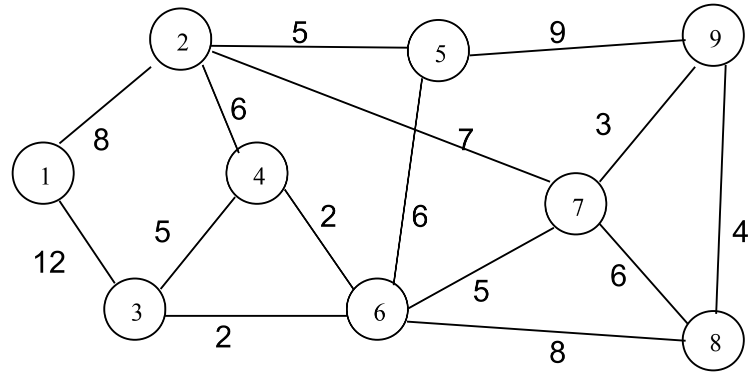
Merging Clouds Efficiently (Naive way)

- Keep track of clouds using disjoint sets (Union-Find)
  - Edge forms a cycle if both ends of the edge belong to the same disjoint set (have the same root!)
  - Checking that an edge forms a cycle has amortized complexity $O(\log^*(|V|)) \approx O(1)$

Kruskal Runtime

- $O(|V|)$ for iterating through vertices
- $O(|E|\log|E|)$ for sorting edges
- $O(|E|\times 1)$ for iterating through edges and merging clouds with path compression
- $O(|V|+|E|\log|E|+|E|\times 1)$
  - $= O(|V|+|E|\log|E|)$
- $O(|V|+|E|\log|E|)$
  - Better than simple $O(|V|^3)$ without disjoint sets

Example



Disjoint components: Minimum Spanning Tree

1 2 3 4 5 6 7 8 9 Empty

1) Add {3,6}

Disjoint components: Minimum Spanning Tree

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

|

6

2) Add {4,6}

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

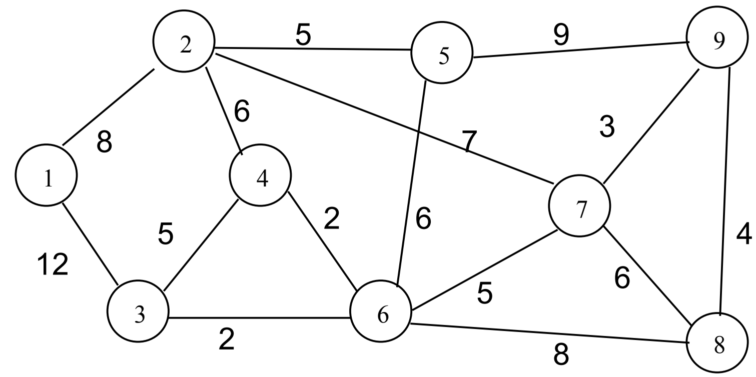
/ \

6 4

Sorted edges

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

Example



Disjoint components:

3) Add {7,9}

```
 3     7   1  2   5   8
/ \    |
6  4   9
```

4) Add {8,9}

```
 3       7   1  2   5
/ \     /  \
6  4   9   8
```

Minimum Spanning Tree

3--6--4   7--9

3--6--4   7--9--8

Sorted edges

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

Example

Disjoint components:

5) Discard {3,4} (cycle); add {6,7}

```
   3       1  2   5
  /  | \
6  4   7
       /  \
     9    8
```

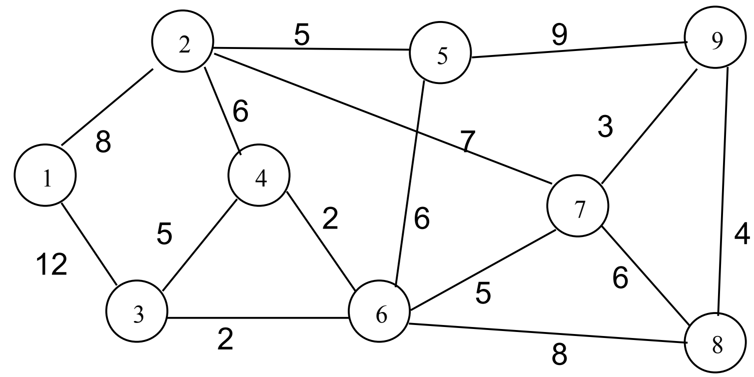
6) Add {2,5}

```
   3       1  2
  /  | \         \
6   4   7         5
       /  \
     9    8
```

Minimum Spanning Tree

```
3--6--4
       \
        7--9--8
```

```
3--6--4      2--5
      \
       7--9--8
```



Sorted edges

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

Example

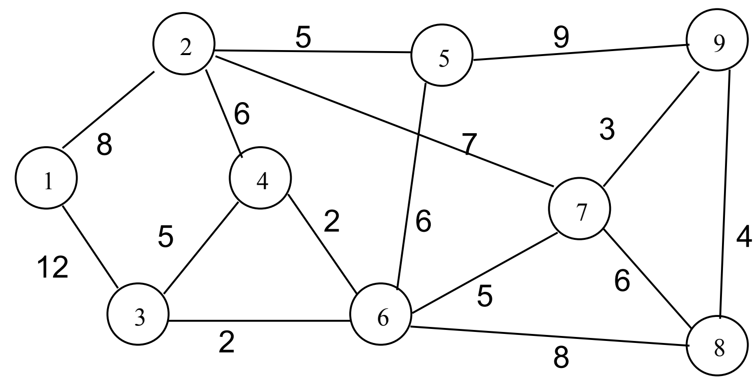
Disjoint components: Minimum Spanning Tree

