# Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

# Minimum Spanning Tree
# Shortest Path
# Topological Sorting
# P vs. NP

# Minimum Spanning Tree

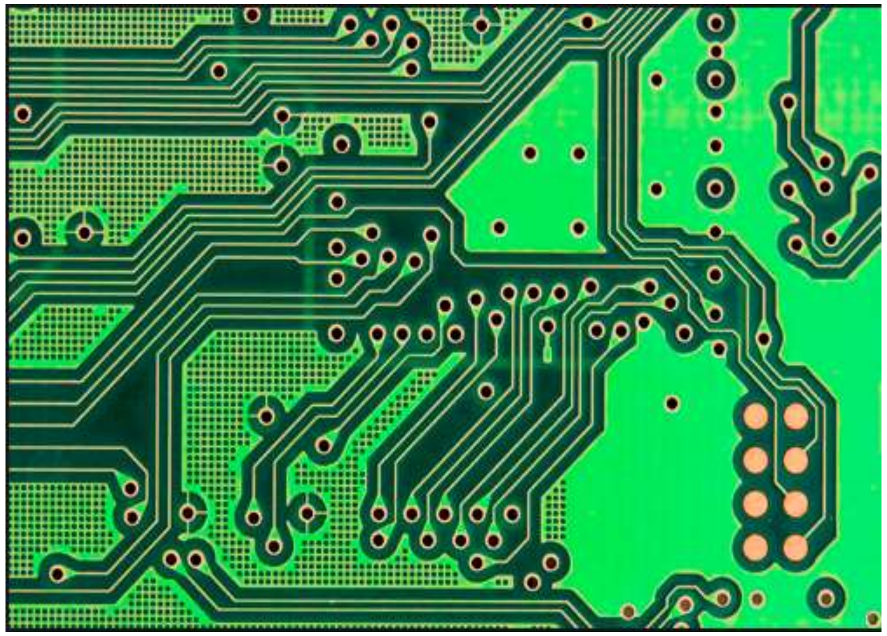
# Minimum Spanning Tree

- Suppose you have designed a printed circuit board

- You want to make sure that you have used the minimum number of traces

  - no extra connections between pins; would take up extra room

- How can you find these extra traces, if any?



# Minimum Spanning Tree

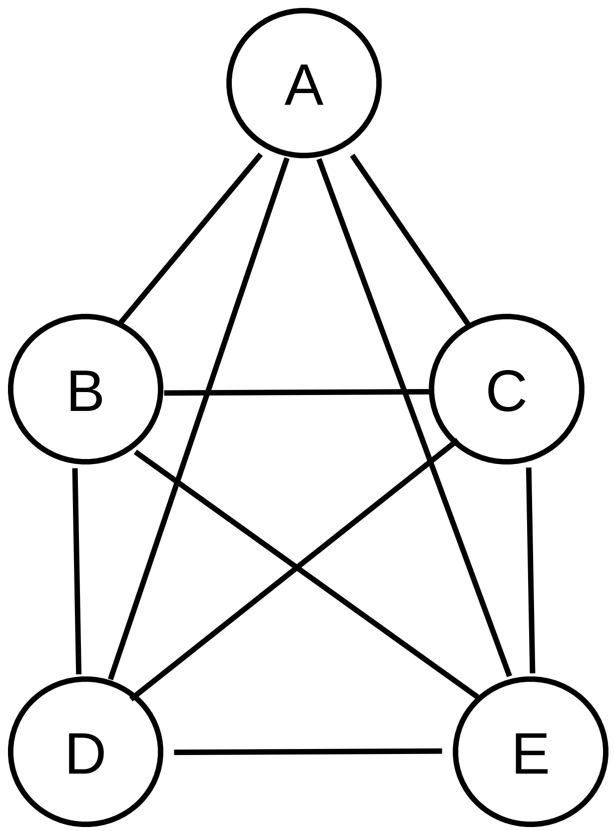
- The trick is to imagine this circuit as a graph

  Pins & Traces -> Vertices & Edges

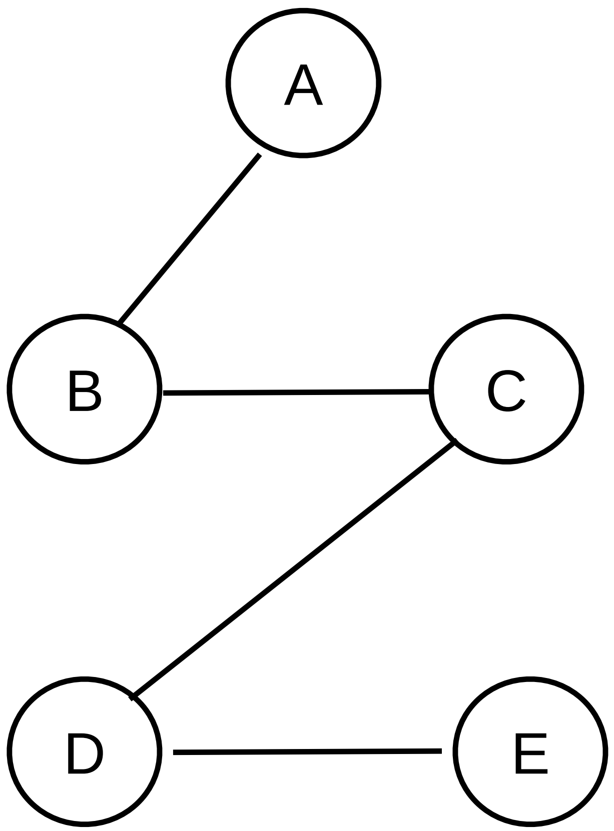
- Reduce this graph such that it has the same vertices with the minimum number of edges to connect them

  **Minimum Spanning Tree**

# Minimum Spanning Tree




Extra Edges




Minimum Number of Edges

Remember: There are many possible minimum spanning trees for a given set of vertices

Notice that number of edges in MST is one less than the number of vertices!

# Minimum Spanning Tree

- For unweighted graphs, every spanning tree is the minimum spanning tree

- DFS and BFS can be used

- Execute, and record the edges travelled

# Minimum Spanning Tree

```
while( !theStack.isEmpty() )        // until stack empty
   {                                // get stack top
   int currentVertex = theStack.peek();
   // get next unvisited neighbor
   int v = getAdjUnvisitedVertex(currentVertex);
   if(v == -1)                      // if no more neighbors
      theStack.pop();               //    pop it away
   else                             // got a neighbor
      {
      vertexList[v].wasVisited = true;  // mark it
      theStack.push(v);                 // push it
                                    // display edge
      displayVertex(currentVertex);     // from currentV
      displayVertex(v);                 // to v
      System.out.print(" ");
      }
   }  // end while(stack not empty)
```

Displaying both the current vertex and its next visited vertex
The two vertices define the edge

# Minimum Spanning Tree

- We can find MST of a graph only if it is connected.
- Otherwise, we can find a union of MSTs for each of its connected component – *Minimum Spanning Forest*

# Minimum Spanning Trees (MST) with Weighted Graphs

# Weighted Graphs

- A graphs where each edge has a weight
- For example, in a weighted graph of cities, a weight could represent the
  - Distance between cities
  - Cost to fly between cities
  - Number of automobile trips made annually between them

# MST with Weighted Graphs

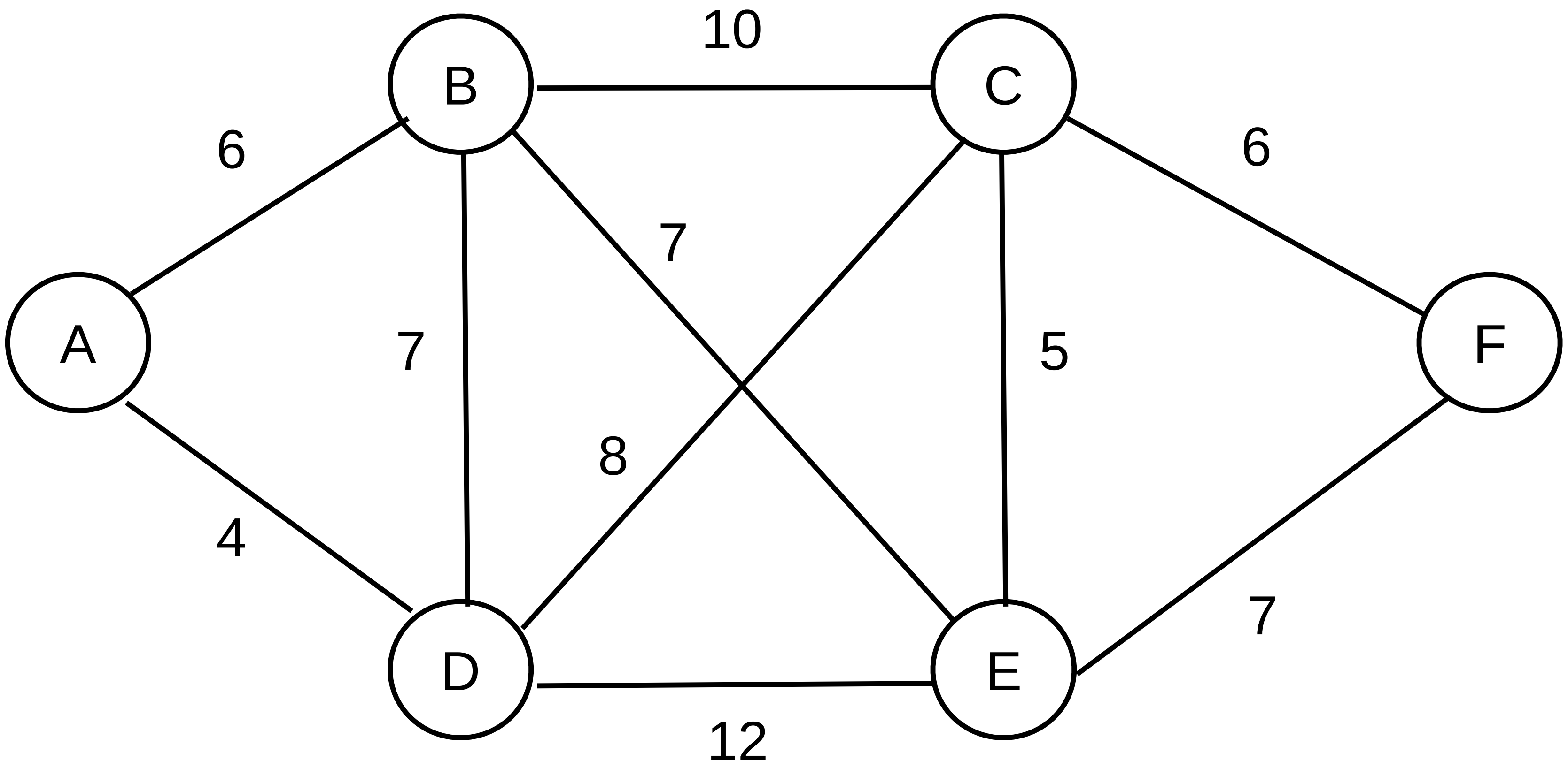
- Creating such a tree is a bit more difficult with weighted graphs

- Let’s try to understand the process with the help of an example

# MST with Weighted Graphs

- We want to install a cable television line that connects six cities
- We need five links (n = 6), but which five links would those be
- Let's say we are given the following information

# MST with Weighted Graphs



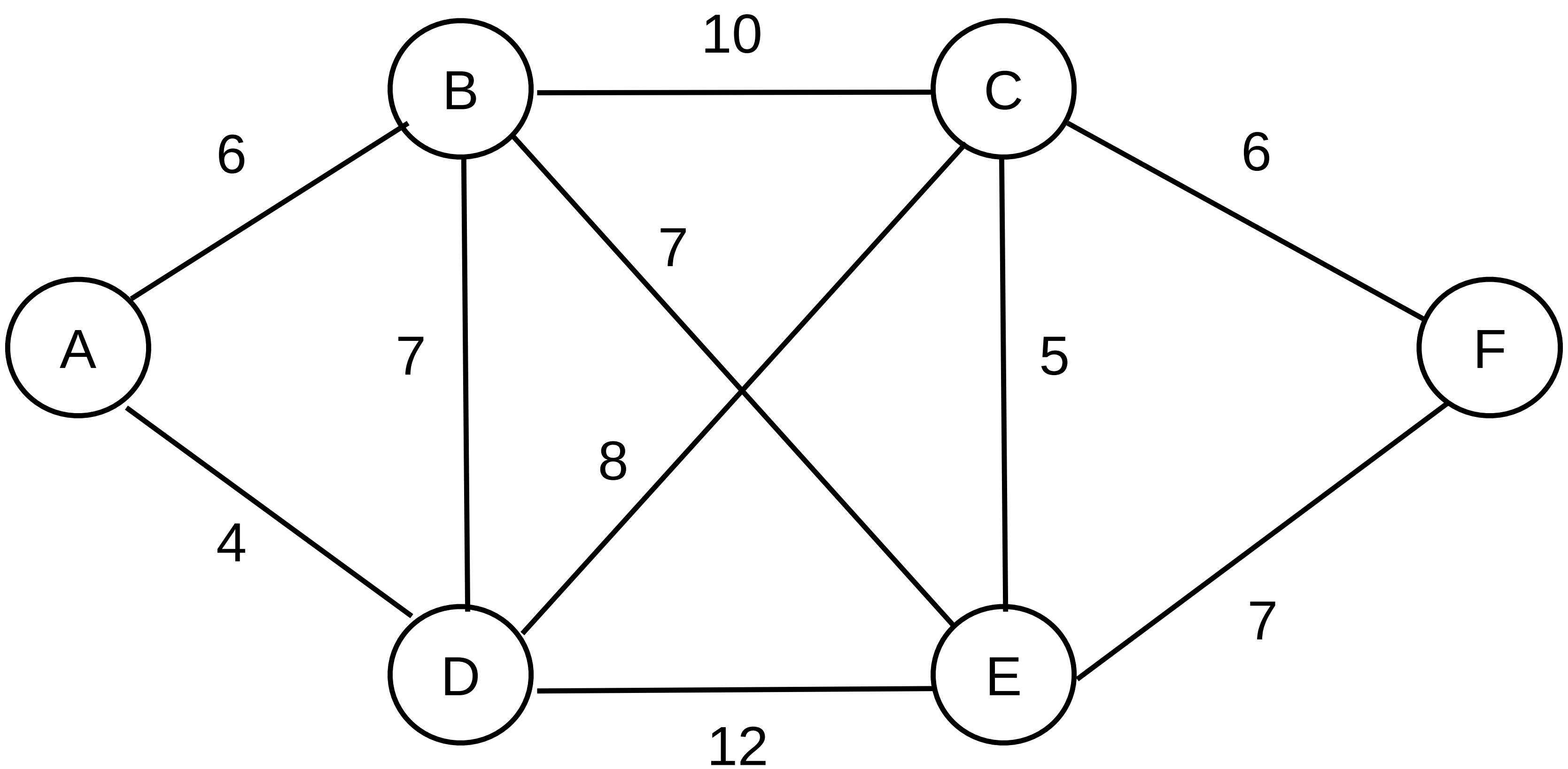

Weight: cost of installing cable
Some links are missing; let’s assume that they are just too expensive that there is no point in considering them
But the algorithm will work fine even if they were present.

