Graphs
Minimum Spanning Trees

CSE 373 - Data Structures

May 29, 2002

Readings and References

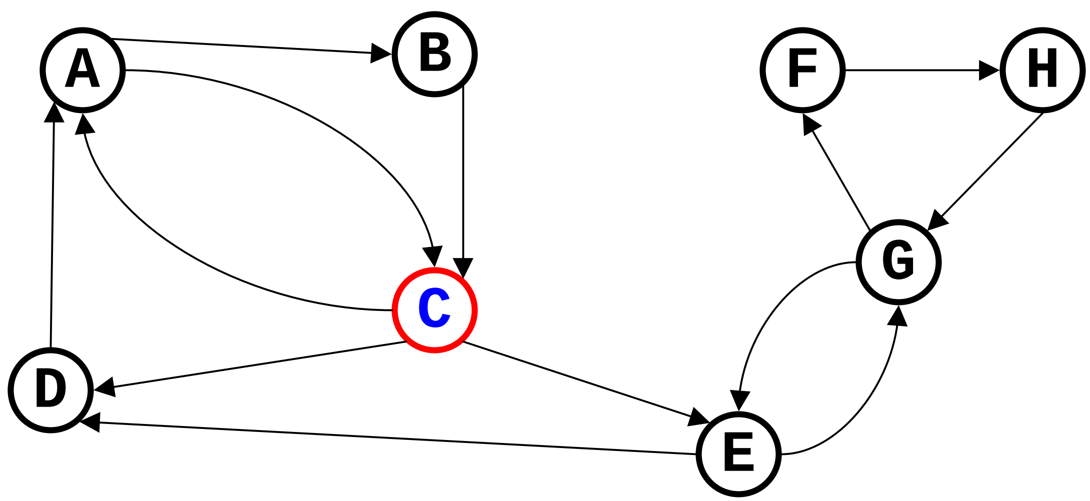
- Reading
  - › Section 9.5, *Data Structures and Algorithm Analysis in C*, Weiss
- Other References

Breadth First Search (BFS)

- We used Breadth First Search for finding shortest paths in an unweighted graph
  - › Use a queue to explore neighbors of source vertex, neighbors of each neighbor, and so on: 1 edge away, two edges away, etc.
- BFS spreads out like ripples in a pond
  - › all nodes at a given distance are looked at before we go any further outward

Breadth-First Search

- Basic Idea: Starting at node s, find vertices that can be reached using 0, 1, 2, 3, …, N-1 edges



Breadth-First Search Algorithm

- Uses a queue to track vertices that are “nearby”
- source vertex is **`s`**

```
Distance[s] = 0
Enqueue(s)
While queue is not empty
    X = dequeue a vertex
    For each vertex Y that is (adjacent to X and not
      previously visited)
      Distance[Y] = Distance[X] + 1
      Previous[Y] = X
      Enqueue Y
```

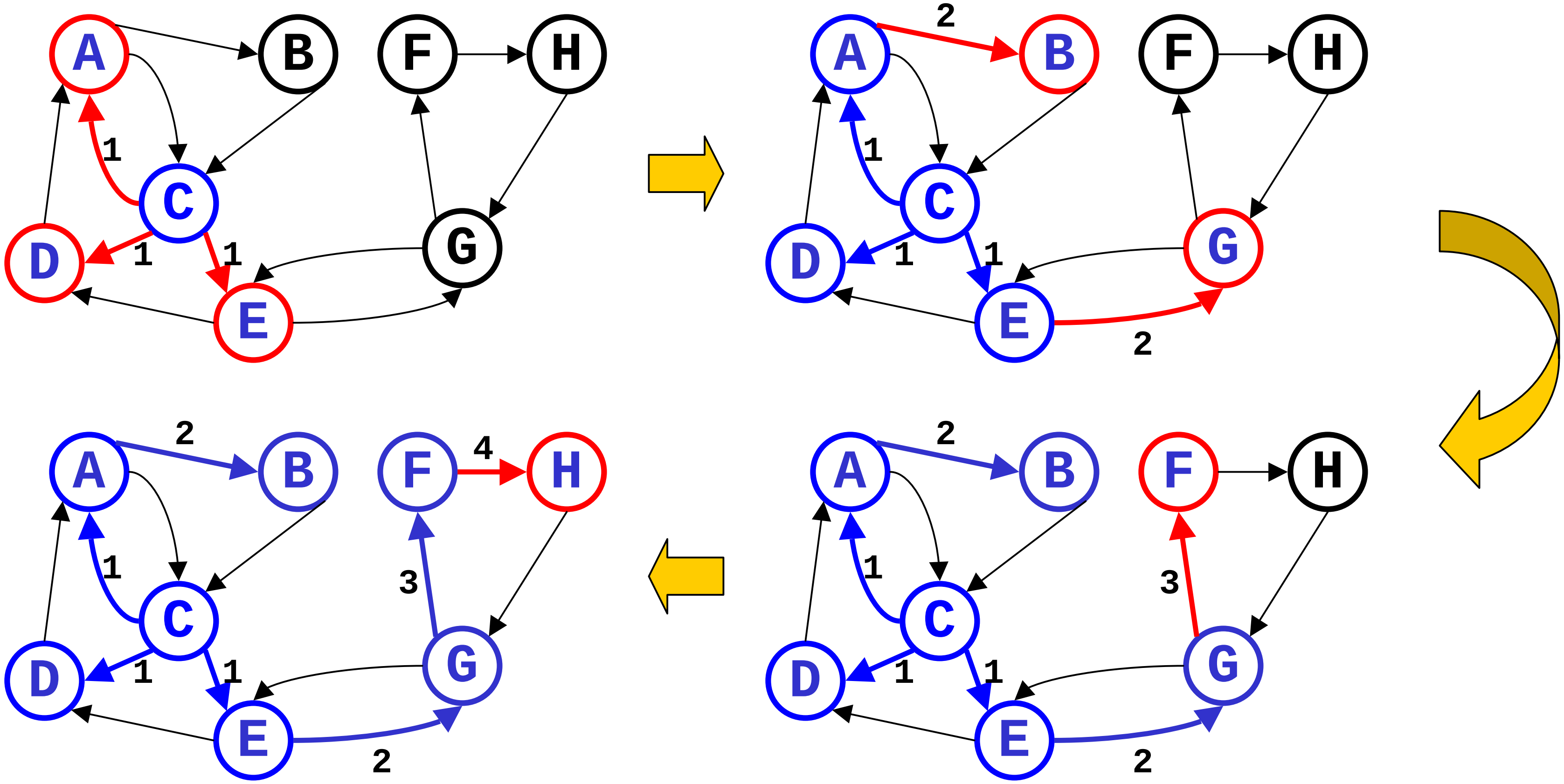
For each vertex

For each edge incident with that vertex

- Running time (same as topological sort) = **O(|*V*| + |*E*|)**

Breadth-First Search

- BFS(C): Starting at node C, find vertices that can be reached using 0, 1, 2, 3, …, N-1 edges



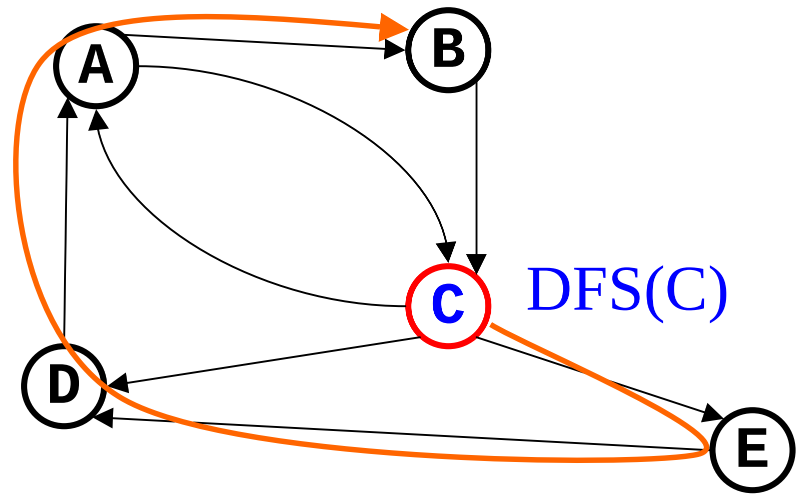
Depth First Search (DFS)