7) Add {2,4}

```
  ( 3   )       1                3--6--4--2--5
 /  | \   \                          \
6  4   7   2                         7--9--8
      /  \   \
     9    8    5
```

8) Discard {5,6},{7,8}, {2,7}. Select arc {1,2} . DONE

```
 (    3       )                3--6--4--2--5
  /  | \   \  \                    \      \
6  4   7   2  1                      \      1
      /  \   \                        7--9--8
     9    8    5
```



Sorted edges

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

Different MST Algorithm: Prim-Jarnik

- Traverse G = (V,E) starting at any node
  - Maintain priority queue of nodes (e.g. binary heap, Fibonacci heap)
  - set priority to weight of the cheapest edge that connects them to MST
- Un-added nodes start with priority ∞
- At each step
  - Add the node with lowest cost to MST
  - Update ("relax") neighbors as necessary
- Stop when all nodes added to MST

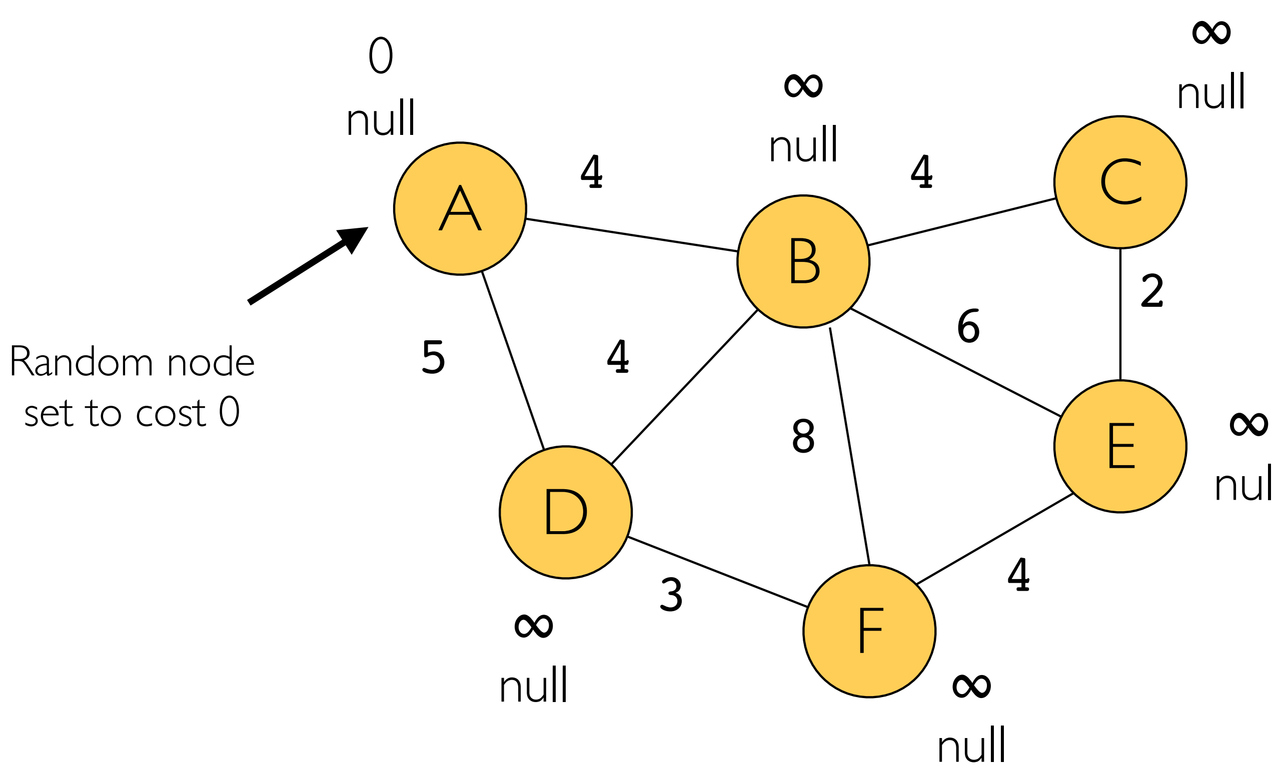
Different MST Algorithm: Prim-Jarnik

- Traverse G = (V,E) starting at any vertex v
- Form priority queue PQ of nodes (e.g. binary heap, Fibonacci heap)
  - Set v.dist = 0, v'.dist = ∞, v' different from v. dist will be the key used in the priority queue
  - Set parent of all vertices, v.pred to NULL; set $T = \emptyset$
- At each step:
  - Remove min from priority queue: call it v. If v.pred not null, add {v,v.pred} to T
  - For all neighbors v' of v: if v' is in PQ and $w(v, v') <$ v'.dist:
    - Decrease v'.dist to $w(v, v')$, update v'.pred = v
- Stop when PQ is empty