# MST with Weighted Graphs

- Given this information, we need to generate the minimum spanning tree for this graph

- Let’s see how to do this.

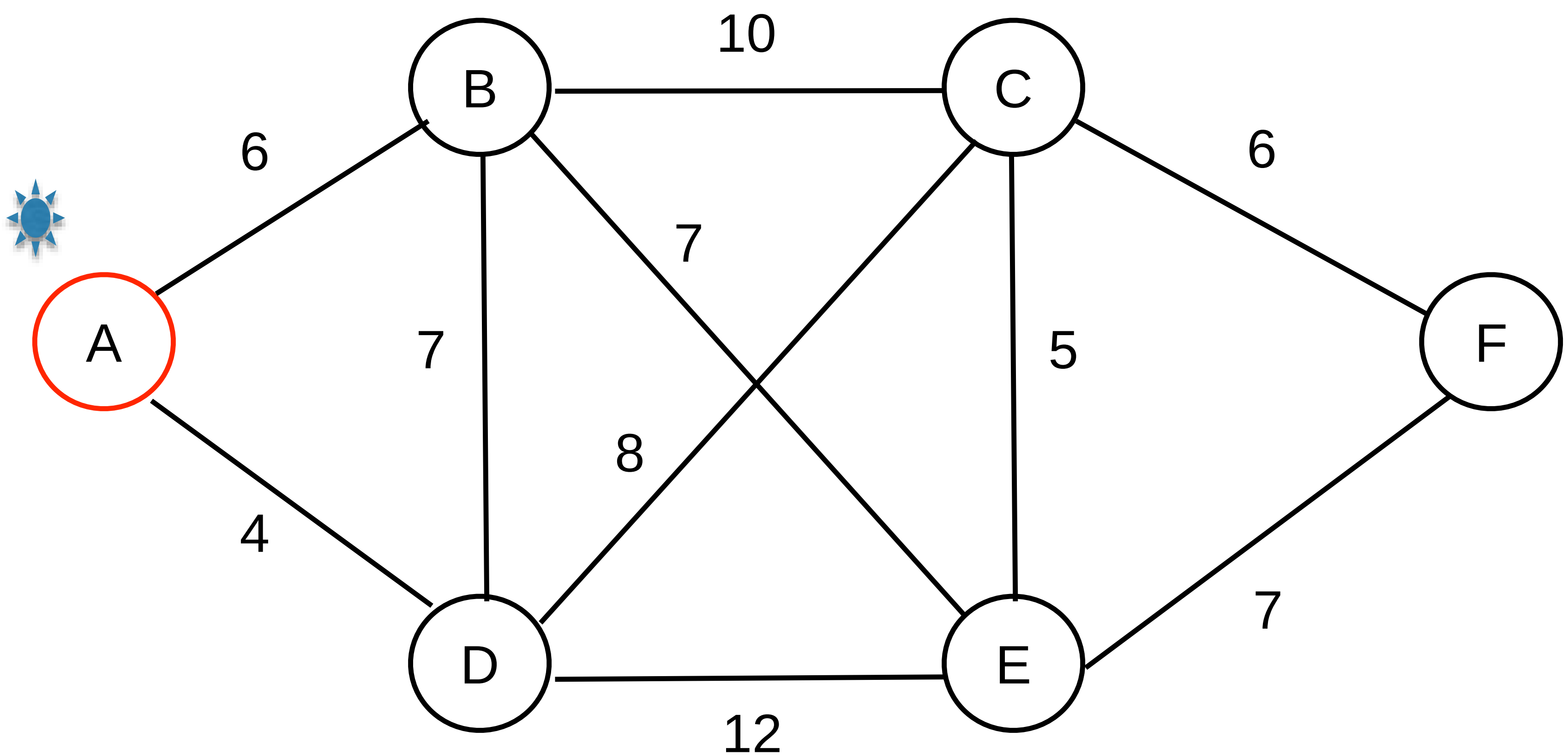
# MST with Weighted Graphs



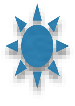

| Edge | Weight |
|---|---|
| | |
| | |
| | |
| | |

- Start with any city; Let’s pick A
- Create an office in A
- Measure the weight of the adjacent edges and insert them in the list on the right

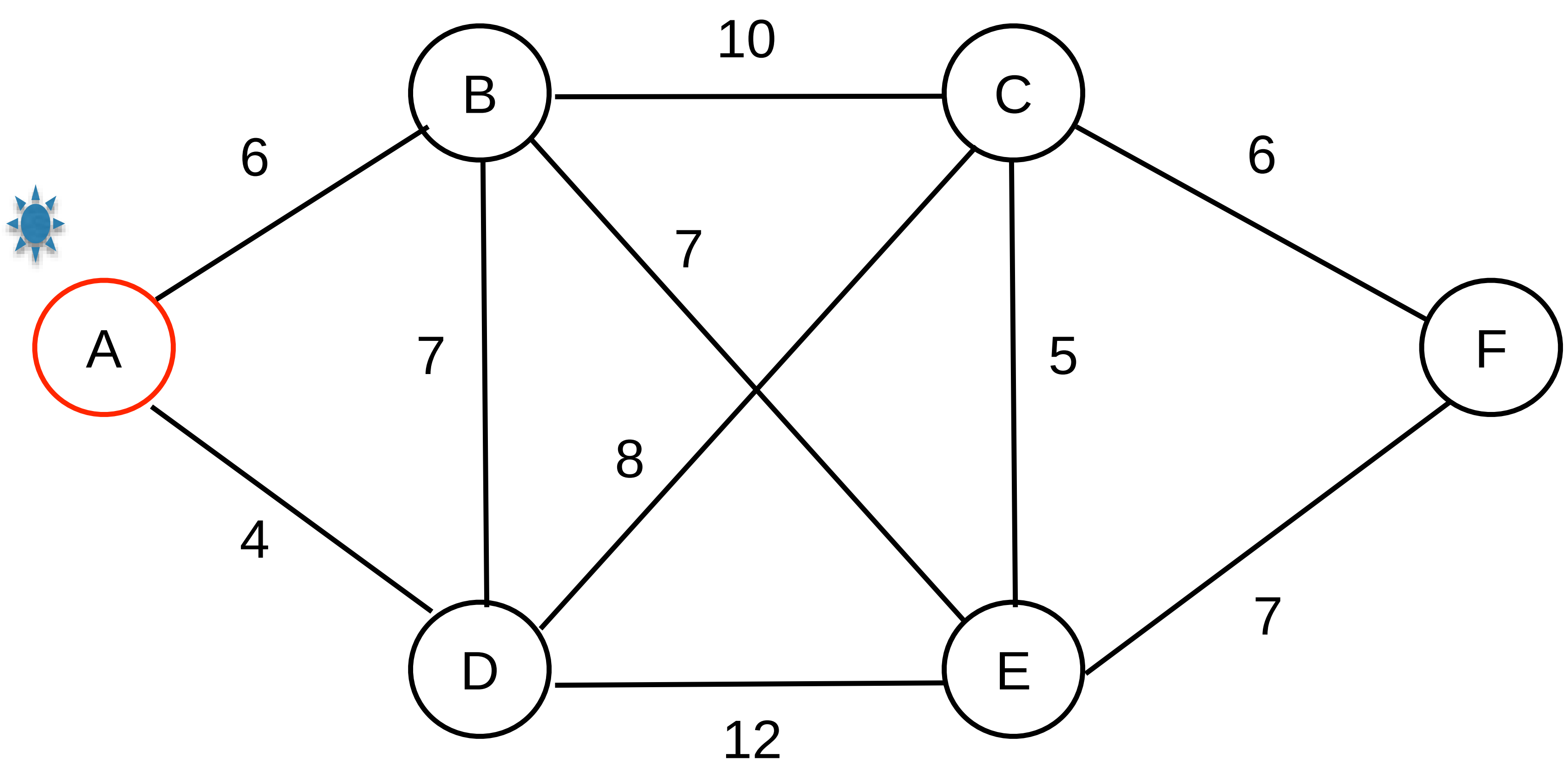
# MST with Weighted Graphs




| Edge | Weight |
|---|---|
| AB | 6 |
| AD | 4 |
| | |
| | |

 Indicates where a new office has just been built!

# MST with Weighted Graphs




| Edge | Weight |
|---|---|
| AB | 6 |
| **AD** | **4** |
| | |
| | |

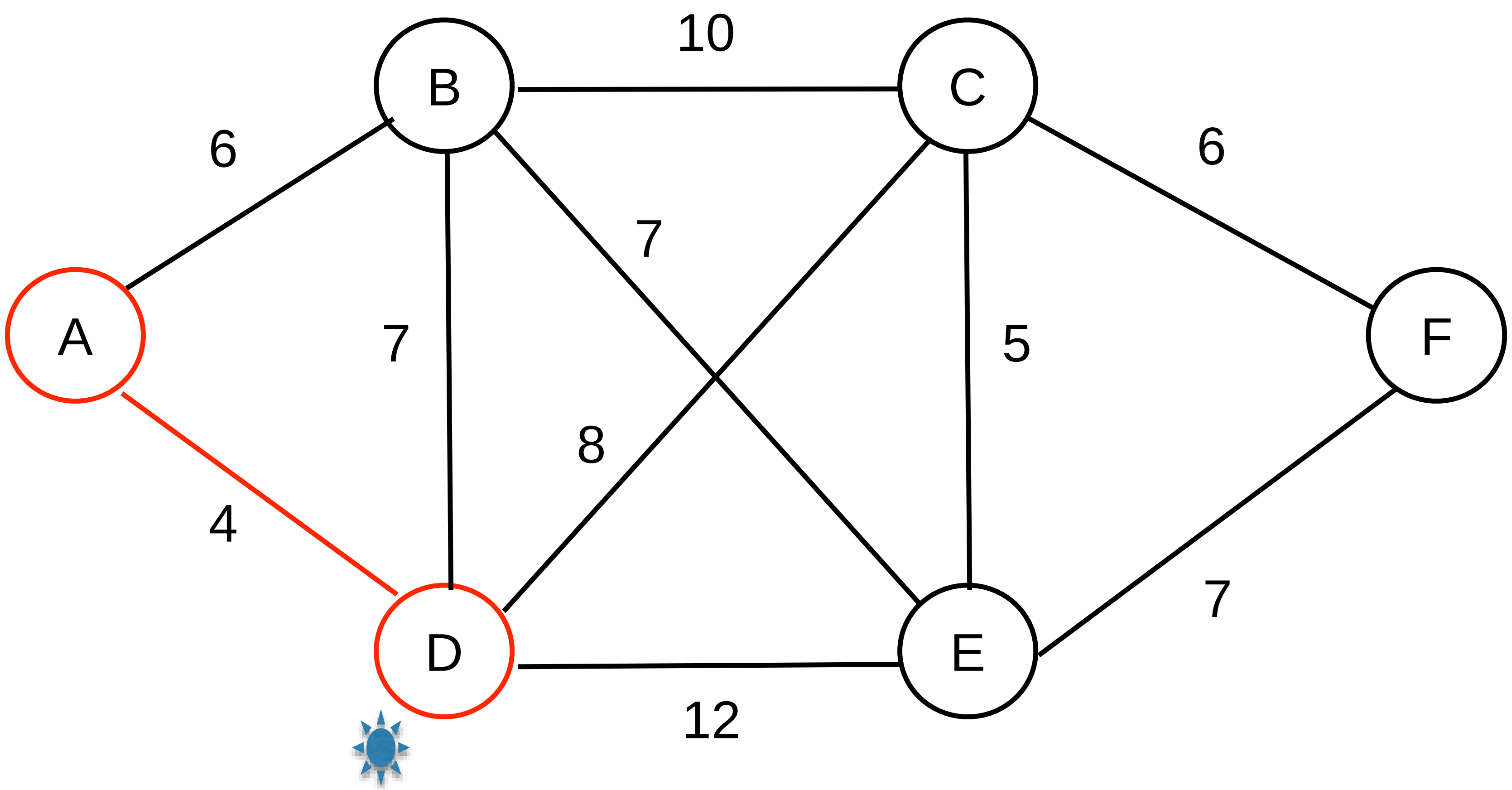
- From the list, pick the cheapest link
- Install it
- Remove it from the list
- Create an office in the new city.

Greedy Algorithm

# Greedy Algorithms

- Try to find solutions to problems step-by-step
  - A partial solution is incrementally expanded towards complete solution
  - In each step, there are several ways to expand the partial solution:
  - The best alternative for the moment is chosen, the others are discarded
- At each step the choice must be **locally optimal** – this is the central point of this technique