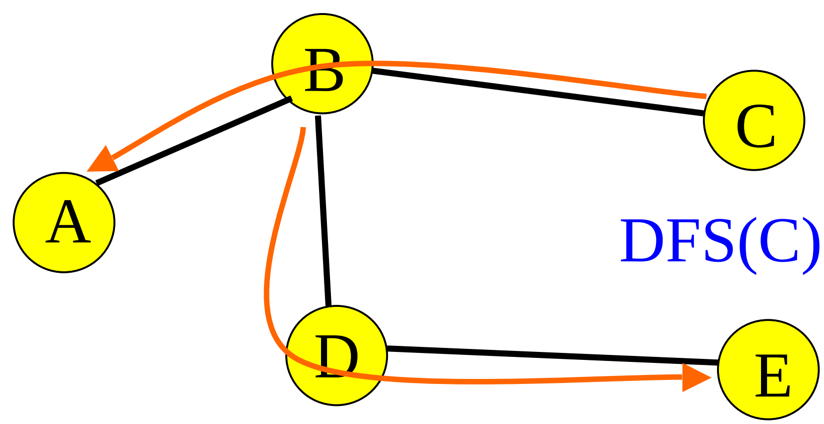
- A second way to explore all nodes in a graph
- DFS searches down one path as deep as possible
  - › When no new nodes available, it *backtracks*
  - › When backtracking, we explore side-paths that weren’t taken
- DFS allows an easy recursive implementation
  - › So, DFS uses a stack while BFS uses a queue

DFS Pseudocode

- Pseudocode for DFS:

```
DFS(v)
If v is unvisited
   mark v as visited
   print v (or process v)
   for each edge (v,w)
     DFS(w)
```



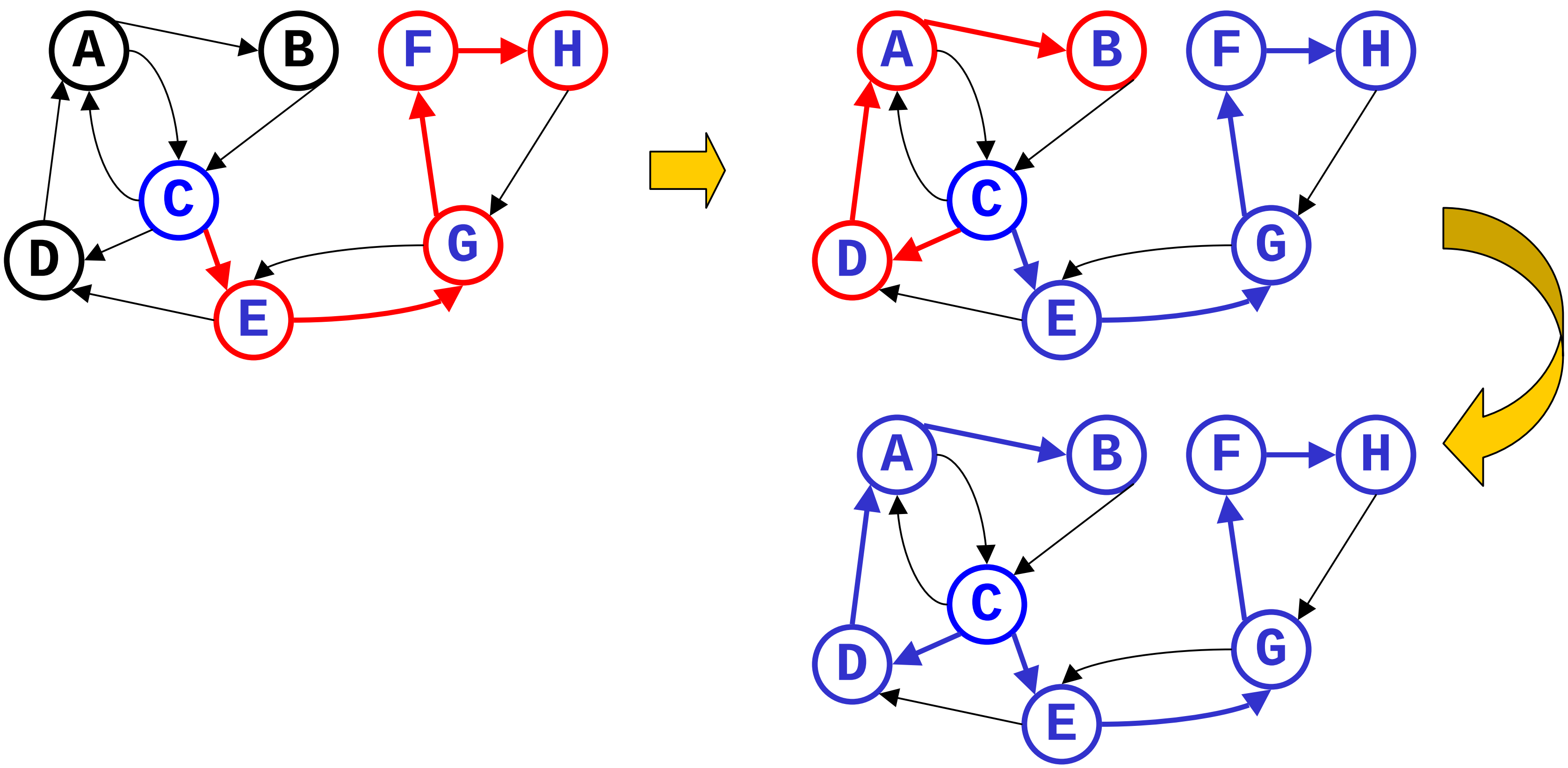


- Works for directed or undirected graphs

- Running time = **O(|*V*| + |*E*|)**

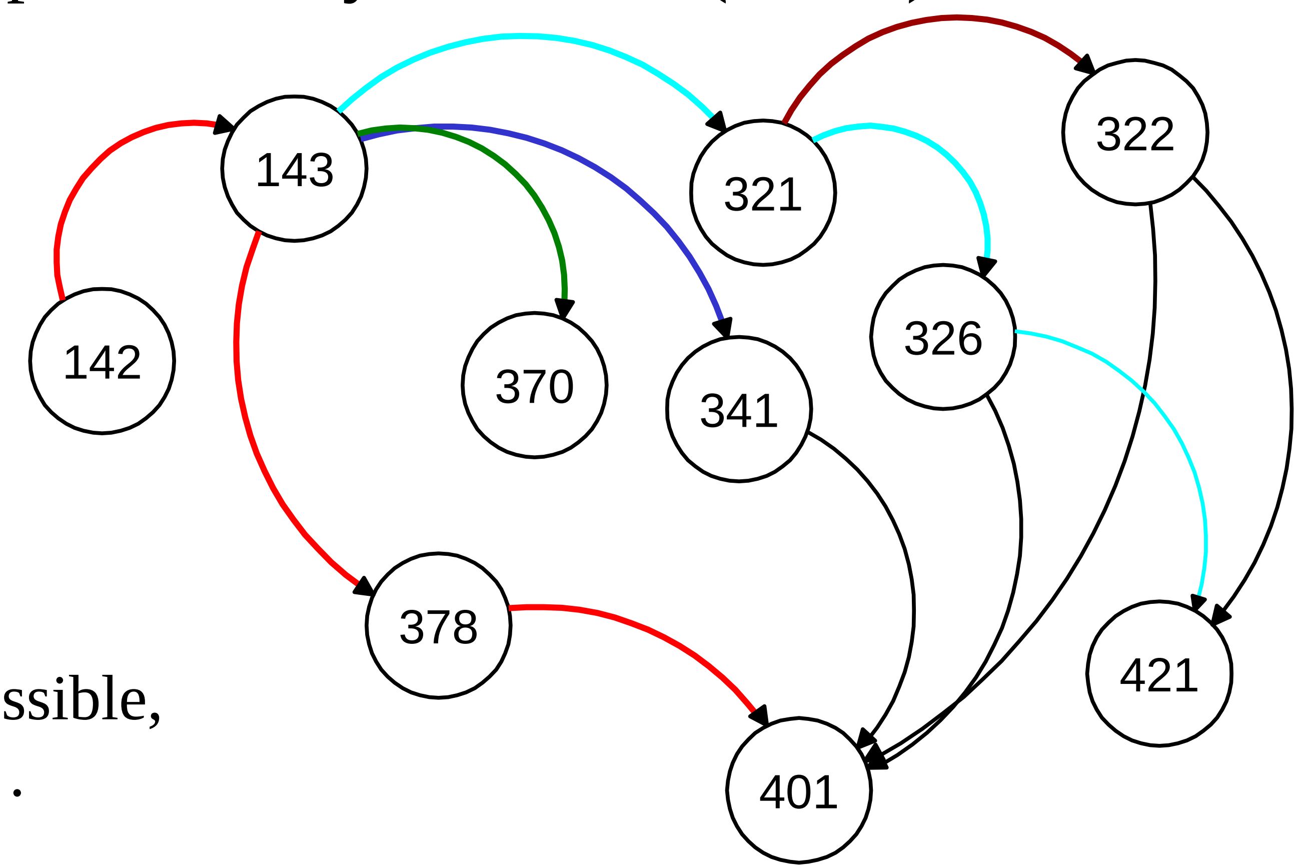
Depth-First Search

- DFS(C): searches down one path as deeply as possible, then backtracks and does it again