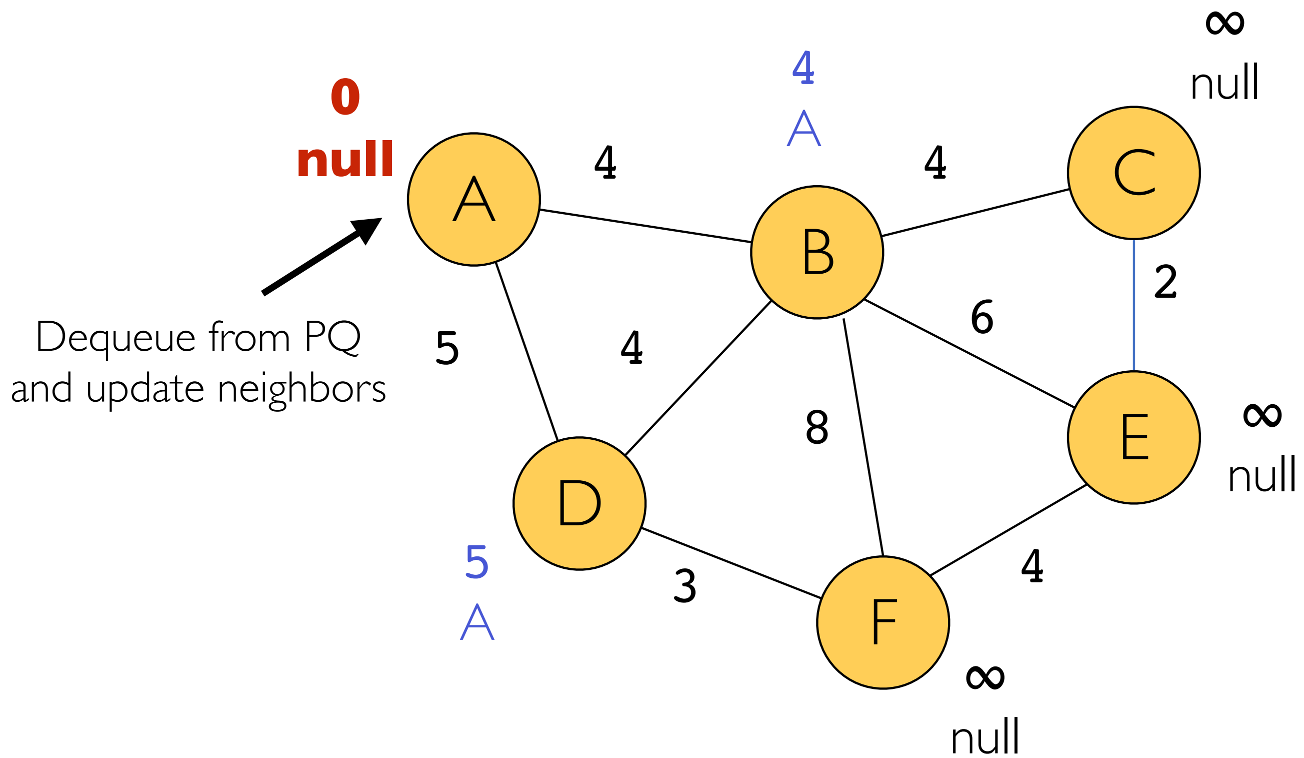
Pseudo-code

```
function prim(G):
   // Input: weighted, undirected graph G with vertices V
   // Output: list of edges in MST
   for all v in V:
      v.cost = ∞
      v.prev = null
   s = a random v in V // pick a random source s
   s.cost = 0
   MST = []
   PQ = PriorityQueue(V) // priorities will be v.cost values
   while PQ is not empty:
      v = PQ.removeMin()
      if v.prev != null:
         MST.append((v, v.prev))
      for all incident edges (v,u) of v such that u is in PQ:
         if u.cost > (v,u).weight:
            u.cost = (v,u).weight
            u.prev = v
            PQ.decreaseKey(u, u.cost)
  return MST
```