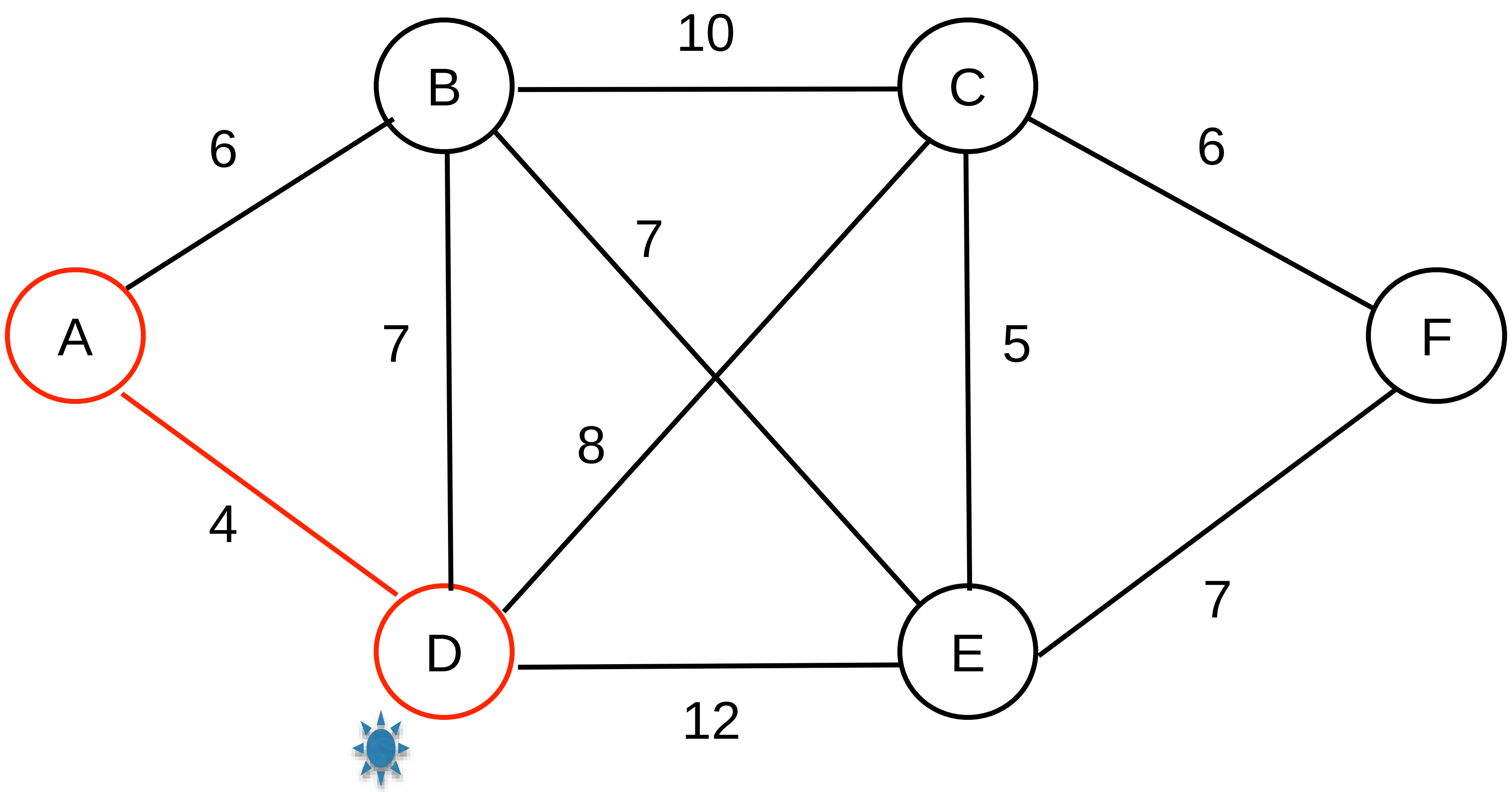
# MST with Weighted Graphs




| Edge | Weight |
|---|---|
| AB | 6 |
| | |
| | |
| | |

- Now we have offices at A, and D
- And one link AD
- D is the newest office

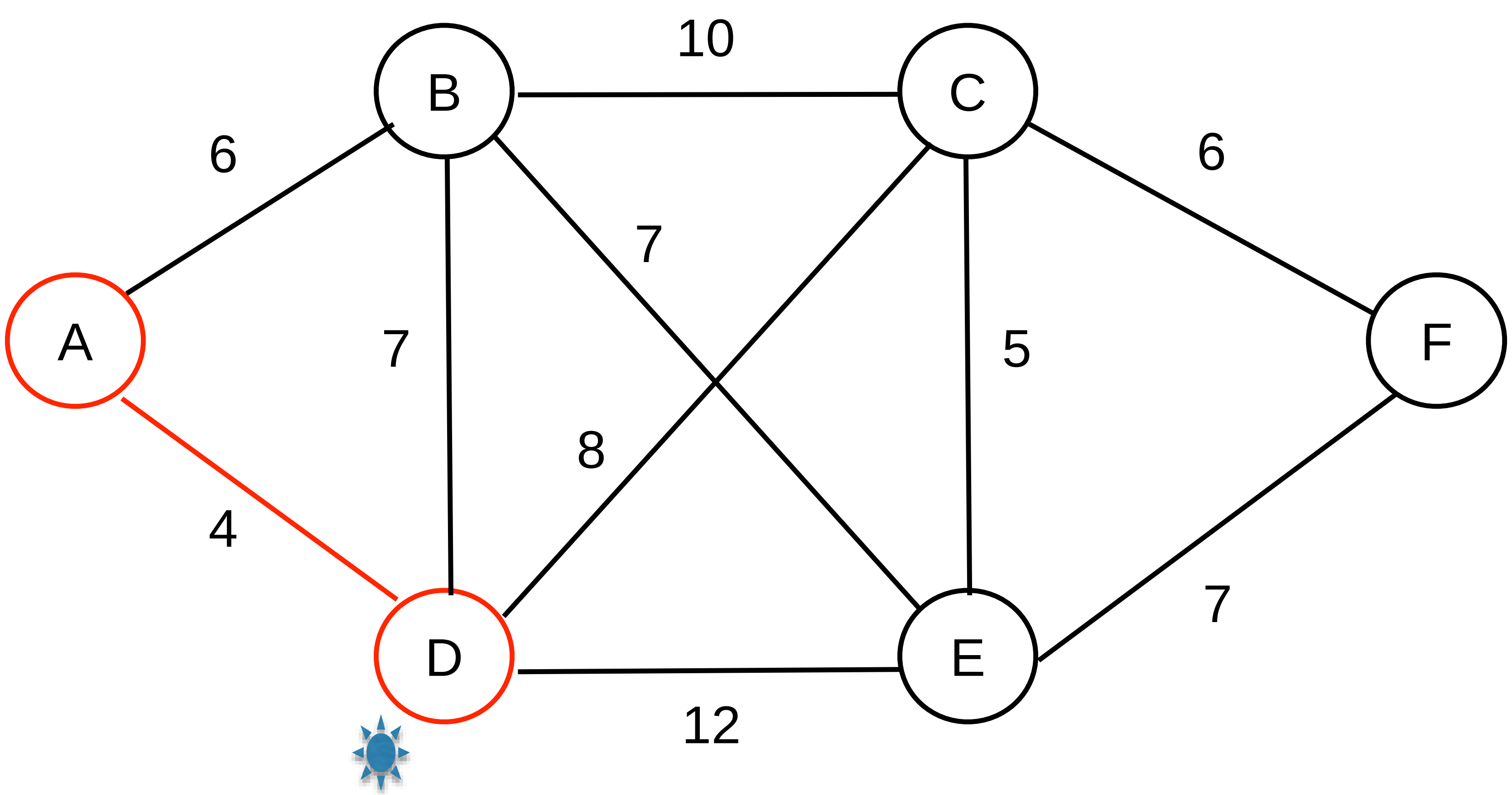
# MST with Weighted Graphs




| Edge | Weight |
|---|---|
| AB | 6 |
| | |
| | |
| | |

- Repeat the process for D.

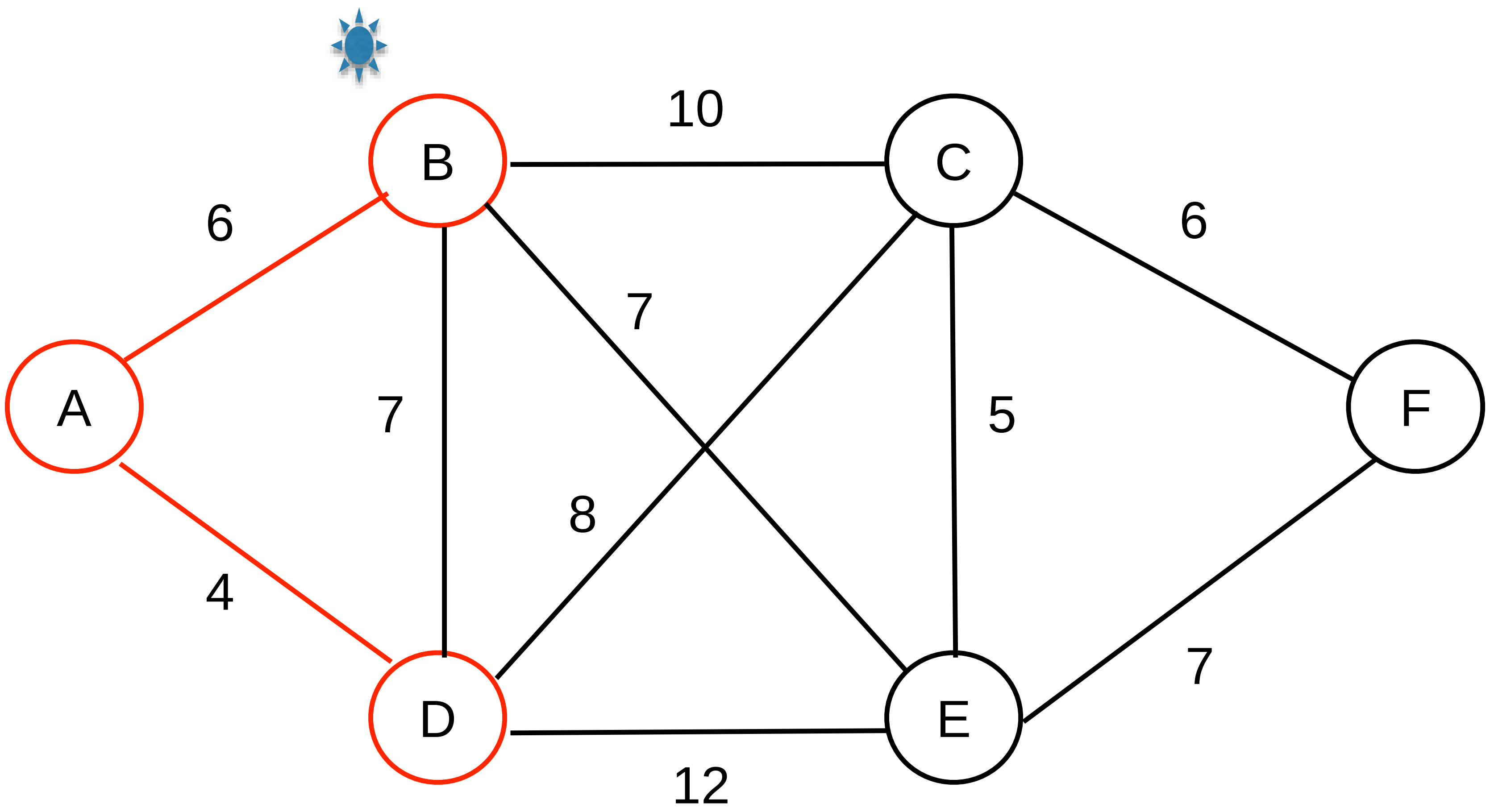
# MST with Weighted Graphs




| Edge | Weight |
|---|---|
| **AB** | **6** |
| DB | 7 |
| DC | 8 |
| DE | 12 |

- Again, choose the cheapest from the list.
- Yeah, you got it, we always choose the cheapest from the list

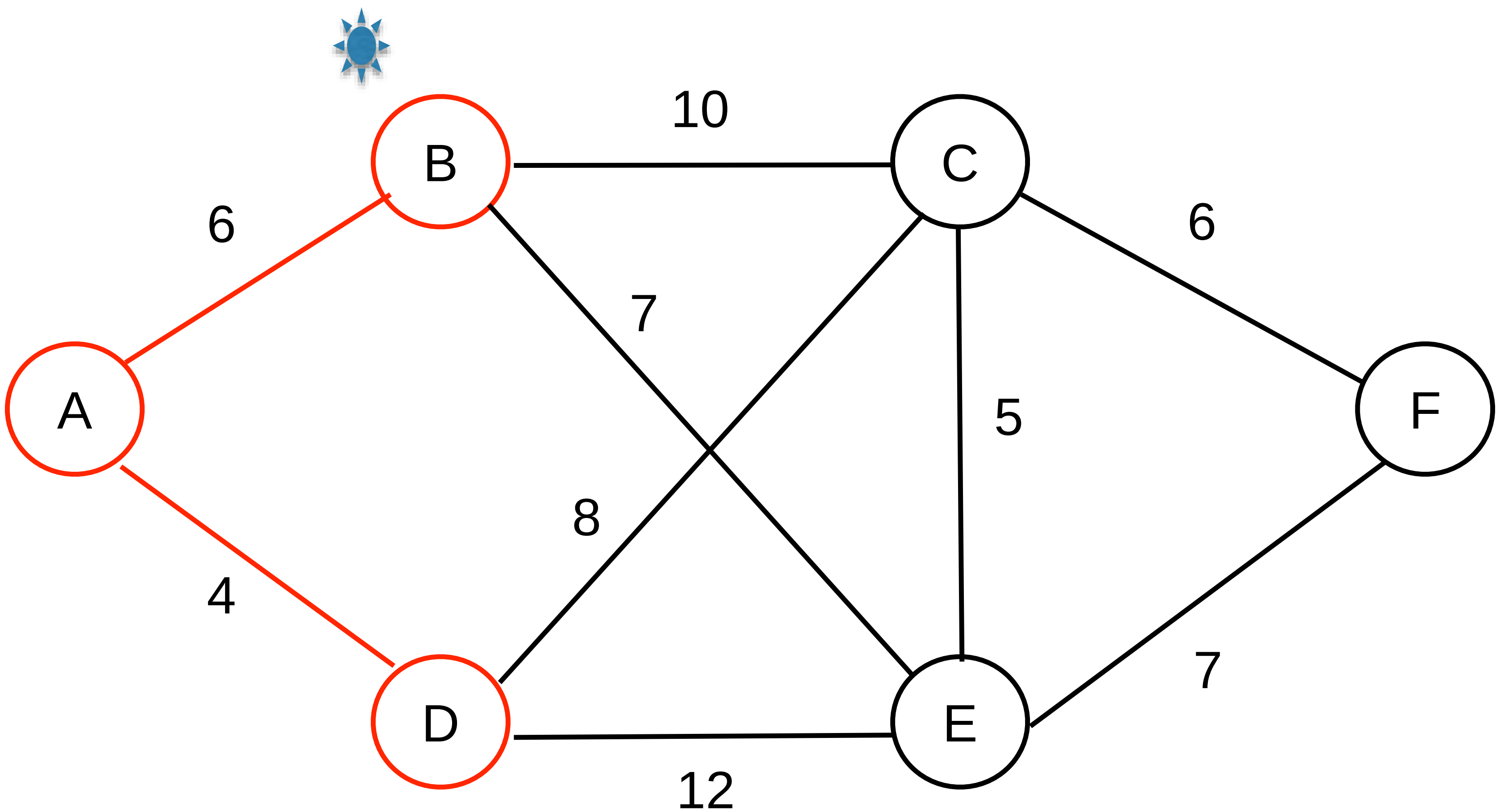
# MST with Weighted Graphs




| Edge | Weight |
|---|---|
| DB | 7 |
| DC | 8 |
| DE | 12 |

- Note:
  - DB is redundant, that is, B already has an office
  - Thus we will remove it from the list, too!
  - Remove from the list:
    - The one that is the cheapest
    - And those that are redundant