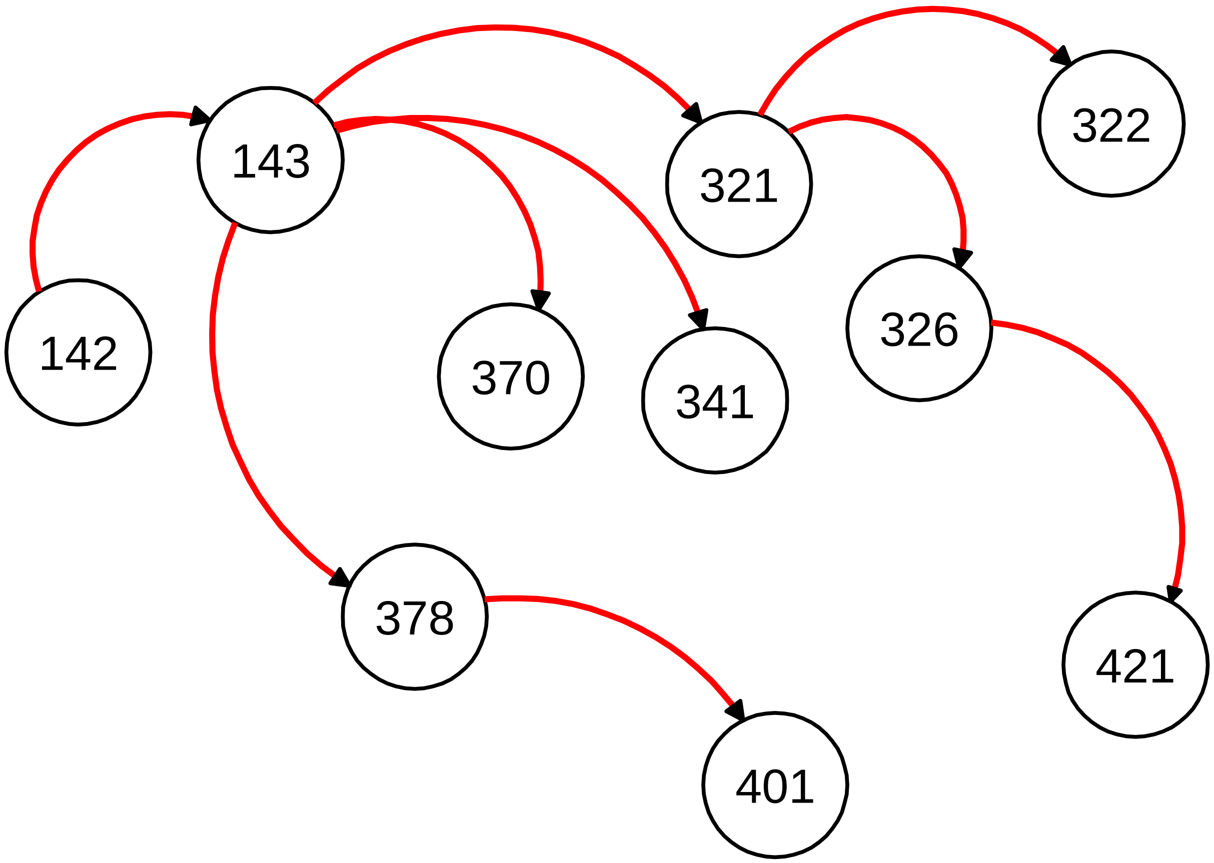
What about DFS on this graph?

- What happens when you do DFS(“142”)?

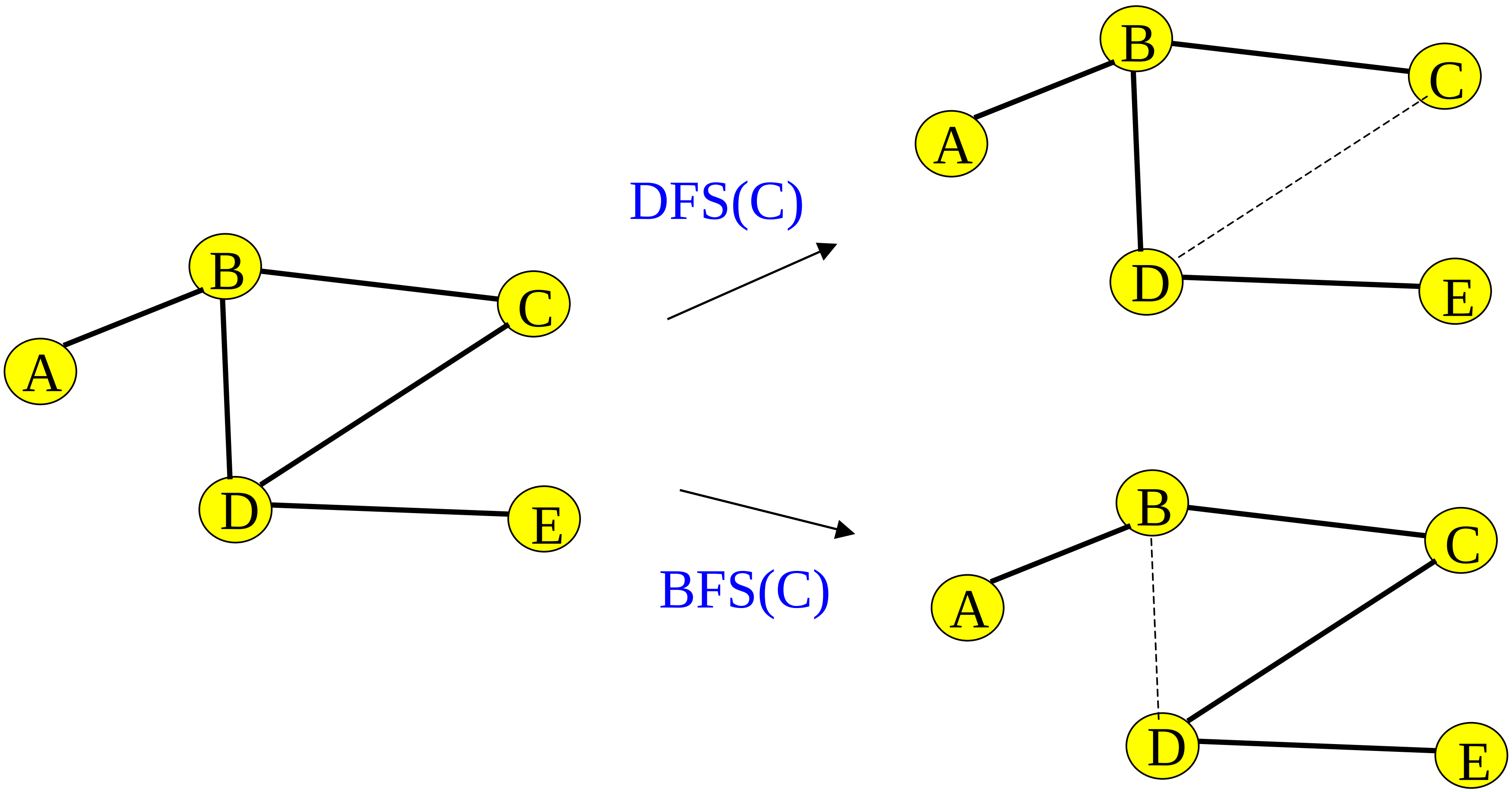


Go as deep as possible,
Then backtrack…

We get a “spanning” tree…

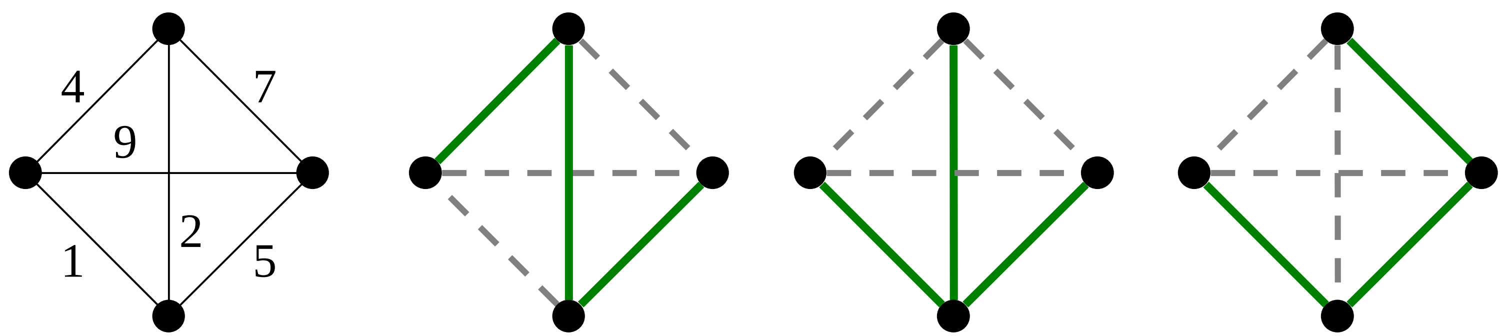


DFS and BFS may give different trees…



Spanning Tree Definition

- *Spanning tree*: a subset of edges from a connected graph that:
  - touches all vertices in the graph (*spans* the graph)
  - forms a tree (is connected and contains no cycles)