Example

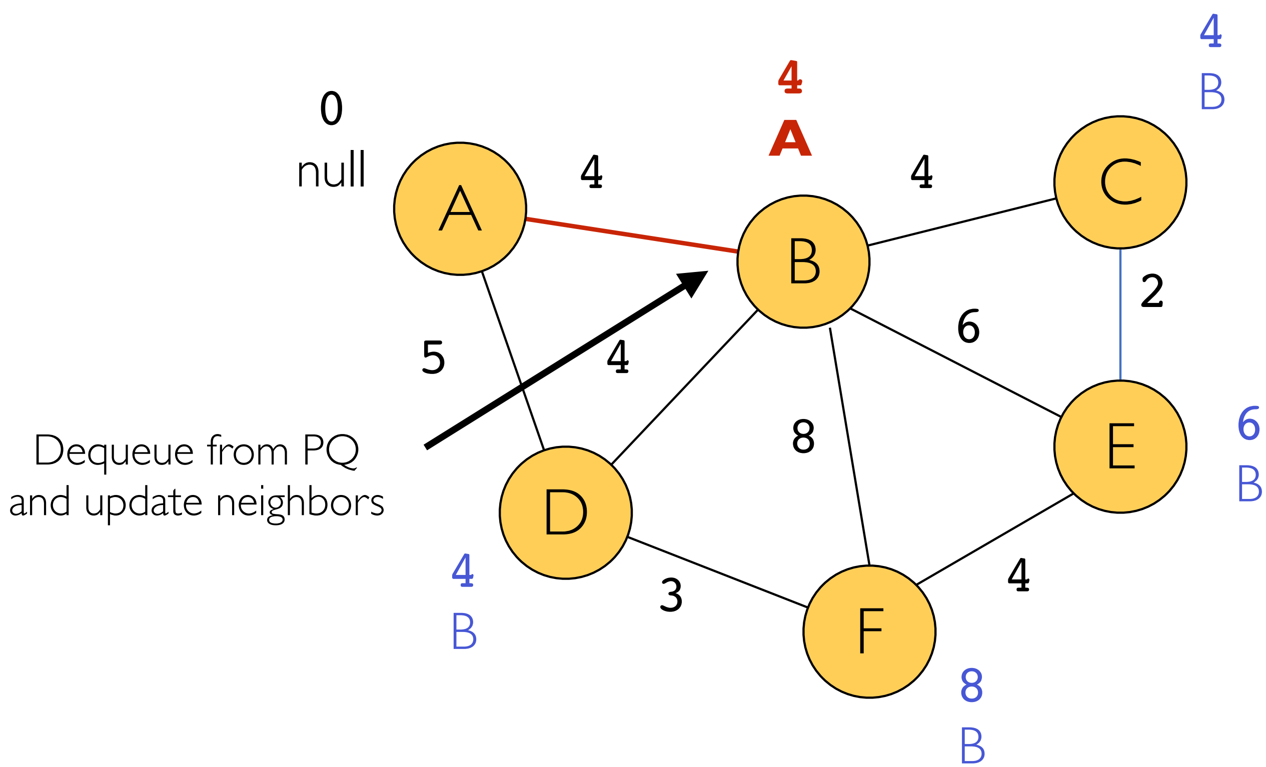


PQ = [(0,A),(∞,B),(∞,C),(∞,D),(∞,E),(∞,F)]



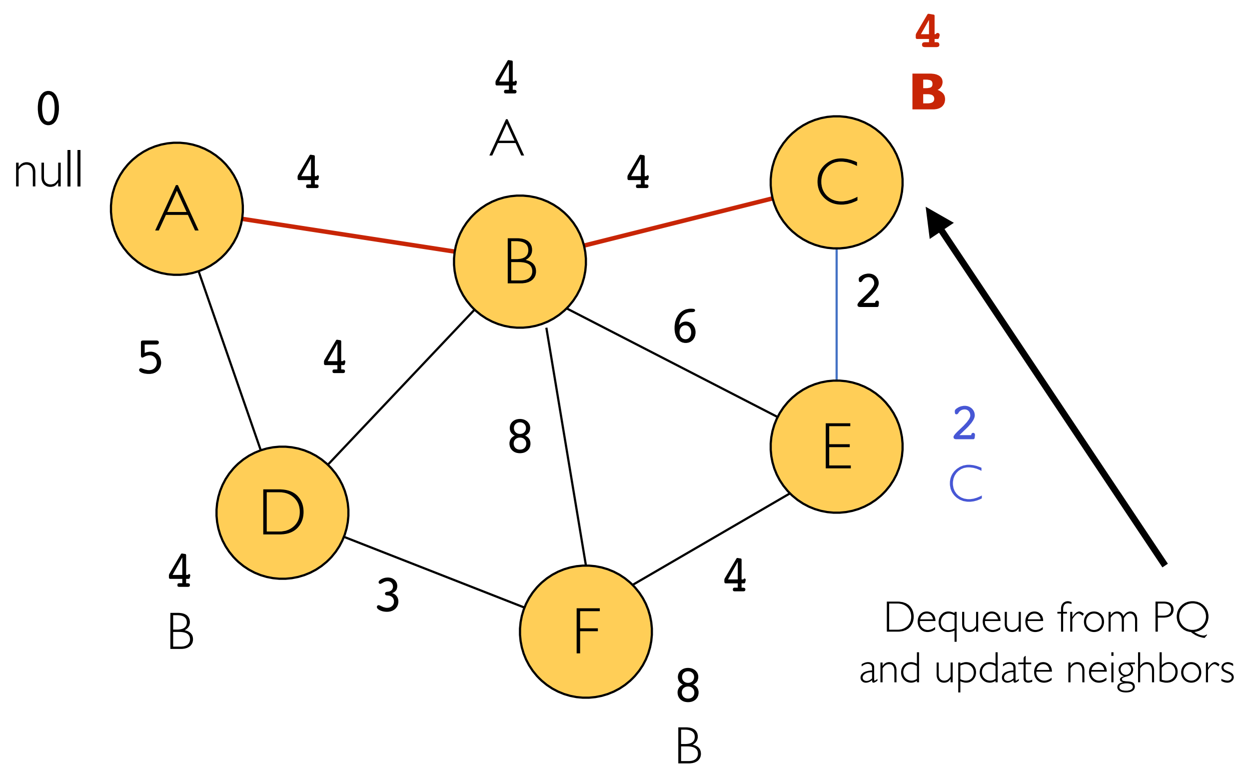
0
null
Dequeue from PQ
and update neighbors
A
B
C
D
E
F
4
A
∞
null
5
A
∞
null
∞
null
4
4
5
4
6
2
8
3
4