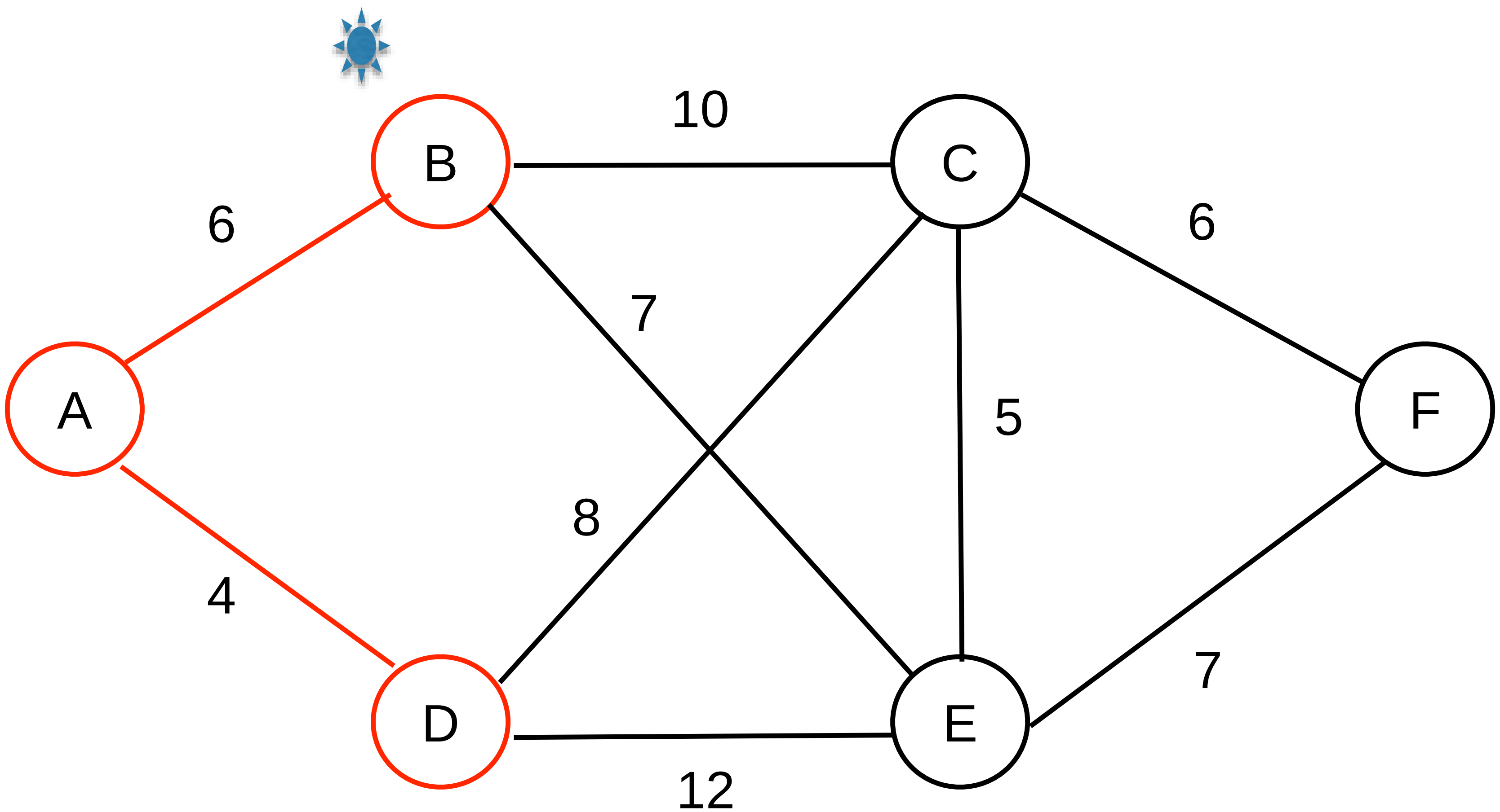
# MST with Weighted Graphs




| Edge | Weight |
|---|---|
| DC | 8 |
| DE | 12 |

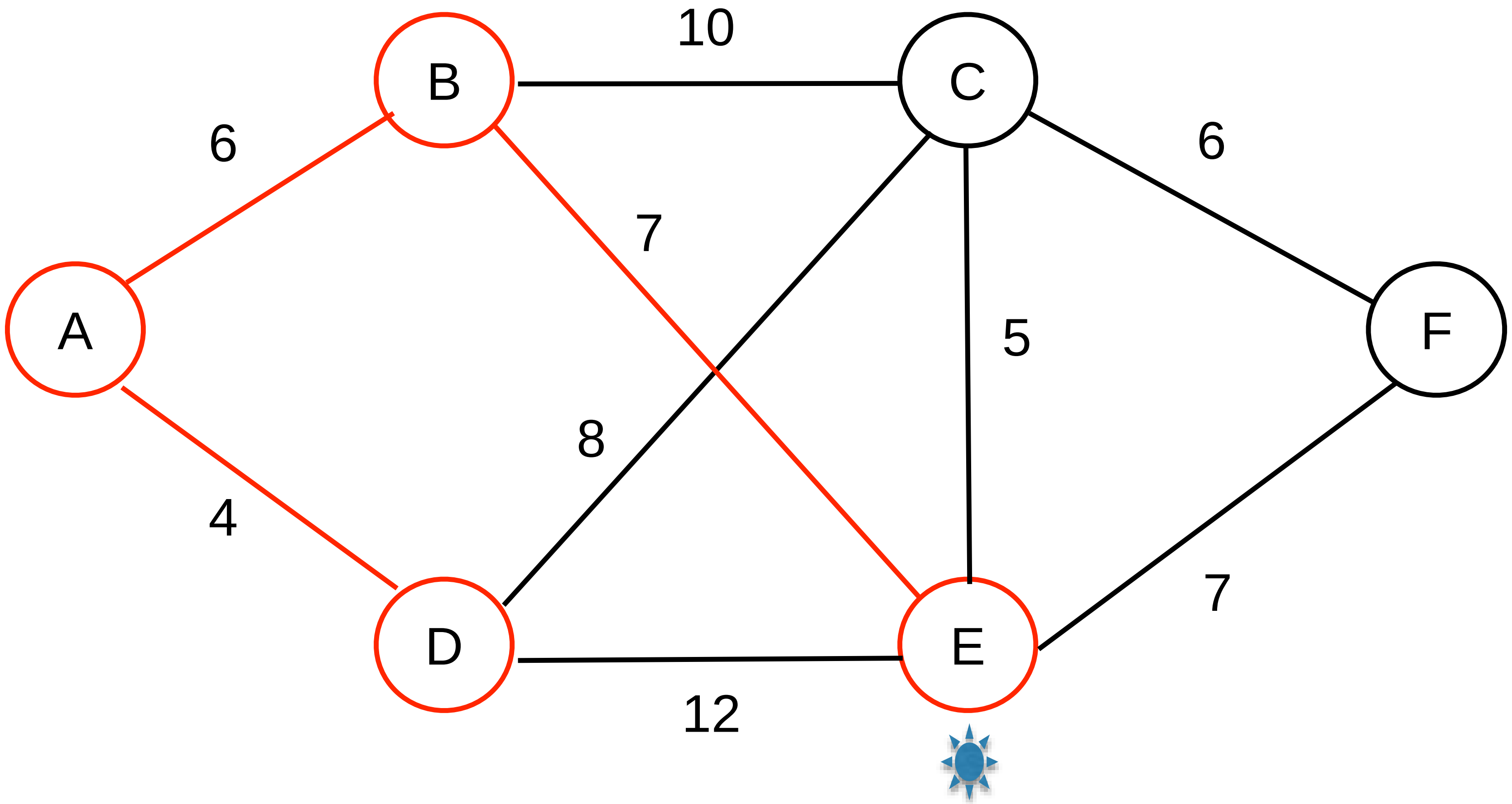
- Now B is the newest office
- Repeat the same for B

# MST with Weighted Graphs



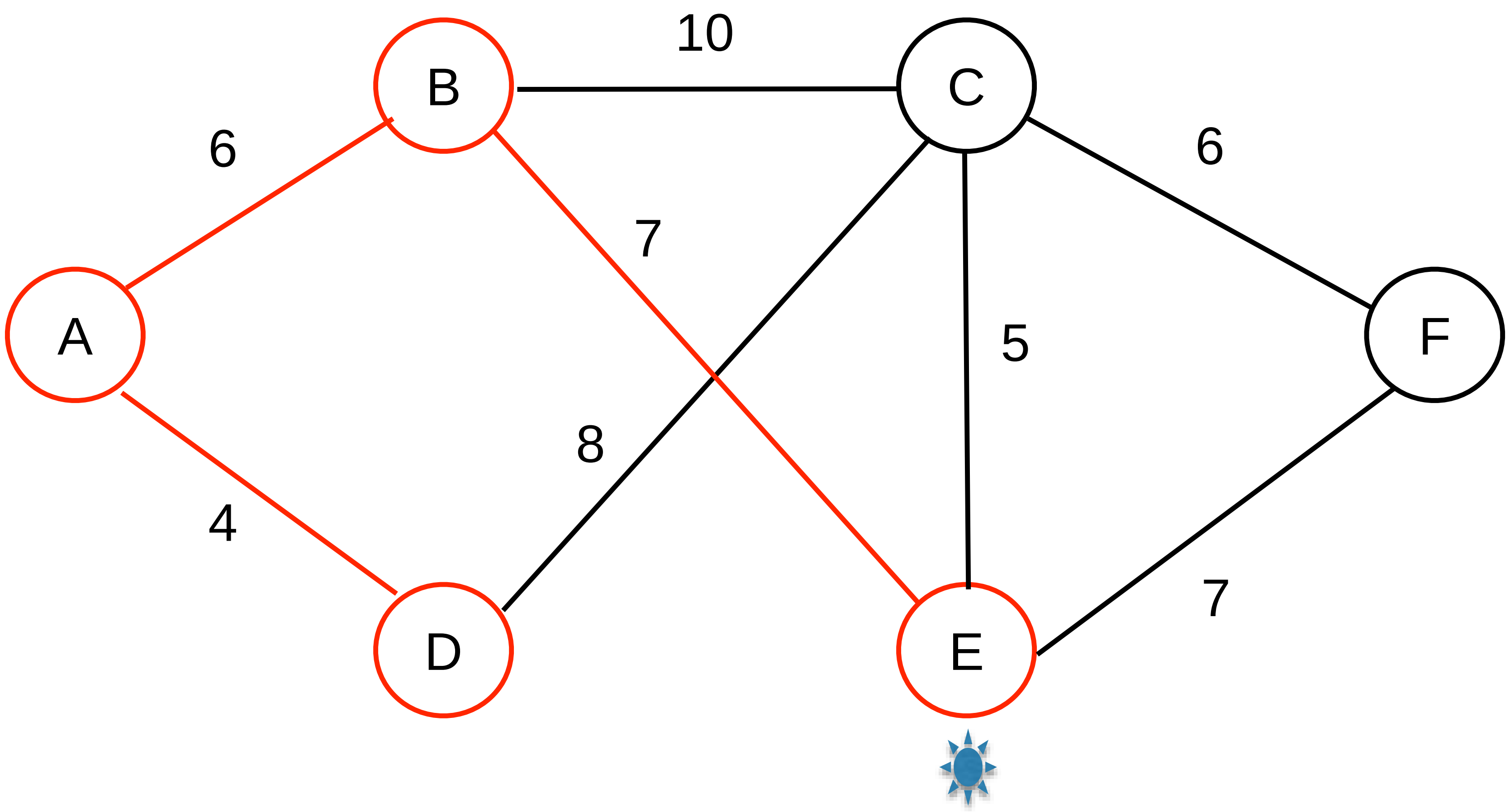

| Edge | Weight |
|---|---|
| DC | 8 |
| DE | 12 |
| BC | 10 |
| **BE** | **7** |