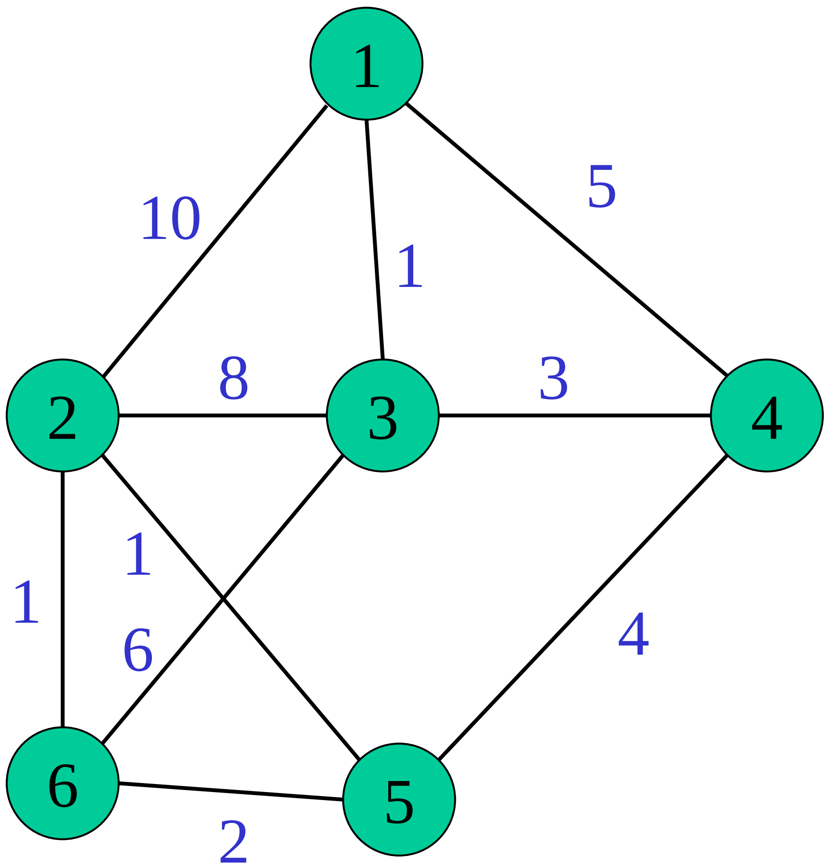


- *Minimum spanning tree*: the spanning tree with the least total edge cost

Minimum Spanning Tree (MST)

We are given a weighted, undirected graph $G = (V, E)$, with weight function $w: E \rightarrow \mathbf{R}$ mapping edges to real valued weights

Problem: Find the minimum cost spanning tree



Why minimum spanning trees?

- Lots of applications
- Minimize length of gas pipelines between cities
- Find cheapest way to wire a house (with minimum cable)
- Find a way to connect various routers on a network that minimizes total delay
- Etc…

Finding Min Spanning Trees

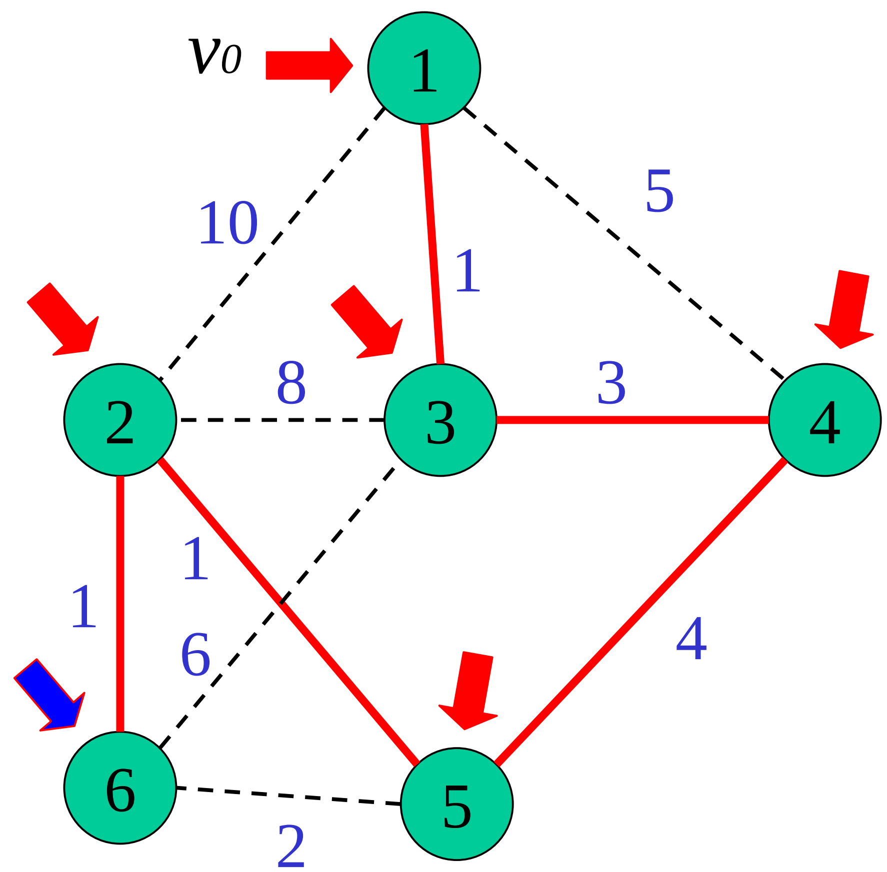
- For any spanning tree T, inserting an edge e_{new} not in T creates a cycle
  - › Removing any edge e_{old} from the cycle gives back a spanning tree
  - › If inserted edge e_{new} has a lower cost than removed edge e_{old}, we get a lower cost spanning tree
- Create a spanning tree as follows:
  - › Add an edge of minimum cost that doesn't create a cycle
  - › Repeat for $|V|$-1 edges
- Resulting spanning tree has minimum cost:
  - › if you could replace an edge with another edge of lower cost without creating a cycle, our algorithm would have picked it

Min Spanning Tree Algorithms

- Prim
  - › pick lowest cost edge *connected to known spanning tree* that doesn't create a cycle and expand to include it in the tree
- Kruskal
  - › pick lowest cost edge *not yet in a tree* that doesn't create a cycle and expand to include it somewhere in the forest

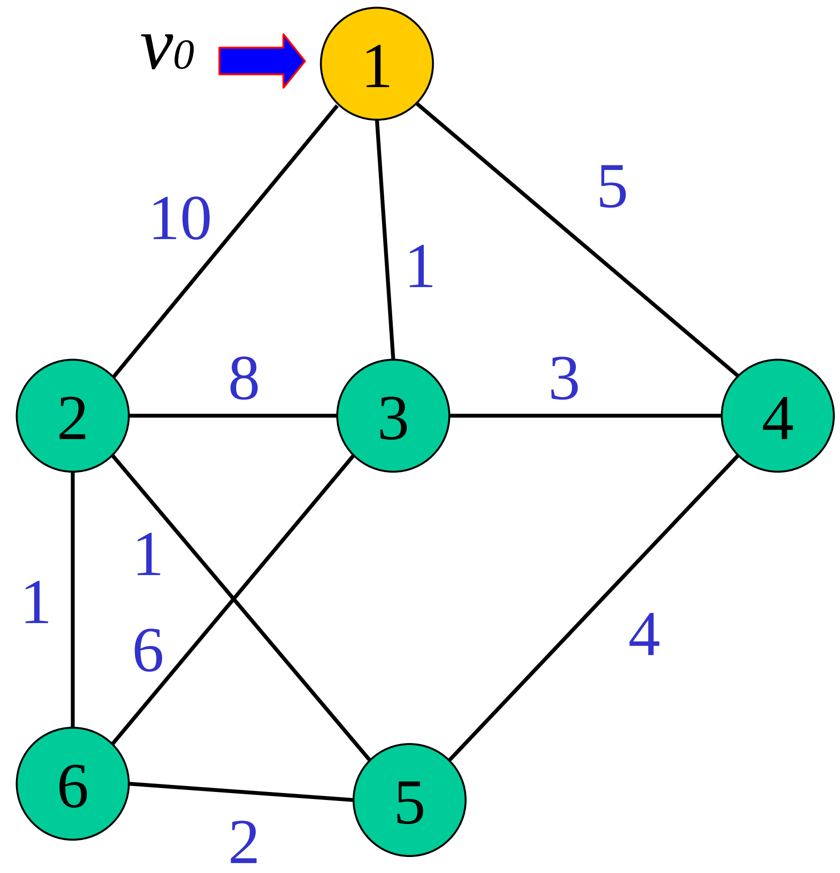
Prim's Algorithm for Finding the MST

- Starting from an empty tree, T, pick a vertex, v_0, at random and initialize: $S = \{v_0\}$ and $E = \{\}$
- Choose the vertex v not in S such that *edge weight from v to a vertex in S is minimal* (get greedy!)
- Add v to S and the edge to E if no cycle is created
- Repeat until all vertices have been added