PQ = [(4,B),(5,D),(∞,C),(∞,E),(∞,F)]



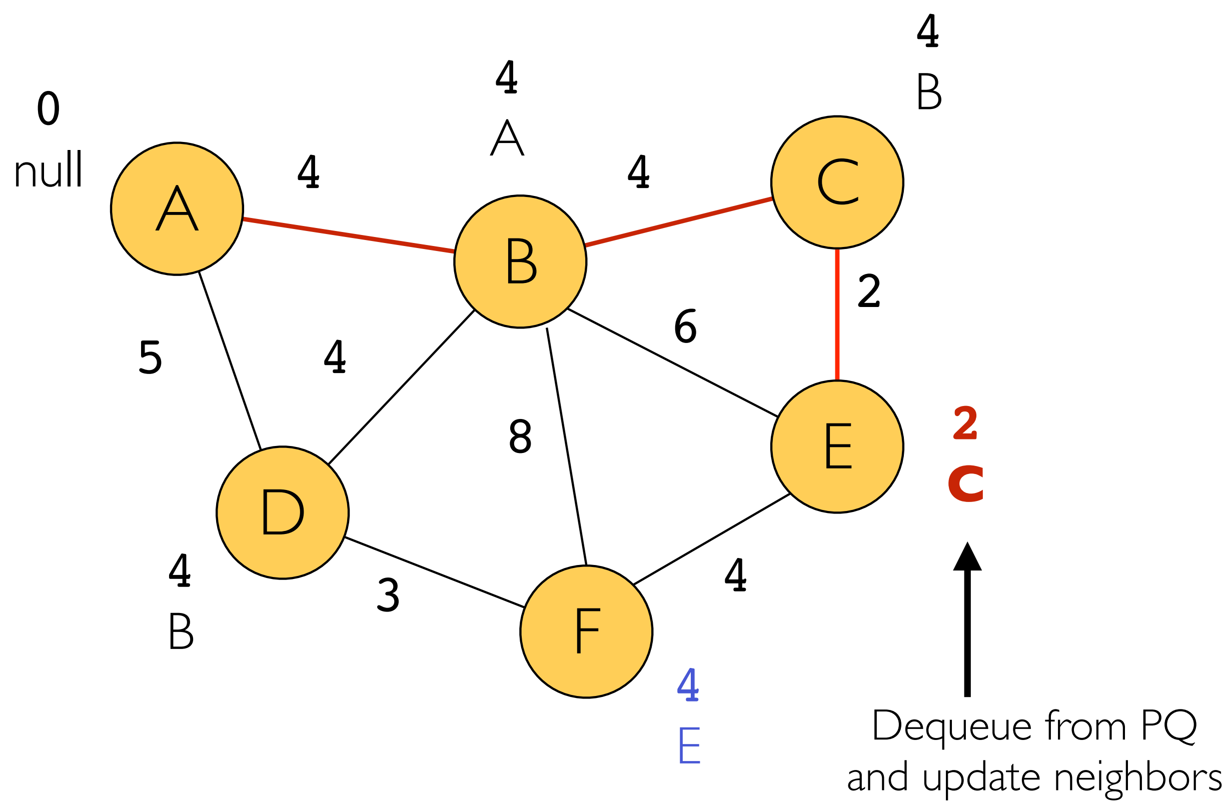
4
A
0
null
4
B
A
4
C
B
5
4
4
6
2
Dequeue from PQ
and update neighbors
8
E
6
B
D
4
B
3
F
4
8
B

PQ = [(4,C),(4,D),(6,E),(8,F)]