# MST with Weighted Graphs



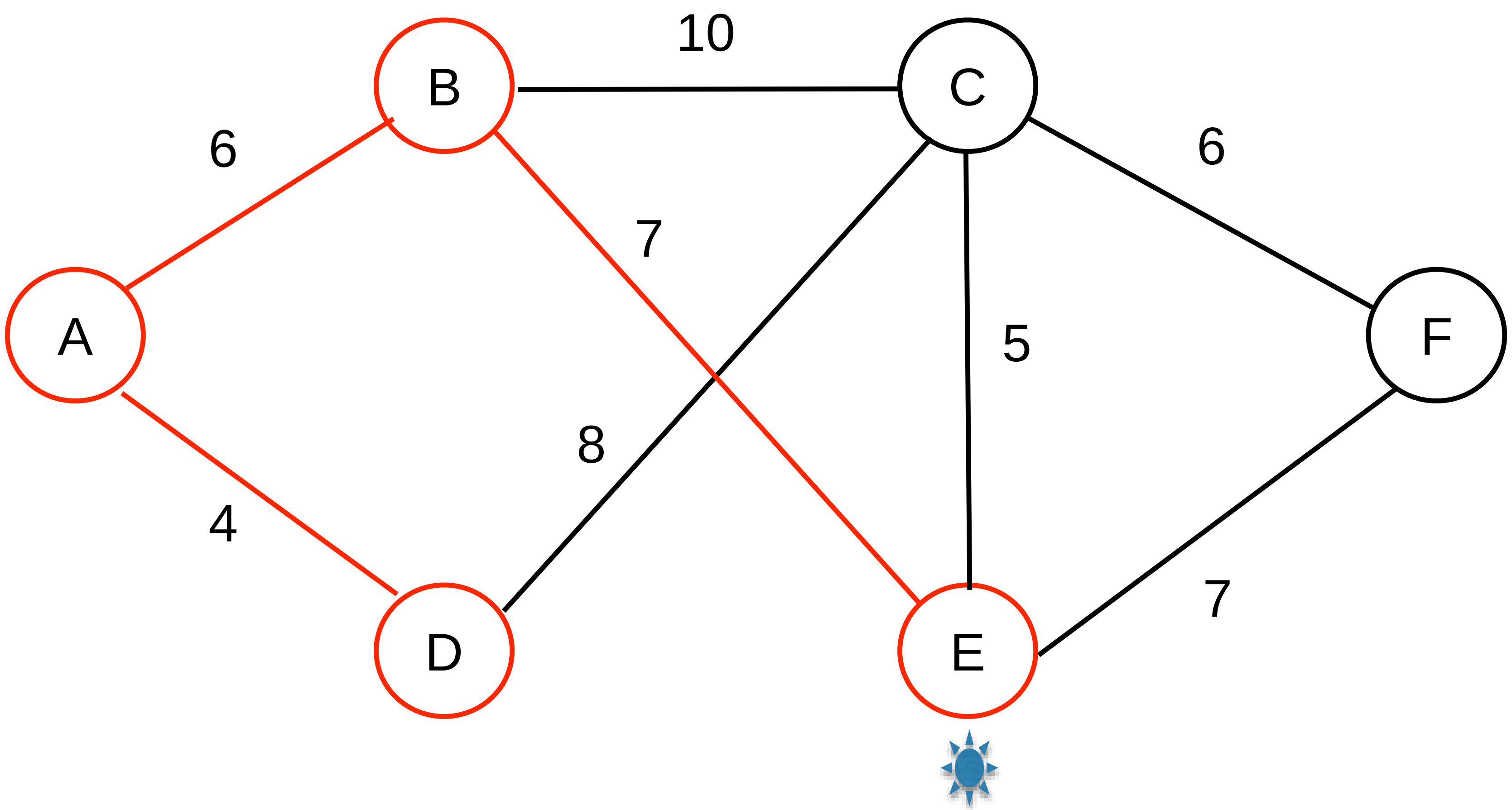

| Edge | Weight |
|---|---|
| DC | 8 |
| **DE** | **12** |
| BC | 10 |
| | |

# MST with Weighted Graphs



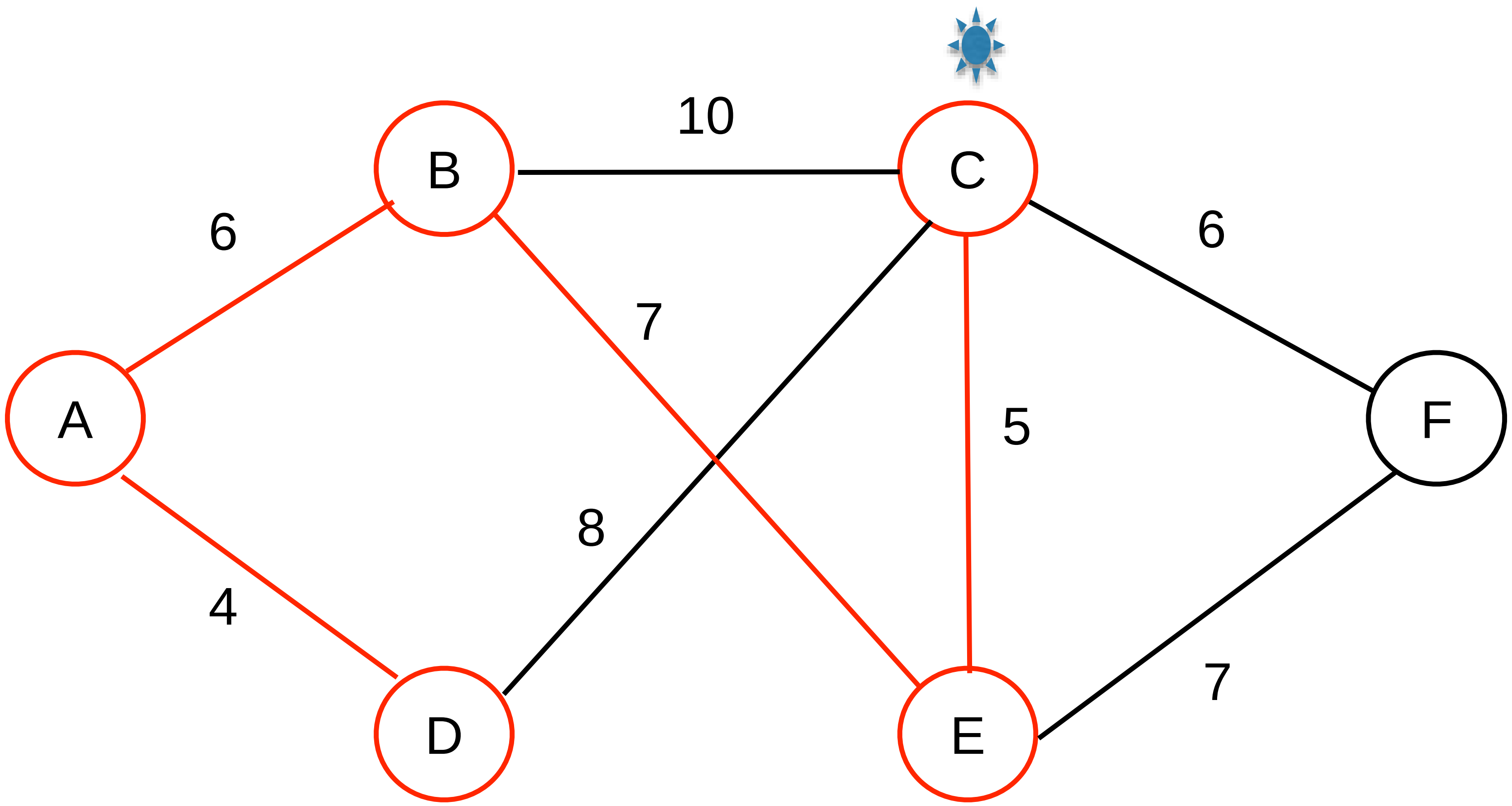

| Edge | Weight |
|---|---|
| DC | 8 |
| BC | 10 |
| | |

# MST with Weighted Graphs



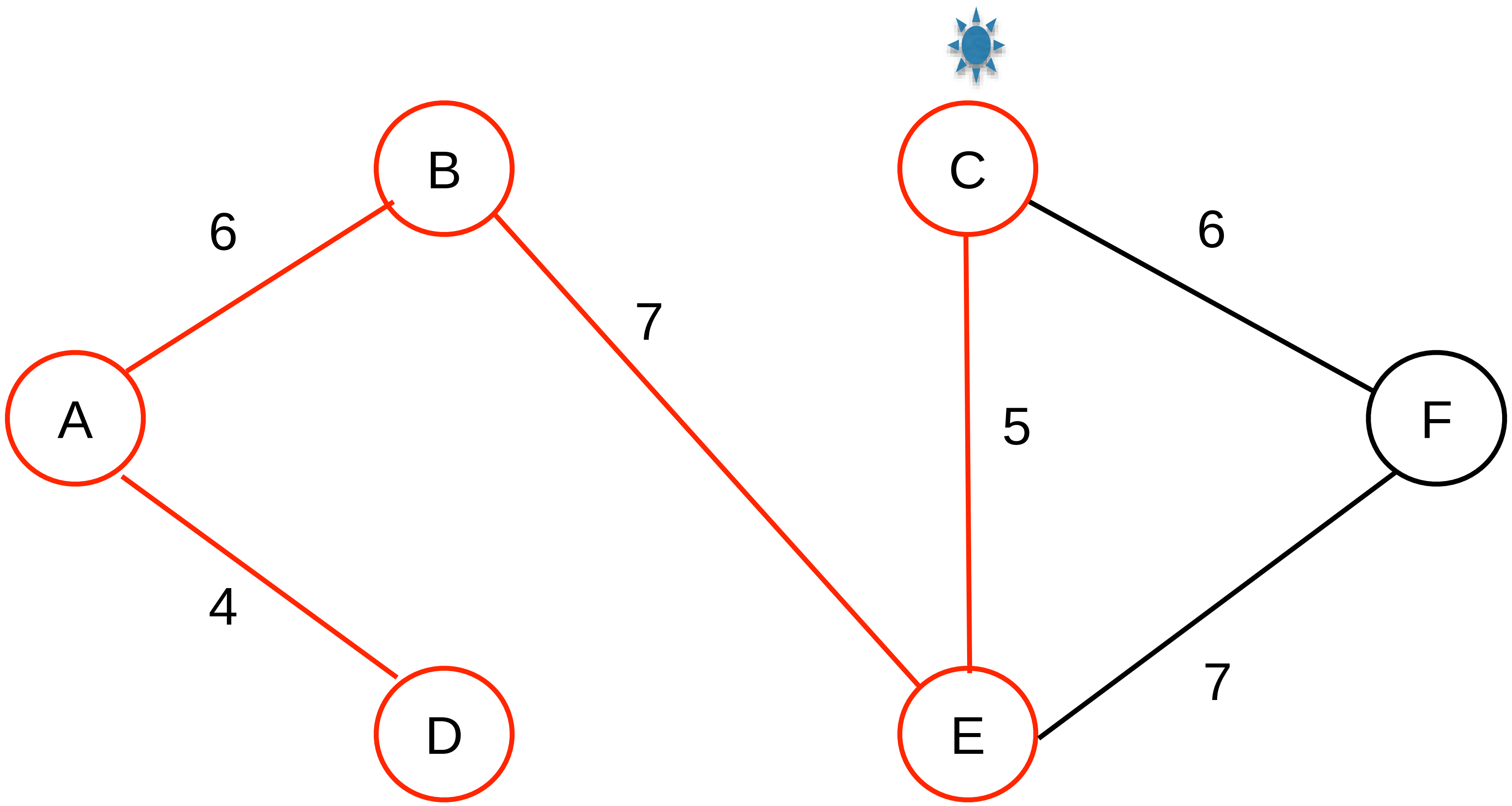

| Edge | Weight |
|---|---|
| DC | 8 |
| BC | 10 |
| **EC** | **5** |
| EF | 7 |

# MST with Weighted Graphs



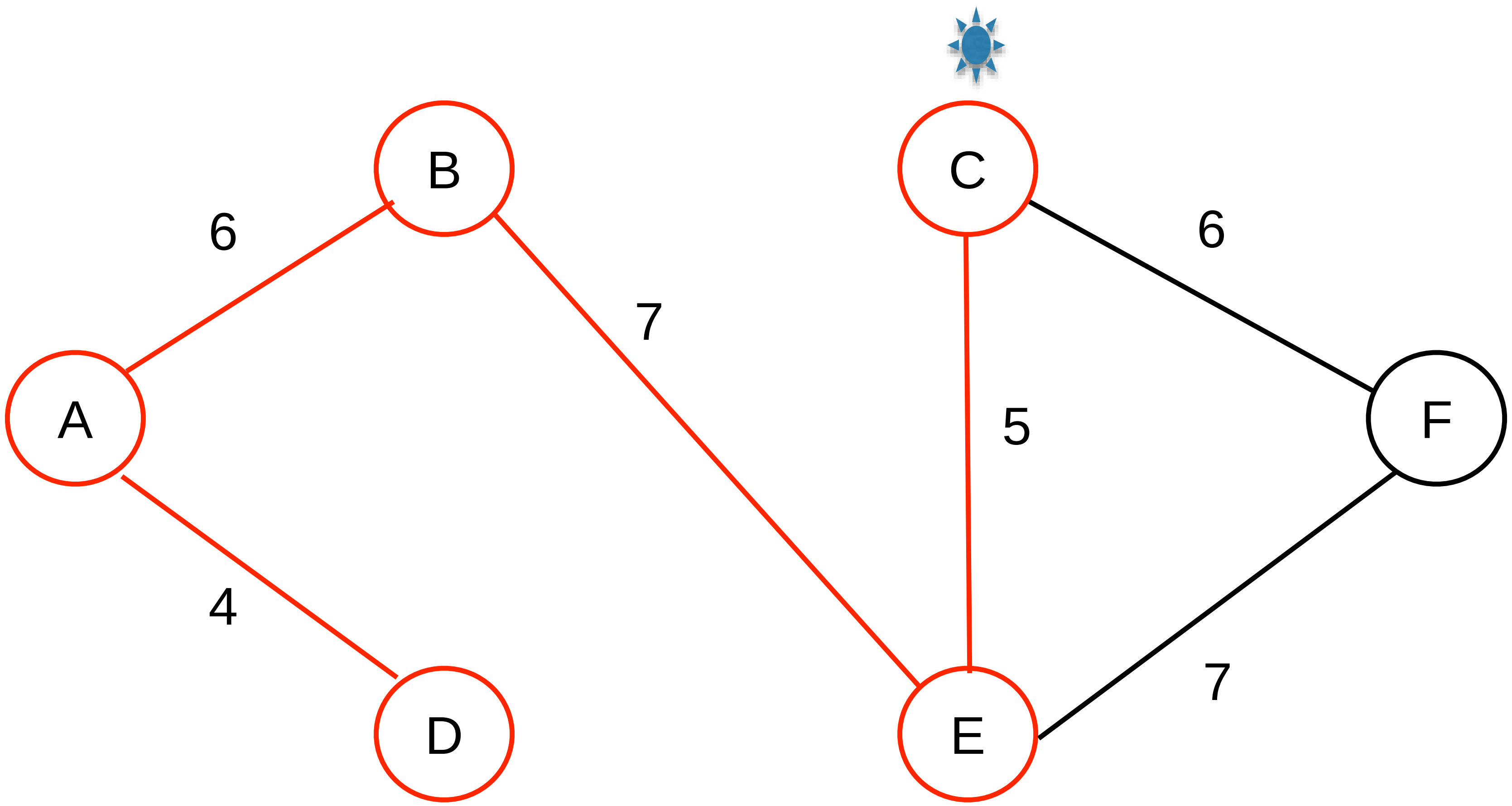

| Edge | Weight |
|---|---|
| **DC** | **8** |
| **BC** | **10** |
| | |
| EF | 7 |