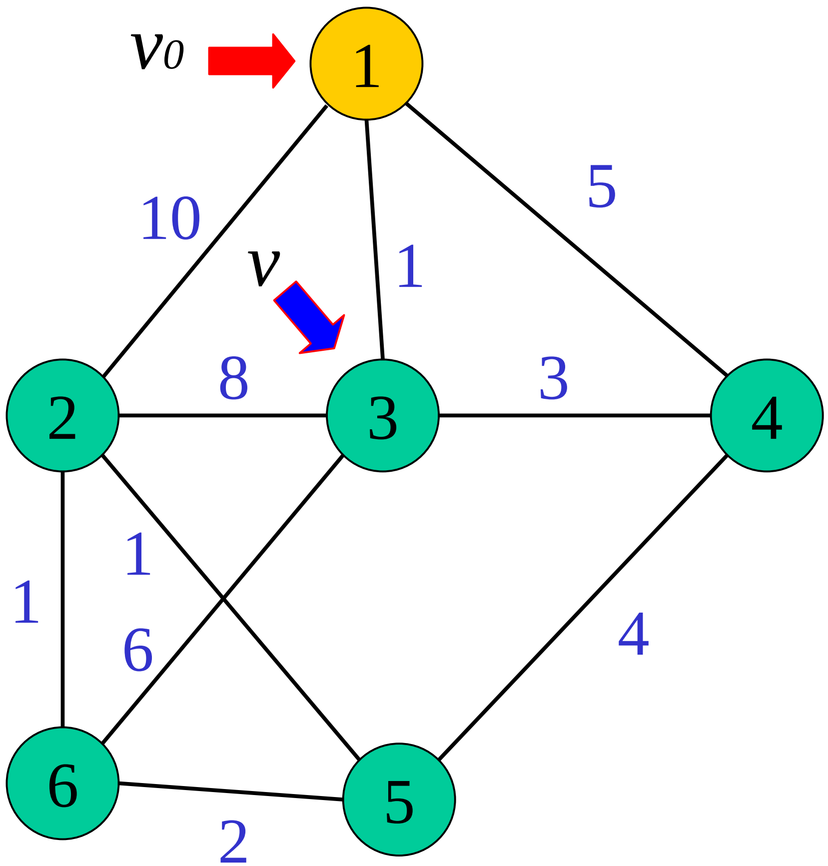
Prim's Algorithm for Finding the MST

- Starting from an empty tree, *T*, pick a vertex, v_0, at random and initialize: $S = \{v_0\}$ and $E = \{\}$



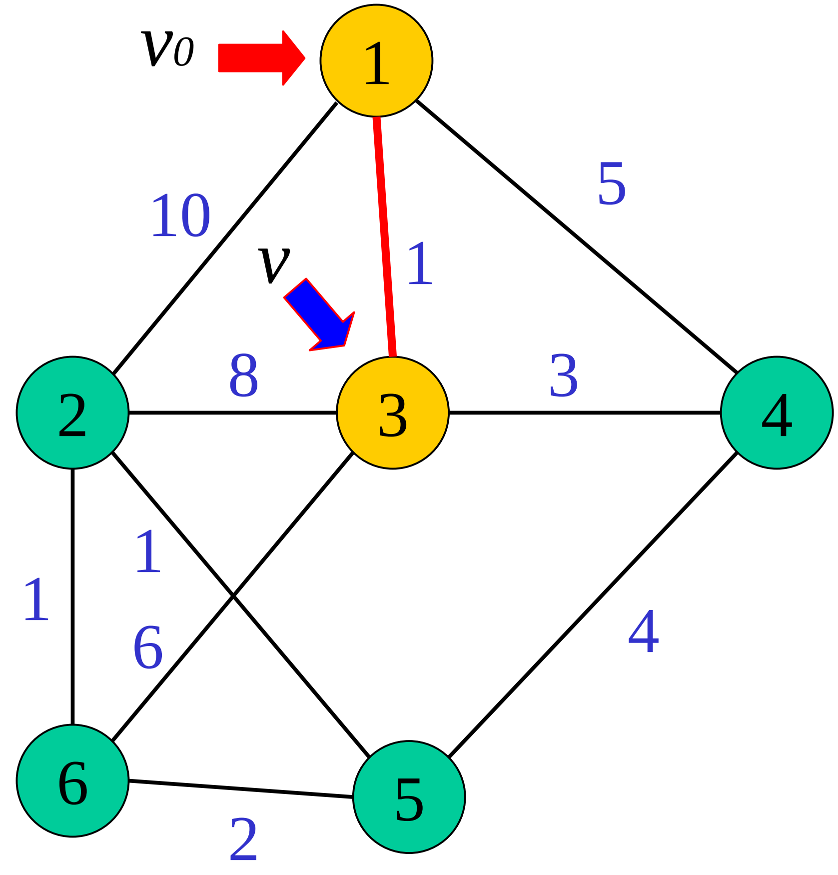
Prim's Algorithm for Finding the MST

- Starting from an empty tree, T, pick a vertex, v_0, at random and initialize: $S = \{v_0\}$ and $E = \{\}$
- Choose the vertex v not in S such that edge weight from v to a vertex in S is minimal (greedy algo)



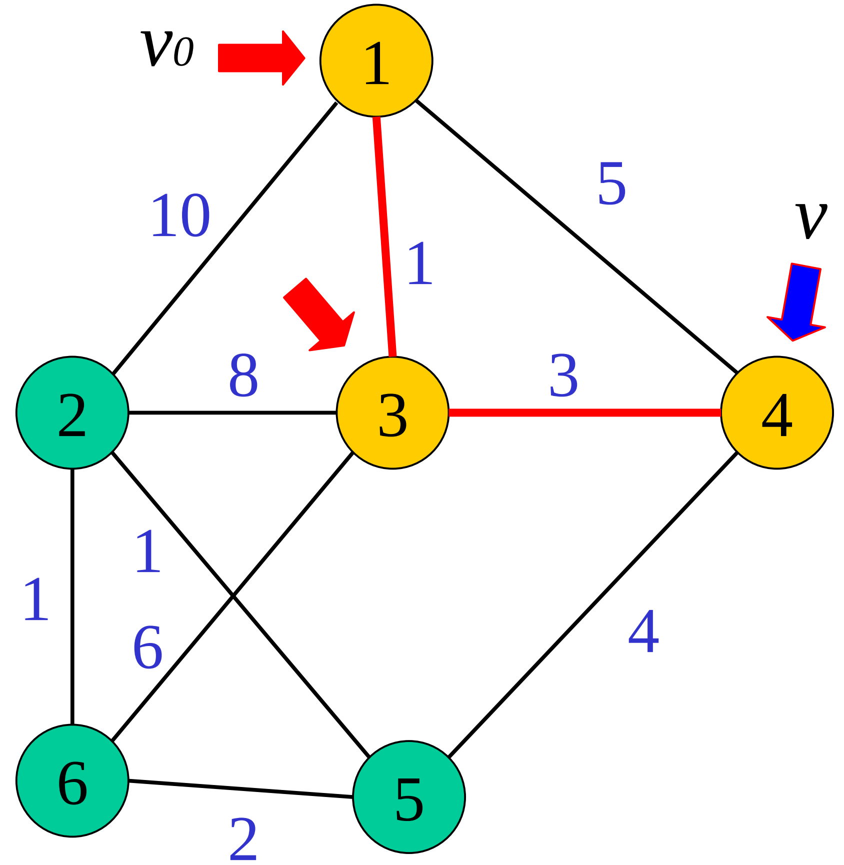
Prim's Algorithm for Finding the MST

- Starting from an empty tree, T, pick a vertex, v_0, at random and initialize:
  $S = \{v_0\}$ and $E = \{\}$
- Choose the vertex v not in S such that edge weight from v to a vertex in S is minimal
- Add v to S and the edge to E if no cycle is created