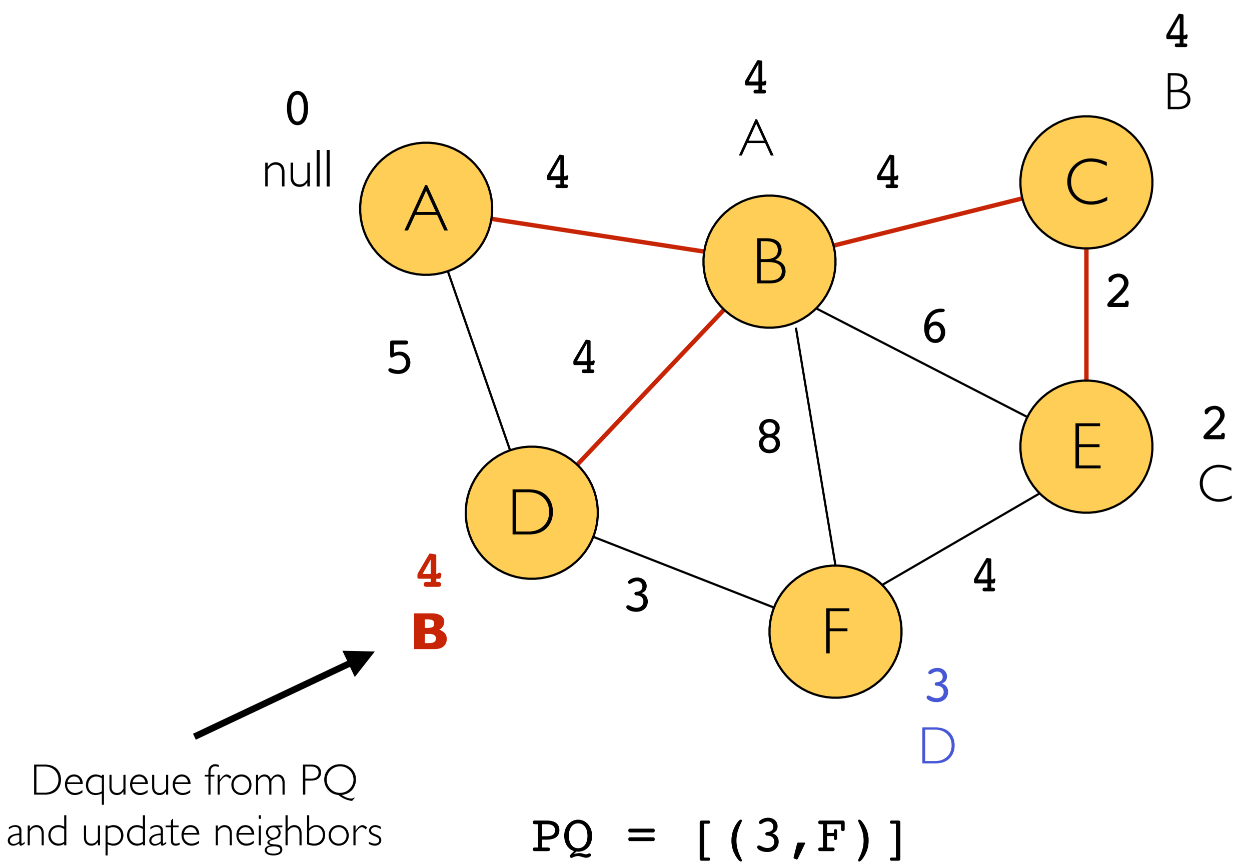


0
null
4
A
4
B
A
B
C
D
E
F
4
4
2
6
5
4
8
3
4
2
C
4
B
8
B
Dequeue from PQ
and update neighbors

PQ = [(2,E),(4,D),(8,F)]

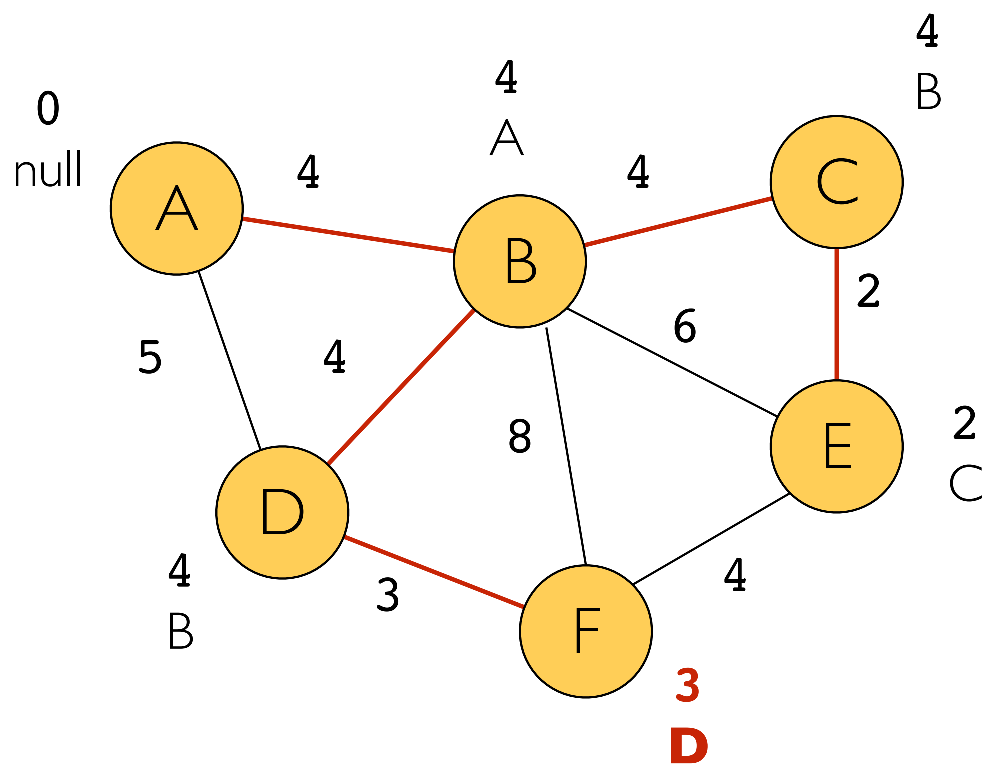

0
null
4
A
4
B
A
B
C
D
E
F
4
4
2
5
4
6
8
3
4
4
B
2
C
4
E
Dequeue from PQ
and update neighbors

PQ = [(4,D),(4,F)]