# MST with Weighted Graphs



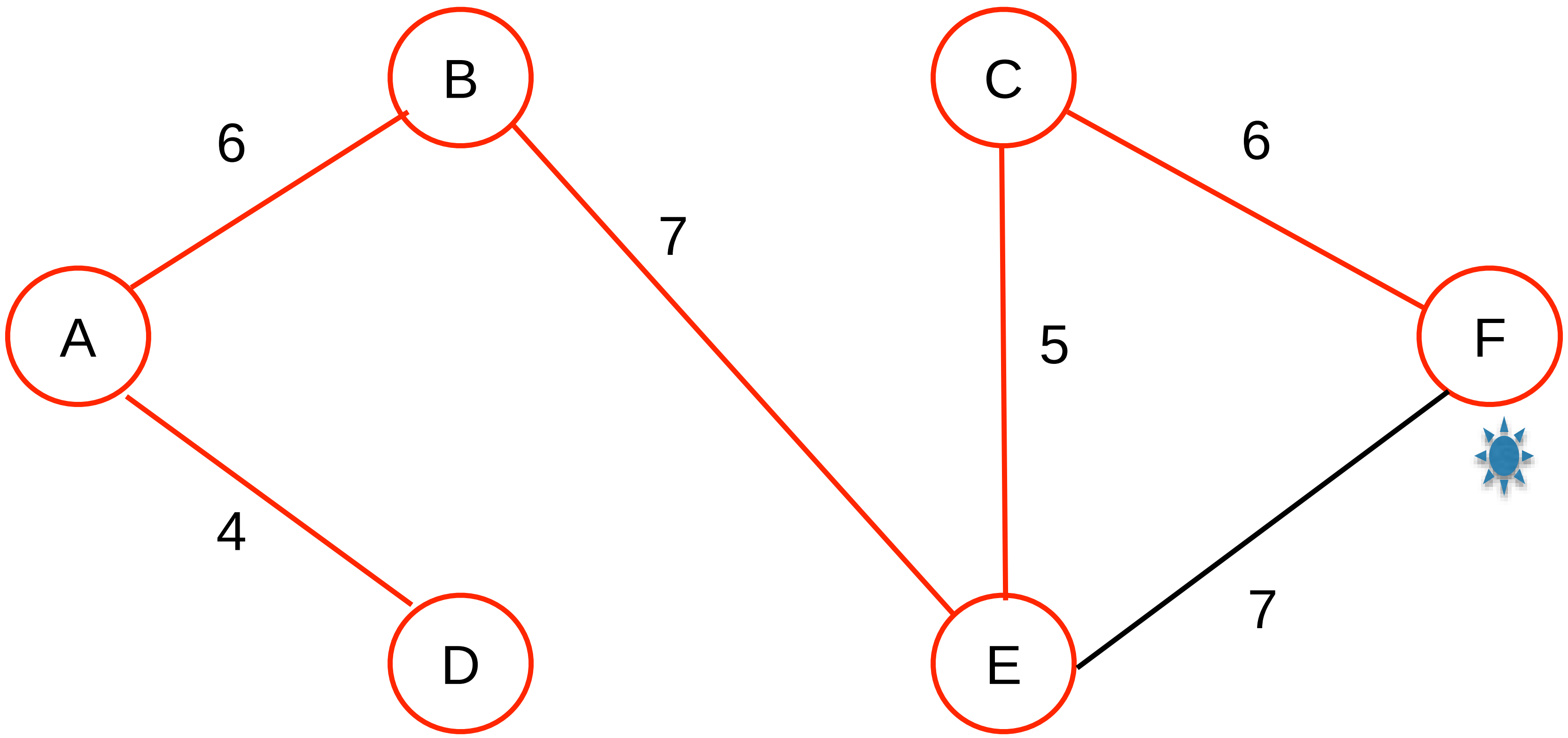

| Edge | Weight |
|---|---|
| EF | 7 |

# MST with Weighted Graphs



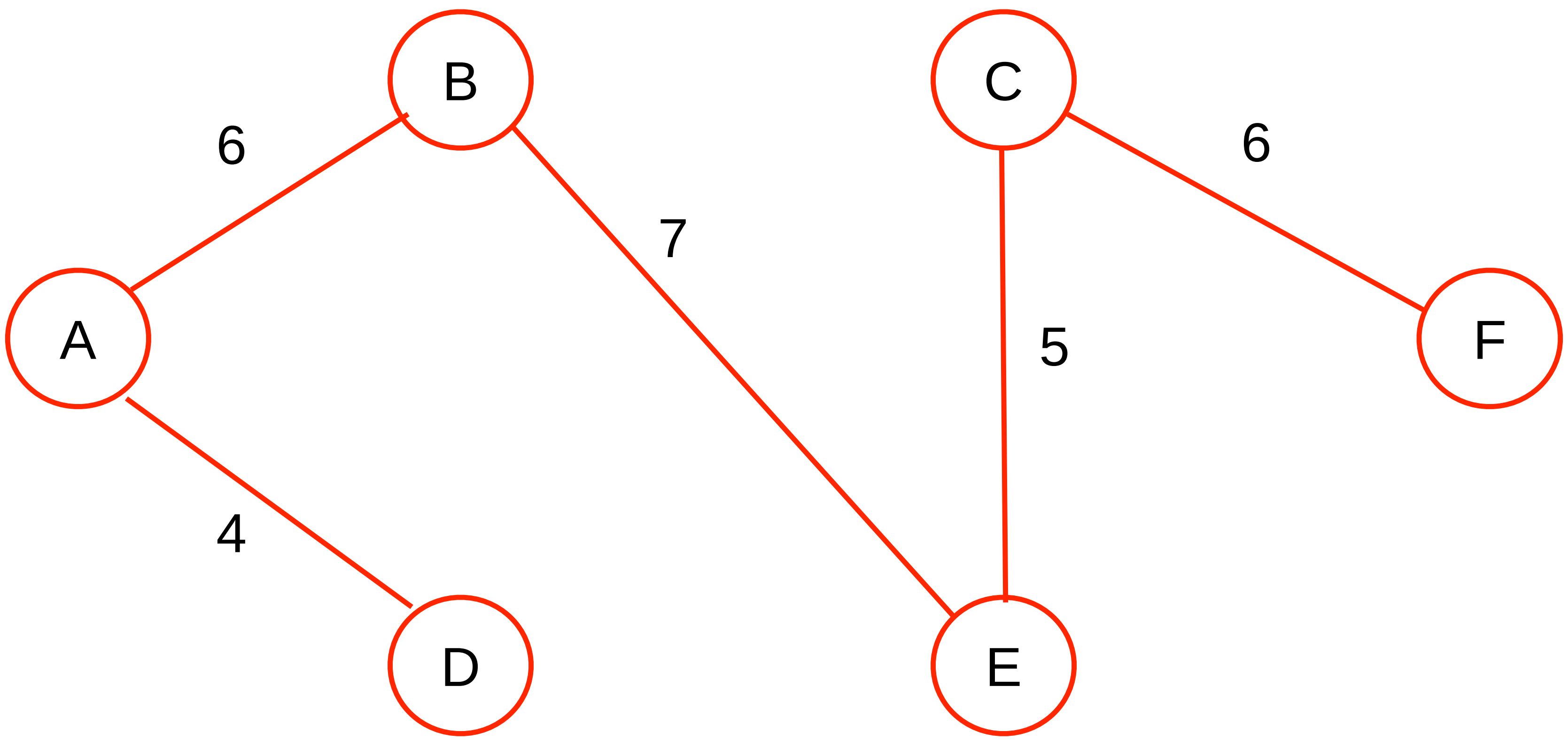

| Edge | Weight |
|---|---|
| EF | 7 |
| **CF** | **6** |

# MST with Weighted Graphs




| Edge | Weight |
| --- | --- |
| EF | 7 |
| | |

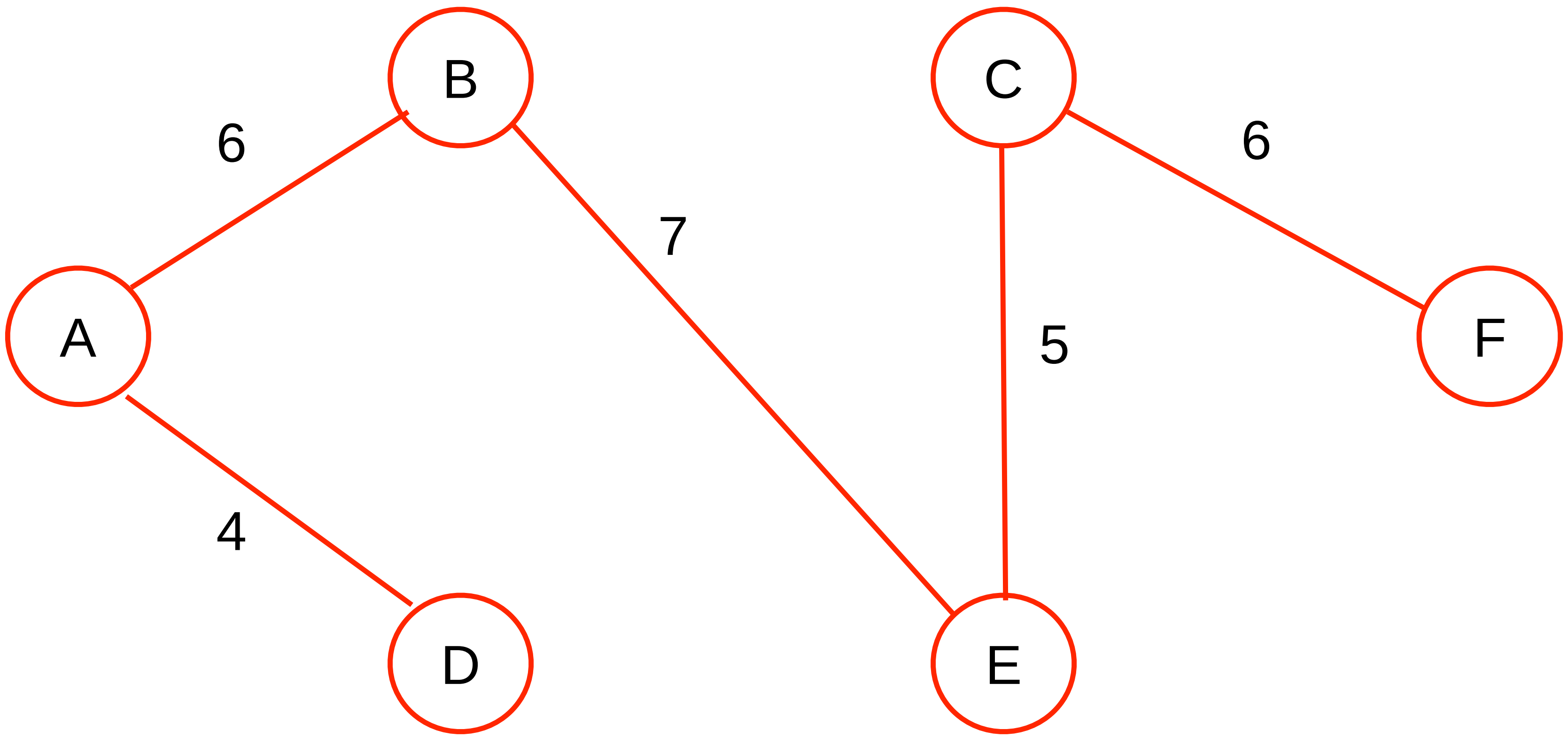
# MST with Weighted Graphs

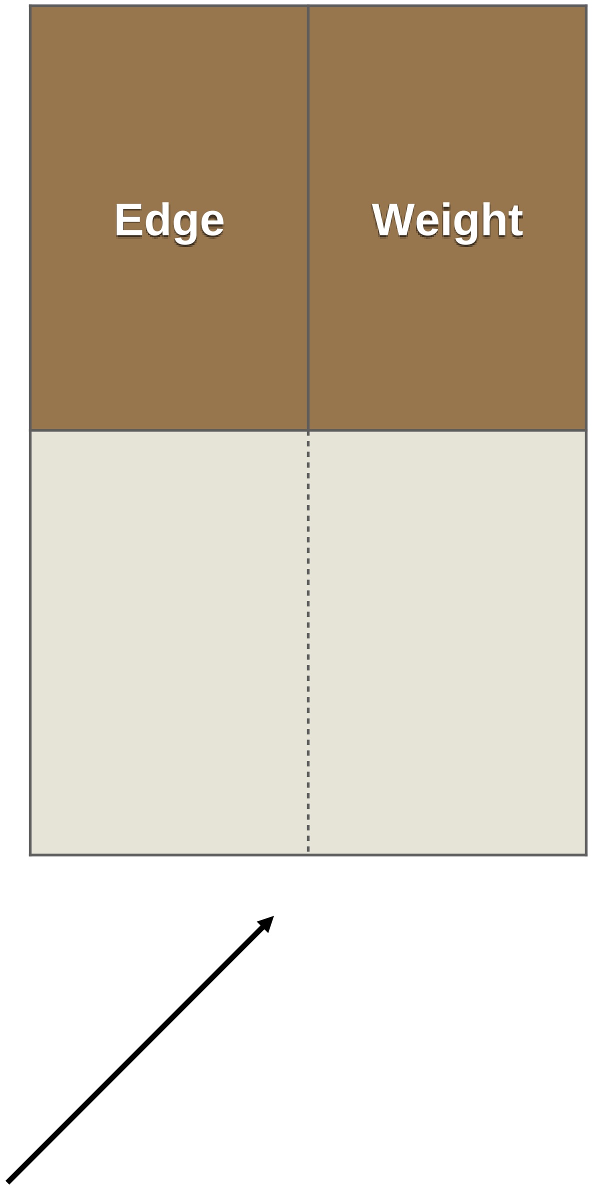



| Edge | Weight |
| --- | --- |
| | |

The Resultant Minimum Spanning Tree

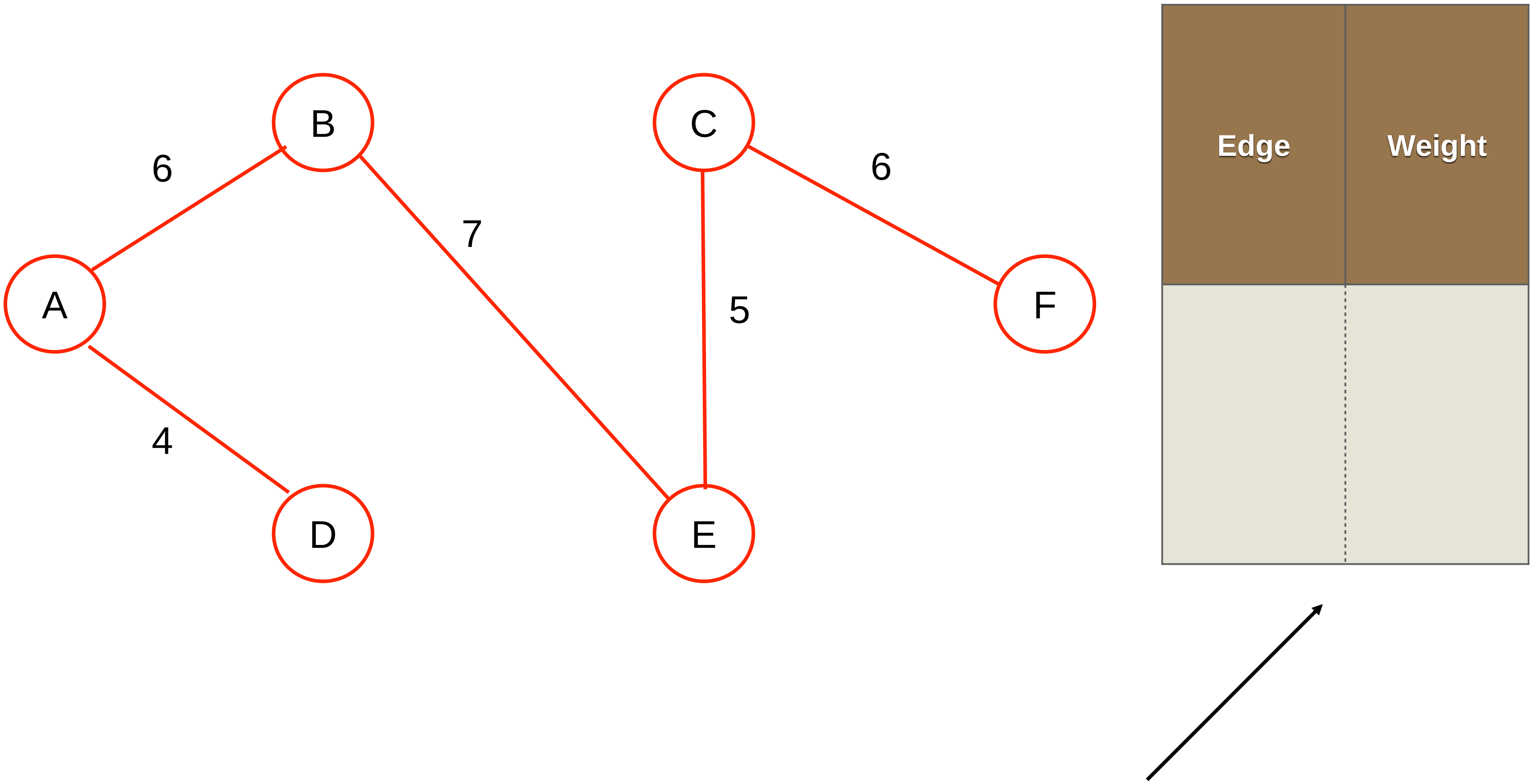
# MST with Weighted Graphs




Which ADT would you choose to implement this list?

# MST with Weighted Graphs




Which ADT would you choose to implement this list?

Priority Queue

# Prim-Jarnik's Algorithm

- This idea (which you just learned) of finding the MST is the basis of Prim's algorithm
- However, there is a slight modification to the basic idea to make it more efficient
  - More specifically, this modification is regarding the PQ and its entries

# PQ in Prim's Algorithm

- Each entry in the PQ is of the following type
  - {D[v], (v, e)}
- Where v is the vertex, D[v] is its key, and e is the incident edge with the minimum weight

# PQ in Prim's Algorithm

- Each entry in the PQ is of the following type
  - {D[v], (v, e)}
- Where v is the vertex, D[v] is its key, and e is the incident edge with the minimum weight

Thus only one edge is stored for each vertex. It is the one that has the lowest weight.

# Prim's Algorithm (1)

**Algorithm** PrimJarnik($G$):

*Input:* An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges
*Output:* A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$
$D[s] = 0$
**for** each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue $Q$ with an entry $(D[v], (v, \text{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \text{None})$ is the associated value.
**while** $Q$ is not empty **do**
  $(u, e)$ = value returned by $Q$.remove_min()
  Connect vertex $u$ to $T$ using edge $e$.
  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**
    {check if edge $(u, v)$ better connects $v$ to $T$}
    **if** $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex $v$ in $Q$ to $D[v]$.
      Change the value of vertex $v$ in $Q$ to $(v, e')$.
**return** the tree $T$

# Prim's Algorithm (2)

**Algorithm** PrimJarnik($G$):

*Input:* An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges

*Output:* A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$ ← Starting vertex

$D[s] = 0$

**for** each vertex $v \neq s$ **do**

  $D[v] = \infty$

Initialize $T = \emptyset$.

Initialize a priority queue $Q$ with an entry $(D[v], (v, \text{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \text{None})$ is the associated value.

**while** $Q$ is not empty **do**

  $(u, e) =$ value returned by $Q$.remove_min()

  Connect vertex $u$ to $T$ using edge $e$.

  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**

    {check if edge $(u, v)$ better connects $v$ to $T$}

    **if** $w(u, v) < D[v]$ **then**

      $D[v] = w(u, v)$

      Change the key of vertex $v$ in $Q$ to $D[v]$.