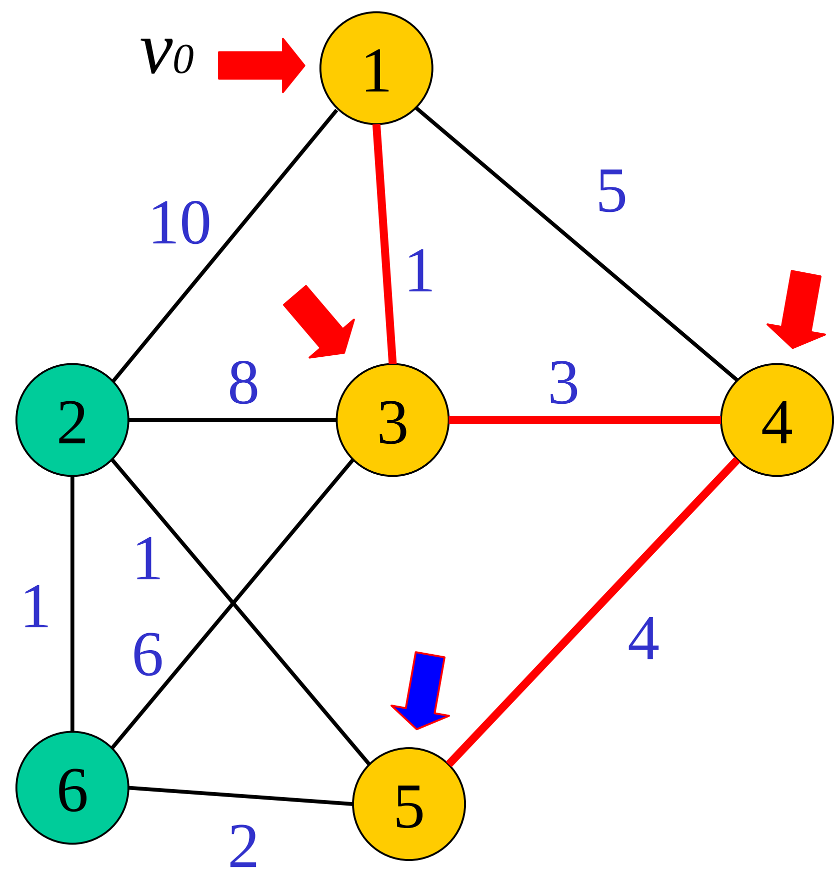
Prim's Algorithm for Finding the MST

- Starting from an empty tree, *T*, pick a vertex, v_0, at random and initialize:
  $S = \{v_0\}$ and $E = \{\}$
- Choose the vertex *v* not in *S* such that edge weight from *v* to a vertex in *S* is minimal
- Add *v* to *S* and the edge to *E* if no cycle is created
- Repeat until all vertices have been added



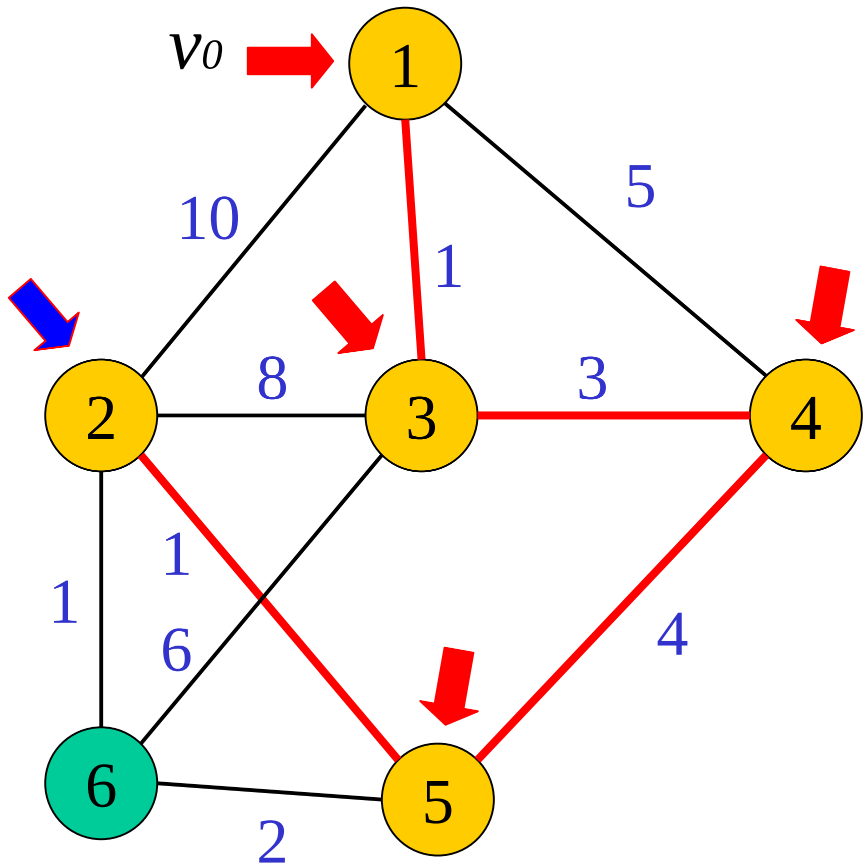
Prim's Algorithm for Finding the MST

- Starting from an empty tree, *T*, pick a vertex, v_0, at random and initialize: $S = \{v_0\}$ and $E = \{\}$
- Choose the vertex *v* not in *S* such that edge weight from *v* to a vertex in *S* is minimal
- Add *v* to *S* and the edge to *E* if no cycle is created
- Repeat until all vertices have been added



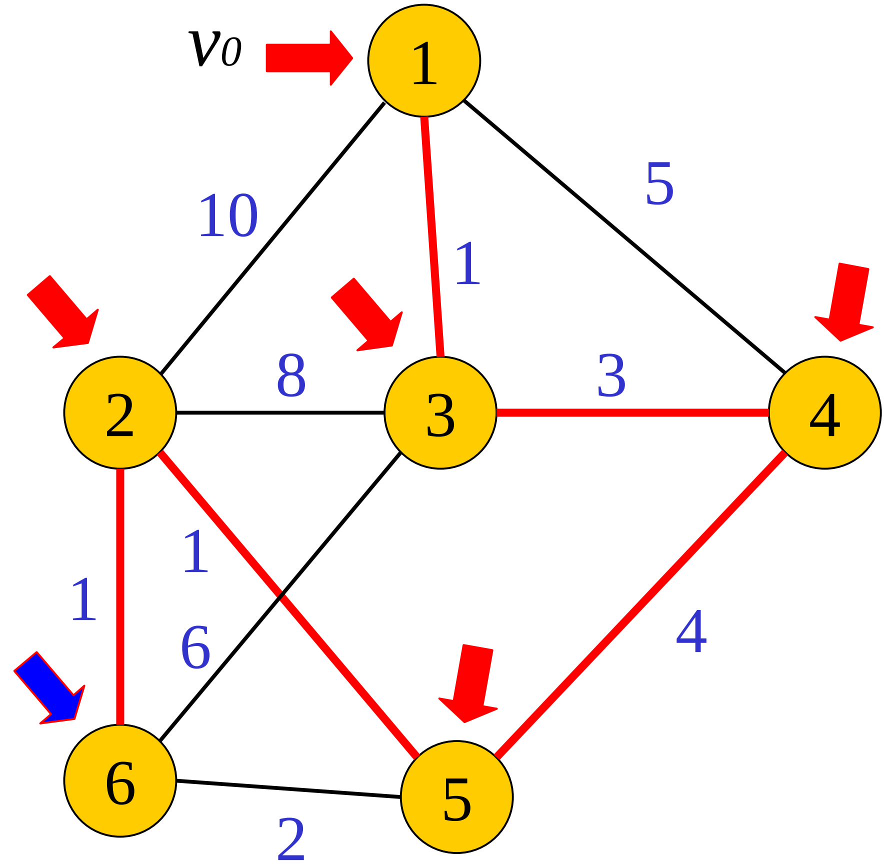
Prim's Algorithm for Finding the MST

- Starting from an empty tree, T, pick a vertex, v_0, at random and initialize:
  $S = \{v_0\}$ and $E = \{\}$
- Choose the vertex v not in S such that edge weight from v to a vertex in S is minimal
- Add v to S and the edge to E if no cycle is created
- Repeat until all vertices have been added