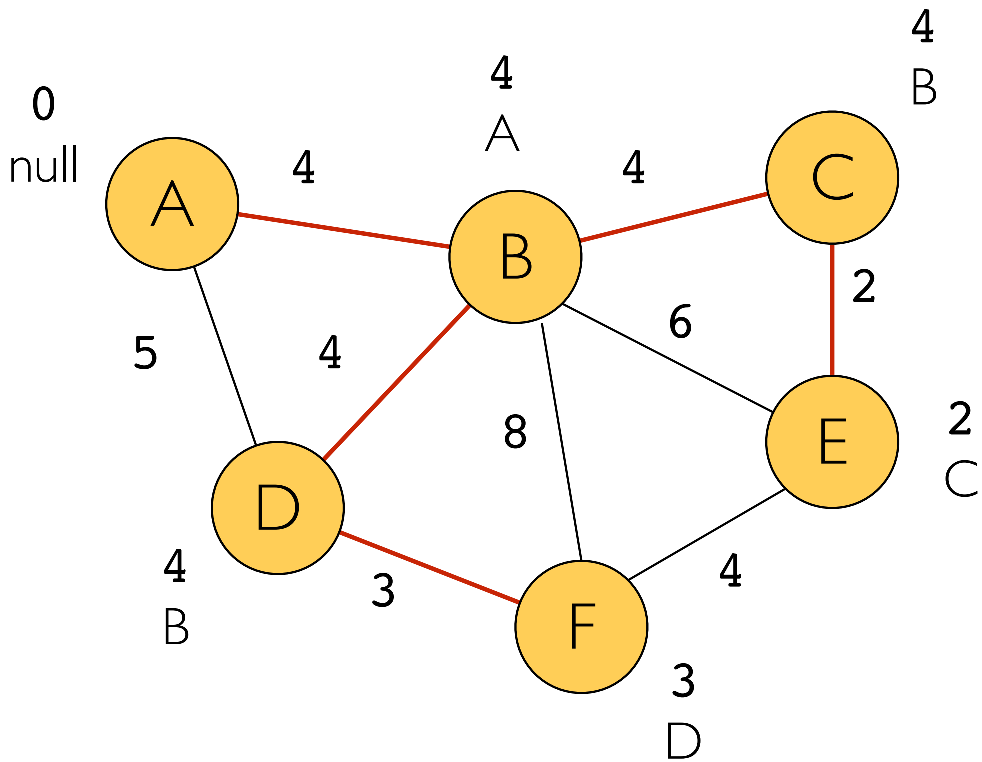
0
null
A
4
A
B
4
4
B
C
2
6
5
4
8
E
2
C
D
4
B
3
F
4
3
D
Dequeue from PQ
and update neighbors

PQ = [(3,F)]



`PQ = [ ]`

Dequeue from PQ
and update neighbors


0
null
A
4
4
A
B
4
4
B
C
2
6
5
4
8
E
2
C
D
4
B
3
4
F
3
D

Proof of Correctness

- Let T be current tree at iteration step
- Claim: T is promising set
  - True initially, as $T = \varnothing$
  - Define B = vertices in T: t any stage, T is a tree over a subset of nodes $B \subset V$
  - Let v be min vertex in PQ; then {v.parent, v} leaves B
    - v has parent in B, and v is still in PQ, so it is not in B
    - If v in PQ, v.dist is set by min weight edge leaving B
  - Thus, {v.pred, v} is min-weight edge leaving B
  - Hence, $T \cup \{\{v.pred, v\}\}$ is promising set after remove min extracts v

Runtime Analysis

- Initializing nodes with distance and previous pointers is O(|V|); putting nodes in PQ is O(|V|)
- While loop runs |V| times
  - removing vertex from PQ is O(log|V|)
  - So O(|V|log|V|)
- For loop (in while loop) runs |E| times in total
  - Determining whether v' is in PQ: O(1) if we build index into PQ (need to find location; not easy!)
  - Decreasing vertex's key in the PQ is log|V| (binary heap), or amortized to $O(1)$ if we use Fibonacci or rank-pairing heaps
  - So O(|E|) in complex data or O(|E| log|V|)
- Overall runtime
  - O(|V| + |V|log|V| + |E|)
    - = O(|E| + |V|log|V|) best

Example

Priority Q

(1,0)

(2,M) (3,M)

(4,M) (5,M) (6,M) (7,M)

(8,M) (9,M)

1) Scan 1, reduce key of 2, 3; remove 1

(2,8)

(9,M) (3,12)

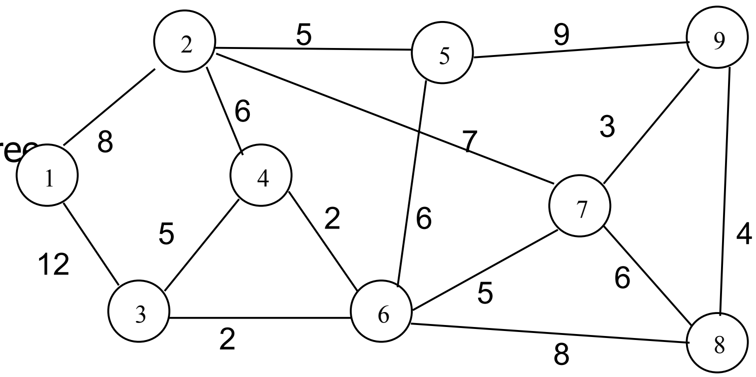
(4,M) (5,M) (6,M) (7,M)