      Change the value of vertex $v$ in $Q$ to $(v, e')$.

**return** the tree $T$

# Prim's Algorithm (3)

**Algorithm** PrimJarnik($G$):

  ***Input:*** An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges
  ***Output:*** A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$
$D[s] = 0$ ← Set its key to 0
**for** each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue $Q$ with an entry $(D[v], (v, \text{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \text{None})$ is the associated value.
**while** $Q$ is not empty **do**
  $(u, e) =$ value returned by $Q$.remove_min()
  Connect vertex $u$ to $T$ using edge $e$.
  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**
    {check if edge $(u, v)$ better connects $v$ to $T$}
    **if** $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex $v$ in $Q$ to $D[v]$.
      Change the value of vertex $v$ in $Q$ to $(v, e')$.
**return** the tree $T$

# Prim's Algorithm (4)

**Algorithm** PrimJarnik($G$):

*Input:* An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges

*Output:* A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$
$D[s] = 0$
**for** each vertex $v \neq s$ **do**
  $D[v] = \infty$ — For every other vertex, set its key to infinity
Initialize $T = \emptyset$.
Initialize a priority queue $Q$ with an entry $(D[v], (v, \text{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \text{None})$ is the associated value.
**while** $Q$ is not empty **do**
  $(u, e) =$ value returned by $Q$.remove_min()
  Connect vertex $u$ to $T$ using edge $e$.
  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**
    {check if edge $(u, v)$ better connects $v$ to $T$}
    **if** $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex $v$ in $Q$ to $D[v]$.
      Change the value of vertex $v$ in $Q$ to $(v, e')$.
**return** the tree $T$

# Prim's Algorithm (5)

**Algorithm** PrimJarnik($G$):

***Input:*** An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges
***Output:*** A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$
$D[s] = 0$
**for** each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.  Initial tree
Initialize a priority queue $Q$ with an entry $(D[v], (v, \text{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \text{None})$ is the associated value.
**while** $Q$ is not empty **do**
  $(u, e) =$ value returned by $Q$.remove_min()
  Connect vertex $u$ to $T$ using edge $e$.
  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**
    {check if edge $(u, v)$ better connects $v$ to $T$}
    **if** $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex $v$ in $Q$ to $D[v]$.
      Change the value of vertex $v$ in $Q$ to $(v, e')$.
**return** the tree $T$

# Prim's Algorithm (6)

**Algorithm** PrimJarnik($G$):

*Input:* An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges

*Output:* A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$
$D[s] = 0$
**for** each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue $Q$ with an entry $(D[v], (v, \text{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \text{None})$ is the associated value.
**while** $Q$ is not empty **do**
  $(u, e) =$ value returned by $Q.\text{remove\_min}()$
  Connect vertex $u$ to $T$ using edge $e$.
  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**
    {check if edge $(u, v)$ better connects $v$ to $T$}
    **if** $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex $v$ in $Q$ to $D[v]$.
      Change the value of vertex $v$ in $Q$ to $(v, e')$.
**return** the tree $T$

Create PQ.
Notice the "None"

# Prim's Algorithm (7)

**Algorithm** PrimJarnik($G$):

  ***Input:*** An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges
  ***Output:*** A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$
$D[s] = 0$
**for** each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue $Q$ with an entry $(D[v], (v, \mathsf{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \mathsf{None})$ is the associated value.
**while** $Q$ is not empty **do**
  $(u, e)$ = value returned by $Q$.remove_min()
  Connect vertex $u$ to $T$ using edge $e$.
  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**
    {check if edge $(u, v)$ better connects $v$ to $T$}
    **if** $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex $v$ in $Q$ to $D[v]$.
      Change the value of vertex $v$ in $Q$ to $(v, e')$.
**return** the tree $T$

1. While Q is not empty, remove the vertex u with the minimum key, and put it in T u sing e

# Prim's Algorithm (8)

**Algorithm** PrimJarnik($G$):

*Input:* An undirected, weighted, connected graph $G$ with $n$ vertices and $m$ edges
*Output:* A minimum spanning tree $T$ for $G$

Pick any vertex $s$ of $G$
$D[s] = 0$
**for** each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue $Q$ with an entry $(D[v], (v, \text{None}))$ for each vertex $v$, where $D[v]$ is the key in the priority queue, and $(v, \text{None})$ is the associated value.
**while** $Q$ is not empty **do**
  $(u, e)$ = value returned by $Q$.remove_min()
  Connect vertex $u$ to $T$ using edge $e$.
  **for** each edge $e' = (u, v)$ such that $v$ is in $Q$ **do**
    {check if edge $(u, v)$ better connects $v$ to $T$}
    **if** $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex $v$ in $Q$ to $D[v]$.
      Change the value of vertex $v$ in $Q$ to $(v, e')$.
**return** the tree $T$

1. While Q is not empty, remove the vertex u with the minimum key, and put it in T us ing e

2. And, do the following

"For every adjacent vertex v of u, replac e its key D[v] with the weight w(u,v), only if it i s less than the key "

# Prim's Algorithm Time Complexity

- Three main tasks

1. Creation of PQ – O(|V| log (|V|)
2. Emptying the PQ – O(|V| log (|V|)
3. Updating the PQ – O(|E| log (|V|)

- Thus, T: O(|E| log (|V|)

# Shortest Path

# Shortest Path

- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v.***
  - Length of a path is the sum of the weights of its edges.
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions

# Shortest Path Properties

Property 1:

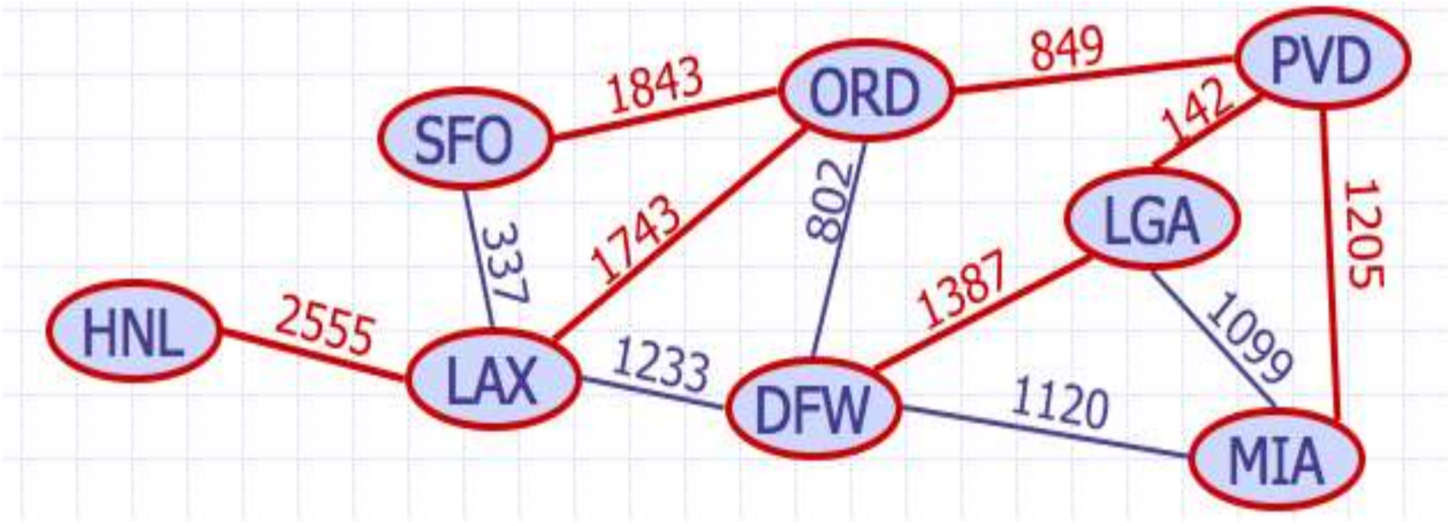
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence (PVD)



# Dijkstra’s Algorithm (1)

- Finds the shortest path from a given node *u* to every other node in G

- Works on the same idea as the Prim’s algorithm, with a small difference

# What’s the Similarity with Prim’s Algorithm?

- We grow a “cloud” of vertices, beginning with ***s*** and eventually covering all the vertices

- We store (in a PQ) with each vertex ***v*** a label ***d***(***v***) representing the distance of ***v*** from ***s***

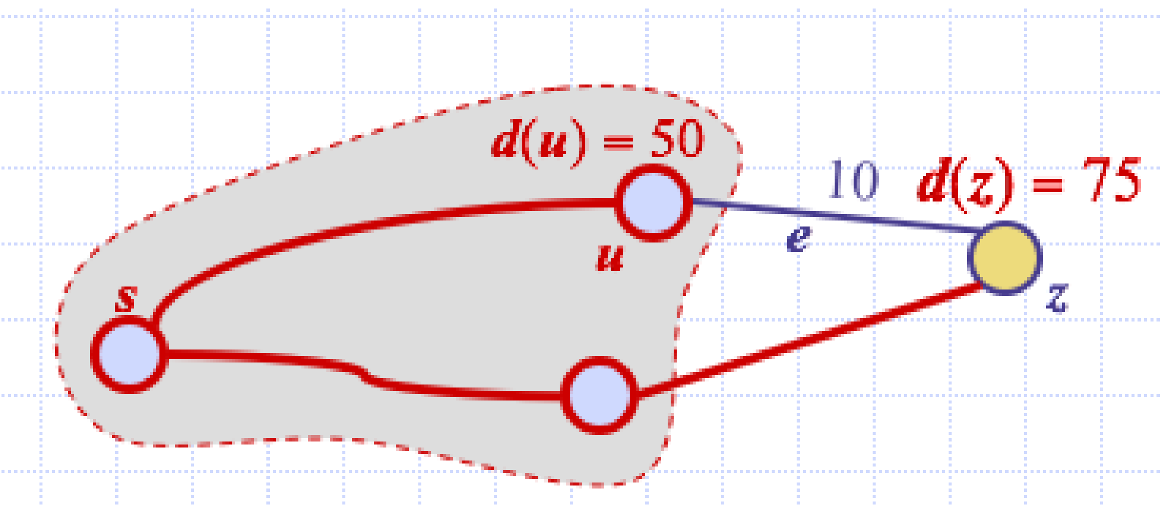
- At each step

  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label, ***d***(***u***)

  - We update the labels of the vertices adjacent to ***u***

# What's the difference?

- Consider an edge ***e* = (*u*,*z*)** such that
  - ***u*** is the vertex most recently added to the cloud
  - ***z*** is not in the cloud



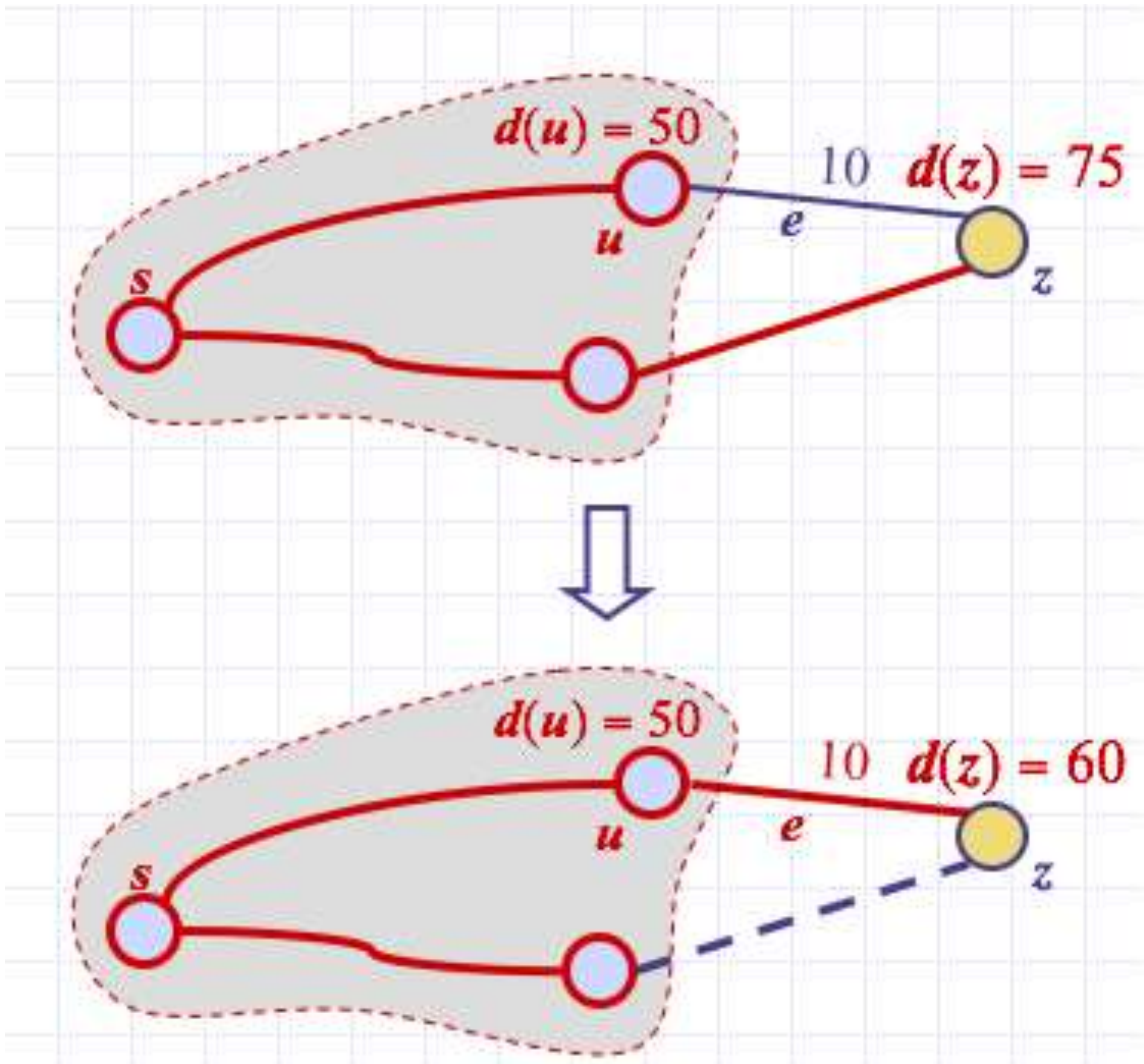

- The relaxation of edge ***e*** updates distance ***d*(*z*)** as follows:

$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$

# What's the difference?

- Consider an edge $e = (u,z)$ such that
  - $u$ is the vertex most recently added to the cloud
  - $z$ is not in the cloud
- The relaxation of edge $e$ updates distance $d(z)$ as follows:

$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



# Pseudo code

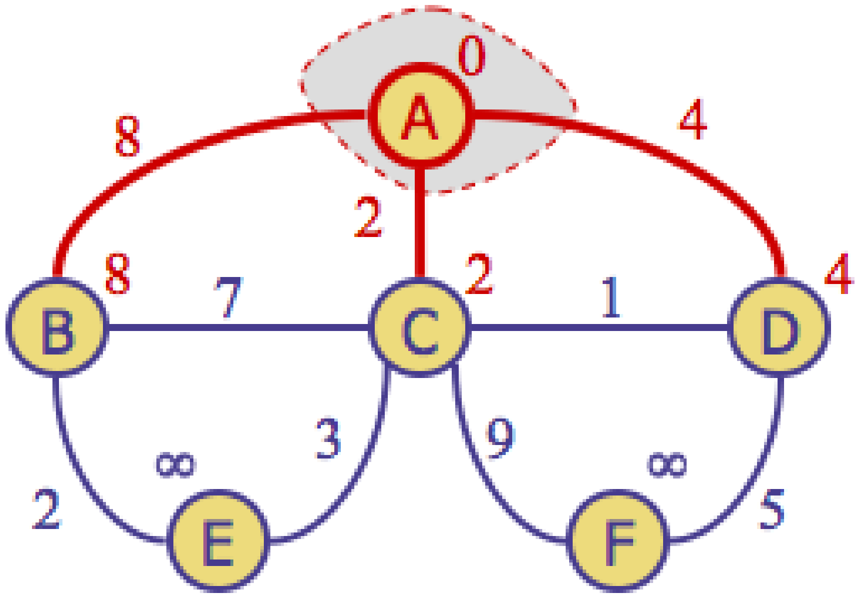
**Algorithm** ShortestPath$(G, s)$:

*Input:* A weighted graph $G$ with nonnegative edge weights, and a distinguished vertex $s$ of $G$.