Prim's Algorithm for Finding the MST

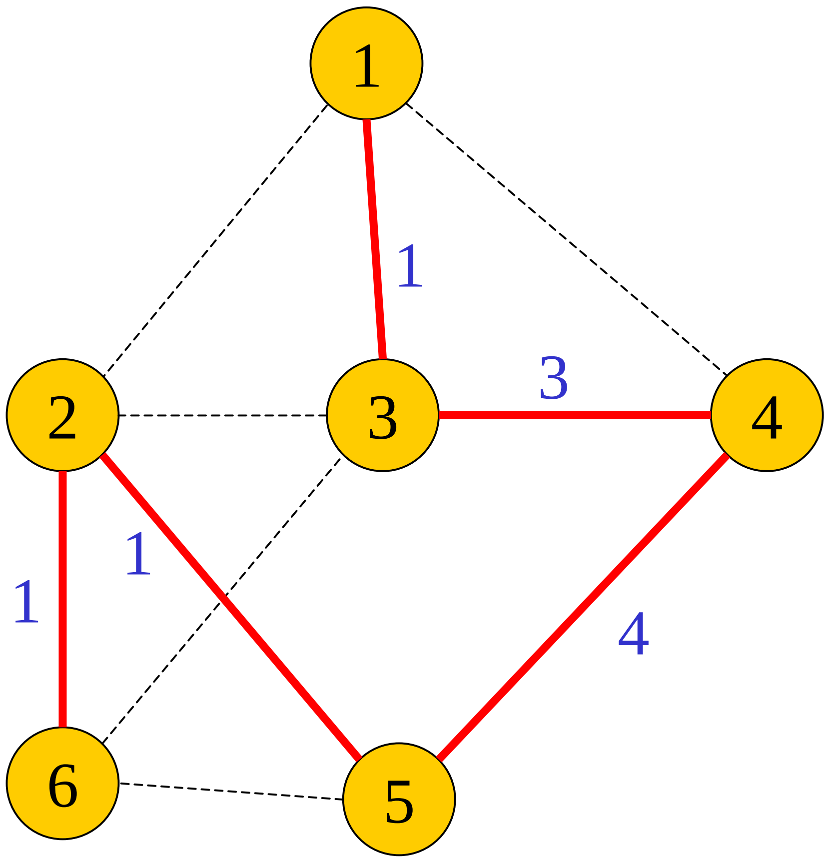
- Starting from an empty tree, *T*, pick a vertex, v_0, at random and initialize: $S = \{v_0\}$ and $E = \{\}$
- Choose the vertex *v* not in *S* such that edge weight from *v* to a vertex in *S* is minimal
- Add *v* to *S* and the edge to *E* if no cycle is created
- Repeat until all vertices have been added



Prim's Algorithm for Finding the MST

Done!

Total cost = 1 + 3 + 4 + 1 + 1
= 10



Prim’s Algorithm Analysis

```
Initialize connection cost of each node to ∞ and mark it unknown
Initialize connection cost of one selected node S to 0, with
    Prev[S] = 0
While there are unknown nodes left in the graph
       Select the unknown node N with the lowest connection cost
       Mark N as known
       For each unknown node A adjacent to N
               If cost of (N, A) < A’s cost
                      A’s cost = cost of (N, A)
                      Prev[A] = N  //store preceding node
```

- This is almost identical to Dijkstra’s algorithm
- Run time is $O(|V|^2)$ without heaps and $O(|V| \log |V| + |E| \log |V|)$ using binary heaps

Kruskal's Algorithm for Finding the MST

Select edges in order of increasing cost and accept an edge only if it does not cause a cycle

```
Put all the vertices into single node trees by themselves
Put all the edges in a priority queue with key = edge cost
Repeat until |V|-1 edges have been accepted {
    Extract cheapest edge from priority queue
    If it forms a cycle
      ignore it
    else
      accept the edge – it will join two existing trees yielding
      a larger tree and reducing the forest by one tree
}
Return the accepted edges (they form the spanning tree)
```

Reducing the forest to a single tree

- Initially, there are *n* different single vertex trees that partition the set of vertices
- After you have added some edges, you have fewer (but larger) trees, which together still partition the set of vertices

Detecting Cycles

- When do you get a cycle? If you add an edge (u,v) where both u and v are already in the same tree T_i, you get a cycle
  - › Therefore, to check for cycles, you only need to find out if u and v are in the same tree
  - › If not, then the edge can be added and we union vertices in u's tree with vertices in v's tree
- What is your favorite data structure for such operations?

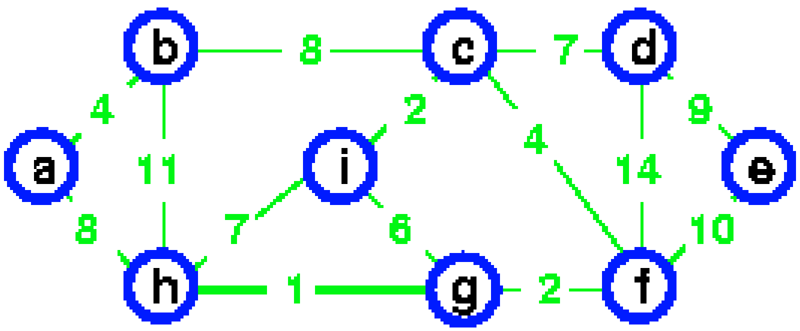
Kruskal's use of Disjoint Set ADT

- In Kruskal's algorithm, connected vertices form equivalence classes
  - each tree is a set of connected vertices
  - *being connected* is the equivalence relation
- Initially, each vertex is in a class by itself
- As edges are added, more vertices become related and the equivalence classes grow in size and are reduced in number
- Until finally all the vertices are in a single equivalence class

Kruskal’s use of Disjoint Set ADT

- Detecting cycles is easy!
- For each edge (u,v) that you’re thinking about adding
  - › If Find(u) == Find(v), then u and v are in the same class (same tree) and therefore the edge will form a cycle, so reject it
  - › Otherwise, we accept the edge and do Union(u,v), thereby indicating that all of the elements in the two trees are now in the same tree

Kruskal initilized

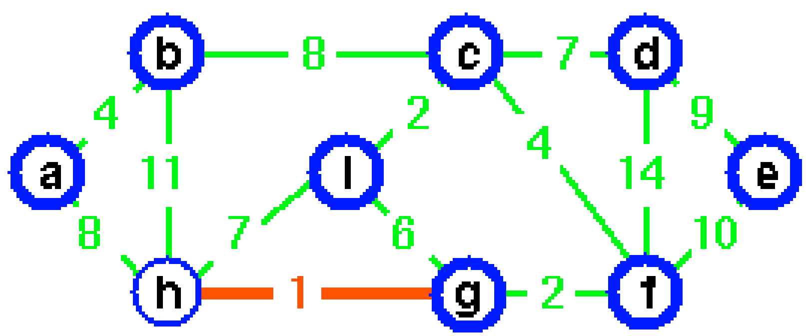


All the vertices are in a forest of single element trees.
All the vertices are in a set of single element equivalence classes.

V = {{a},{b},{c},{d},{e},{f},{g},{h},{i}}

Kruskal in action

The cheapest edge is h-g

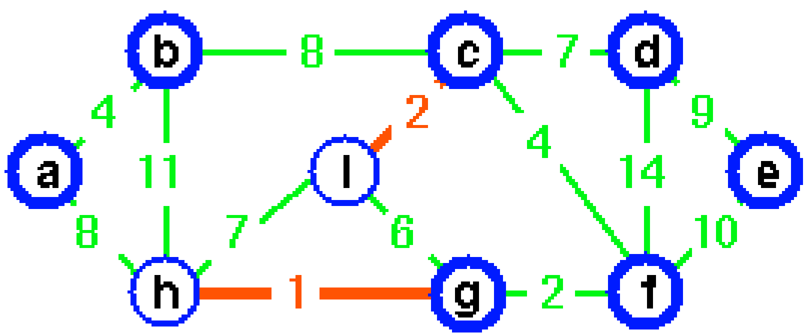


Join h and g into a 2-element tree.