(8,M)

Minimum Spanning Tree

Empty

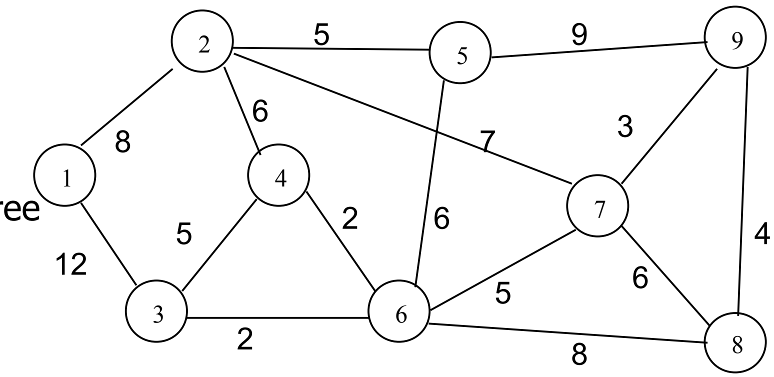
1



Sorted edges

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

Example



2) Scan 2, reduce key of 4,5,7. Remove 2.

Priority Q

(5,5)

(4,6) (7,7)

(9,M) (8,M) (6,M) (3,12)

Minimum Spanning Tree

1—2

3) Scan 5, reduce key of 6, 9; remove 5

(4,6)

(9,9) (6,6)

(3,12) (8,M) (7,7)

1—2—5

Sorted edges

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

Example

4) Remove 4, reduce key of 3,6.

Priority Q

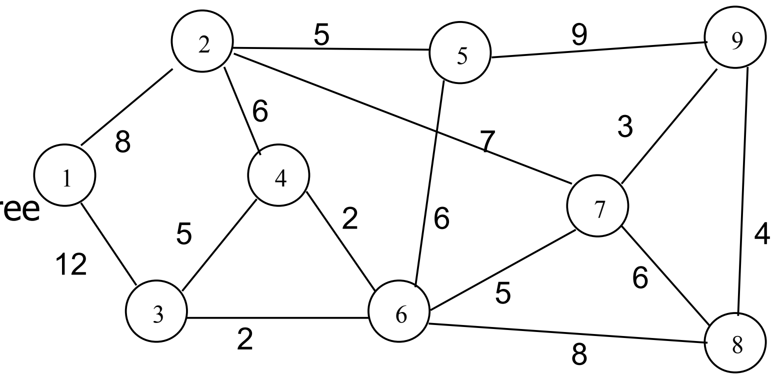
(6,2)

(3,5)　　　(7,7)

(9,9)　(8,M)

Minimum Spanning Tree

1—2—5
|
4



5) Remove 6, reduce key of 3, 7, 8;

(3,2)

(9,9)　　　(7,5)

(8,8)

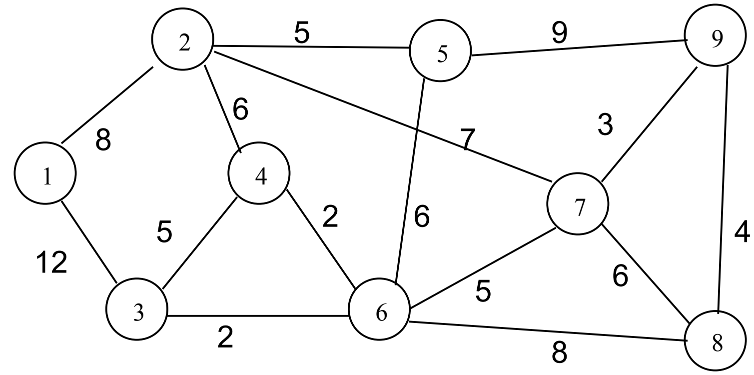
—

1—2—5
|
4 — 6

Sorted edges

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

Example



6) Remove 3.

Priority Q

(7,5)

(9,9) (8,8)

Minimum Spanning Tree

1—2—5

|

4 — 6 — 3

5) Remove 7, reduce key of 8,9;

(9,3)

(8,6)

1—2—5

| /— 7

4 — 6 — 3

6) Remove 9, reduce key of 8

(8,4)

1—2—5

| /— 7— 9— 8

4 — 6 — 3

7) Remove 8. DONE

Sorted edges

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

Summary

- Initializing nodes with distance and previous pointers is O(|V|); putting nodes in PQ is O(|V|)
- While loop runs |V| times
  - removing vertex from PQ is O(log|V|)
  - So O(|V|log|V|)
- For loop (in while loop) runs |E| times in total
  - Determining whether v' is in PQ: O(1) if we build index into PQ (need to find location; not easy!)
  - Decreasing vertex's key in the PQ is log|V| (binary heap), or amortized to $O(1)$ if we use Fibonacci or rank-pairing heaps
  - So O(|E|) in complex data or O(|E| log|V|)
- Overall runtime
  - O(|V| + |V|log|V| + |E|)
    - = O(|E| + |V|log|V|) best