V = {{a},{b},{c},{d},{e},{f},{g,h},{i}}

Kruskal in action

The next cheapest edge is c-i

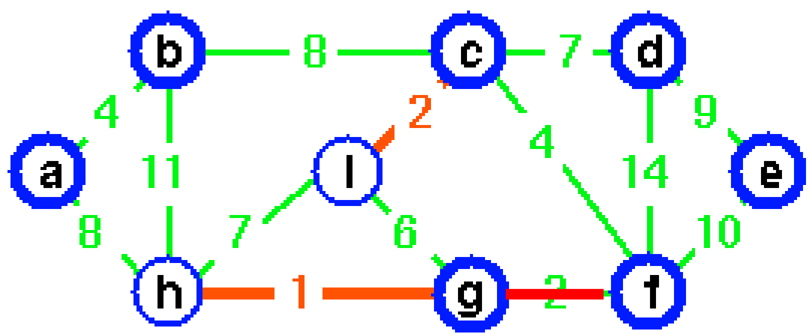


Join c and i into a 2-element tree

V = {{a},{b},{c,i},{d},{e},{f},{g,h}}

Kruskal in action

The next cheapest edge is g-f

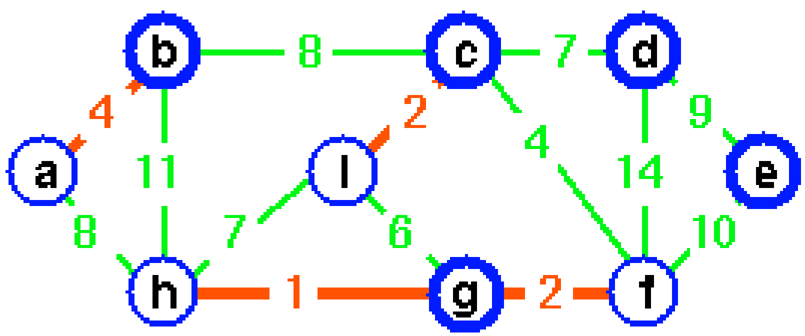


Join g tree and f into a 3-element tree

V = {{a},{b},{c,i},{d},{e},{g,f,h}}

Kruskal in action

The next cheapest edge is a-b

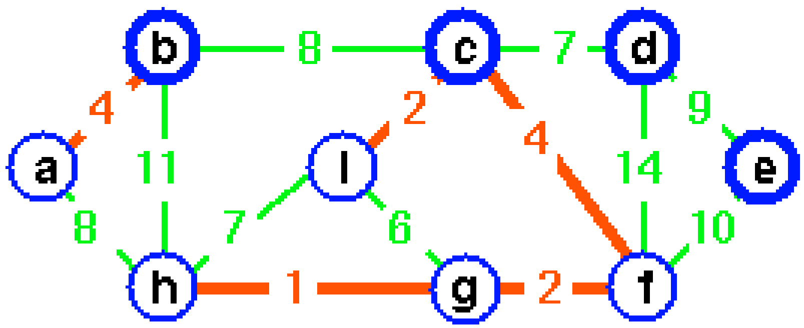


Join a and b into a 2-element tree

V = {{a,b},{c,i},{d},{e},{g,f,h}}

Kruskal in action

The next cheapest edge is c-f

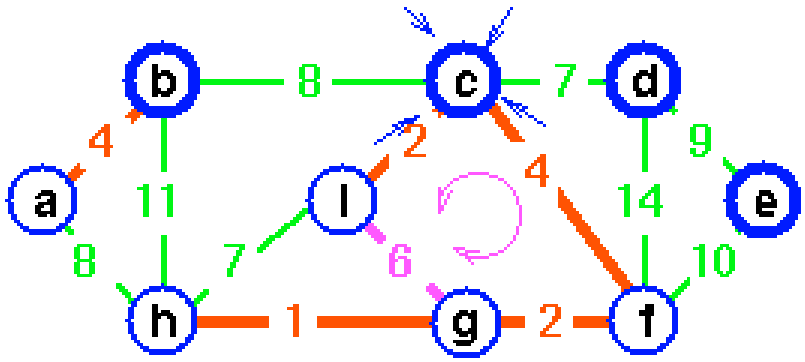


Join c and f trees into one 5-element tree

V = {{a,b},{c,f,g,h,i},{d},{e}}

Kruskal in action

The next cheapest edge is g-i



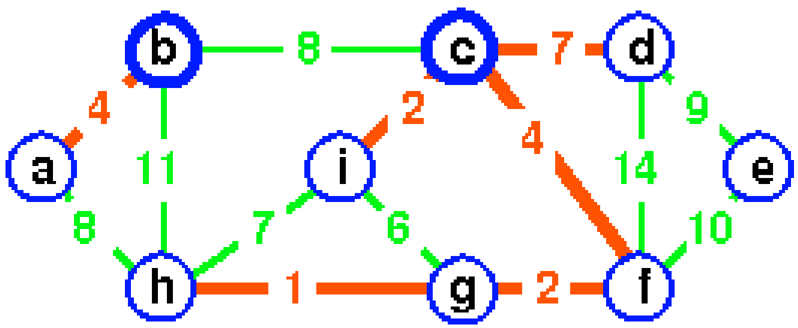
Find(g) is c
Find(i) is also c

g-i forms a cycle. Ignore this edge.

V = {{a,b},{c,f,g,h,i},{d},{e}}

Kruskal in action

The next cheapest edge is c-d

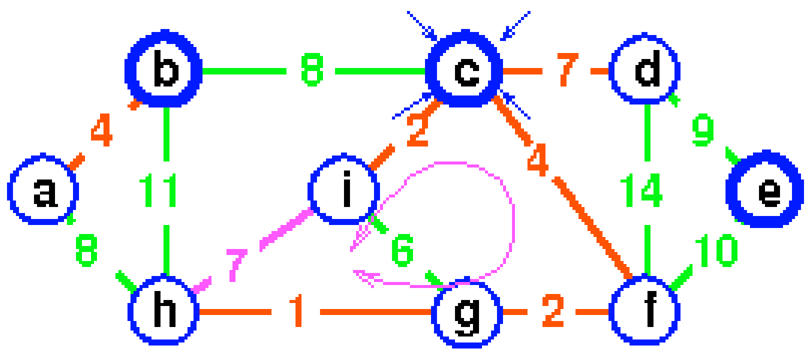


Join c tree and d into one 6-element tree

V = {{a,b},{c,d,f,g,h,i},{e}}

Kruskal in action

The next cheapest edge is h-i



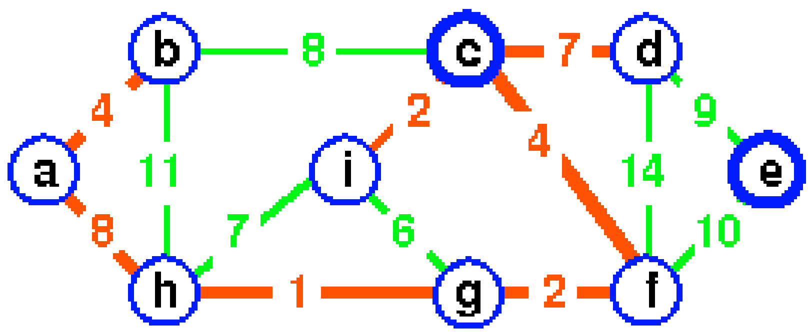
Find(h) is c
Find(i) is c

h-i forms a cycle. Ignore this edge.

V = {{a,b},{c,d,f,g,h,i},{e}}

Kruskal in action

The next cheapest edge is a-h



Join a and h trees into one tree

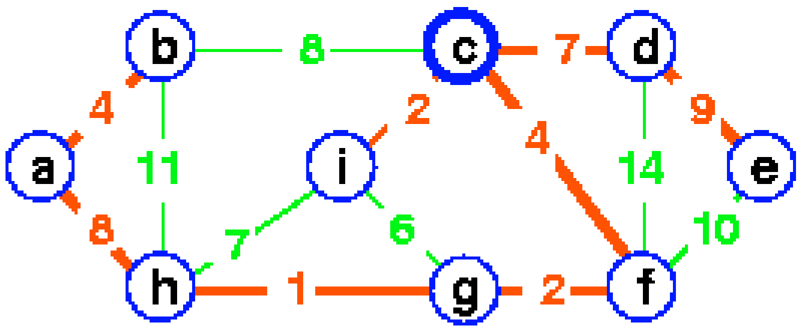
V = {{a,b,c,d,f,g,h,i},{e}}

Kruskal done!

The next cheapest edge is b-c

The next cheapest edge is d-e

Find(b) is c
Find(c) is c



b-c forms a cycle. Ignore this edge.

Join c tree and e into one tree

V = {{a,b,c,d,e,f,g,h,i}}

Kruskal's Algorithm for Finding the MST

Select edges in order of increasing cost and accept an edge only if it does not cause a cycle

```
Put all the vertices into single node trees by themselves
Put all the edges in a priority queue with key = edge cost
Repeat until |V|-1 edges have been accepted {
    Extract cheapest edge from priority queue
    If it forms a cycle
      ignore it
    else
      accept the edge – it will join two existing trees yielding
      a larger tree and reducing the forest by one tree
}
Return the accepted edges (they form the spanning tree)
```

$O(|V|)$

$O(|E|)$

$O(|E|)$

$O(\log |E|)$

Worst case requires |E| DeleteMin operations

total worst case running time is $O(|E| \cdot \log |E|)$

Kruskal versus Prim

- Worst case running time
  - Prim: $O(|V| \log |V| + |E| \log |V|)$
  - Kruskal: $O(|E| \log |E|) = O(|E| \log |V|)$ since $|E| = O(|V|^2)$
- Kruskal usually runs much faster than $O(|E| \log |V|)$ in practice
  - Not all edges need to be DeleteMin-ed typically
  - The required $|V|$-1 edges are usually found quickly
  - So, Kruskal tends to be faster than Prim