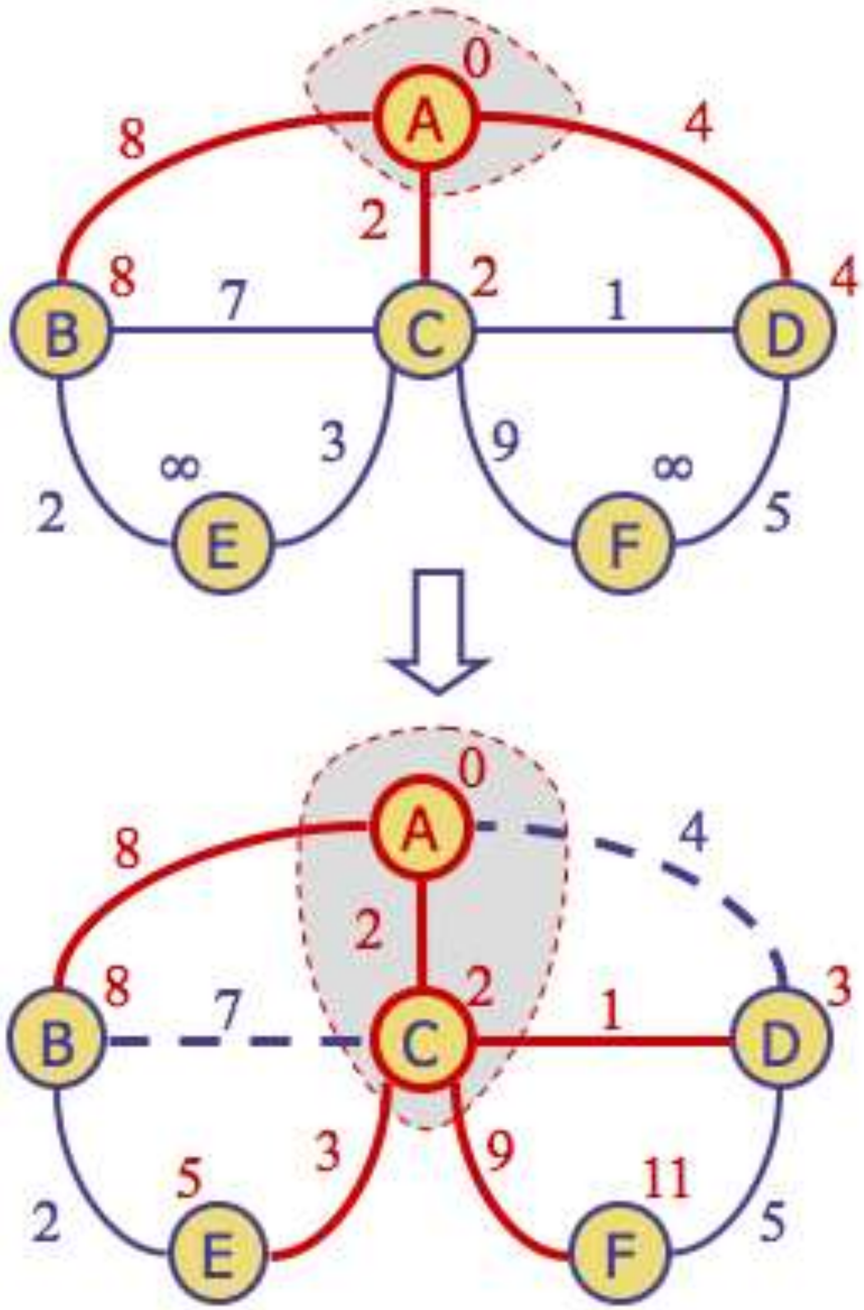
*Output:* The length of a shortest path from $s$ to $v$ for each vertex $v$ of $G$.

Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.
**while** $Q$ is not empty **do**
  {pull a new vertex $u$ into the cloud}
  $u$ = value returned by $Q$.remove_min()
  **for** each vertex $v$ adjacent to $u$ such that $v$ is in $Q$ **do**
    {perform the ***relaxation*** procedure on edge $(u, v)$}
    **if** $D[u] + w(u, v) < D[v]$ **then**
      $D[v] = D[u] + w(u, v)$
      Change to $D[v]$ the key of vertex $v$ in $Q$.
**return** the label $D[v]$ of each vertex $v$

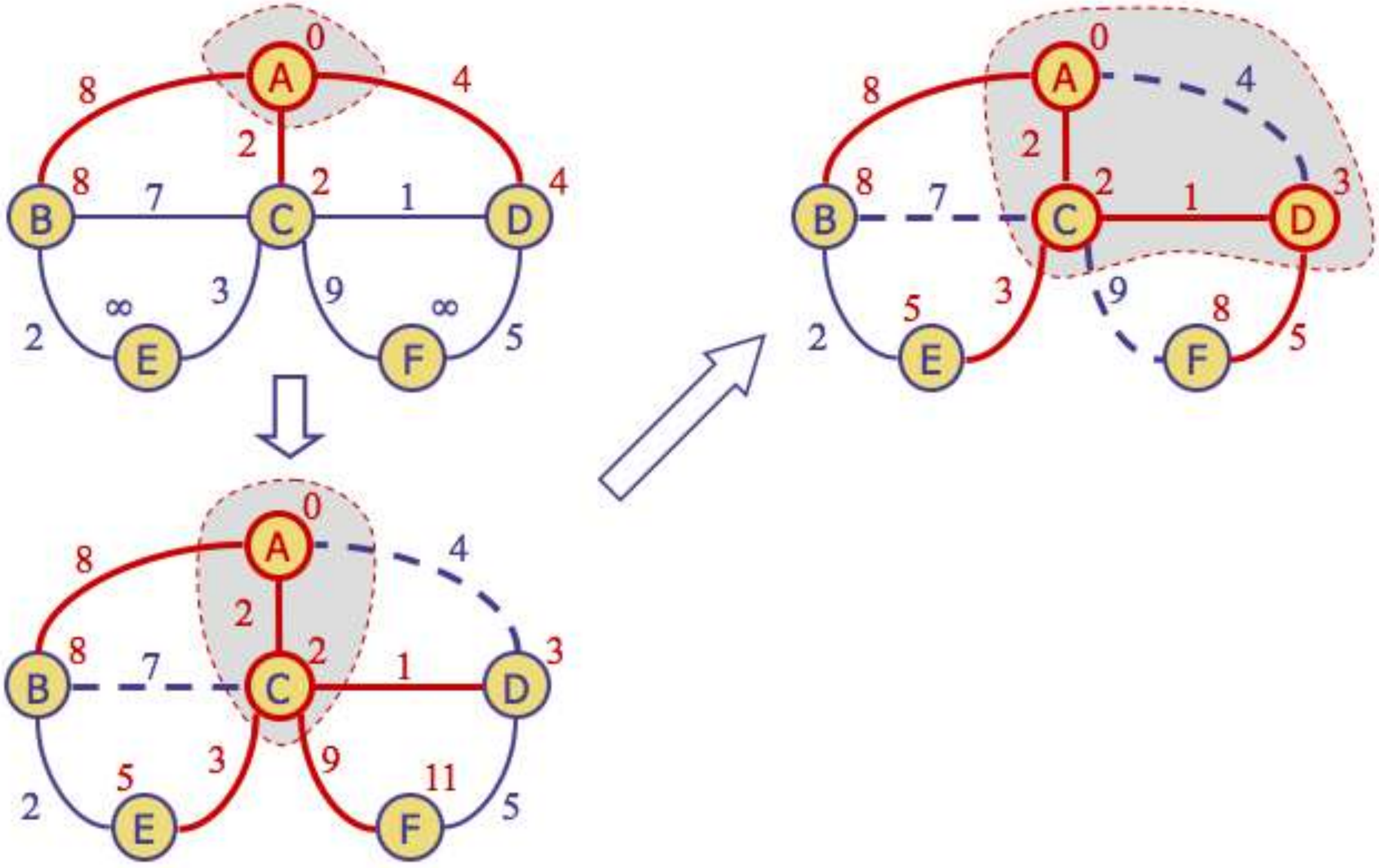
# Example



# Example

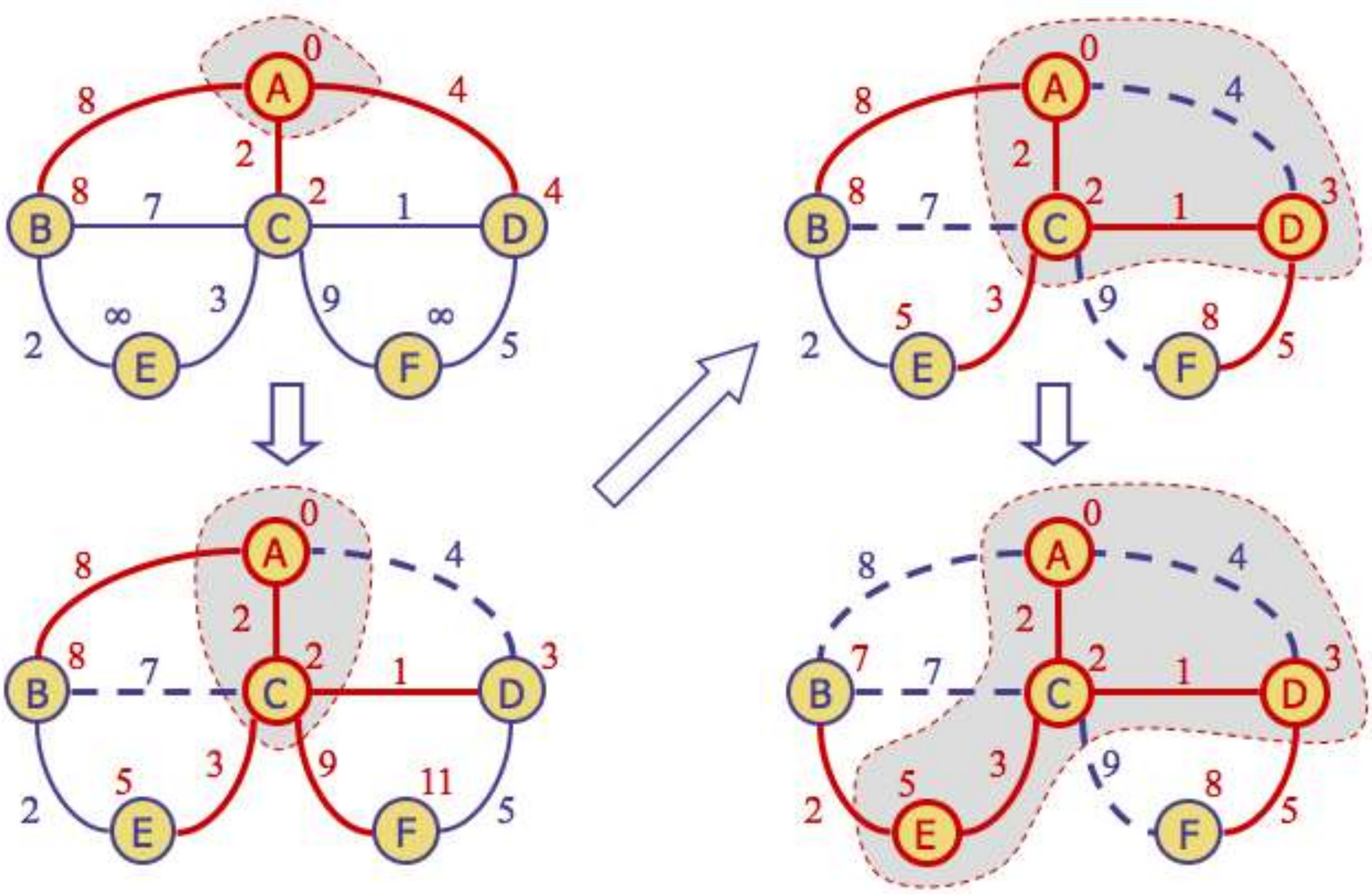


# Example

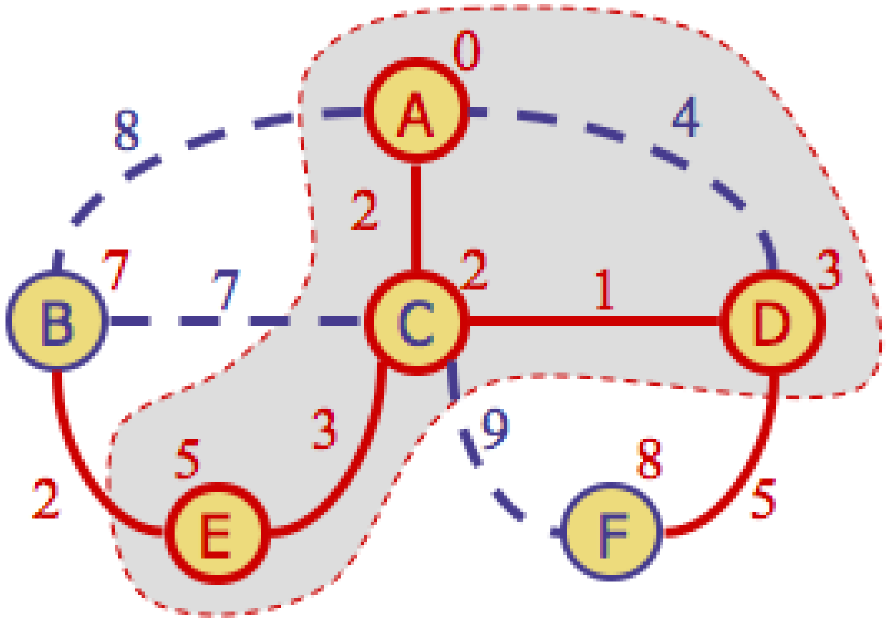


0
A
8
4
2
8
2
4
B
7
C
1
D
∞
3
9
∞
2
E
F
5
0
A
8
4
2
8
2
3
B
7
C
1
D
5
3
9
11
2
E
F
5
0
A
8
4
2
8
2
3
B
7
C
1
D
5
3
9
8
2
E
F
5

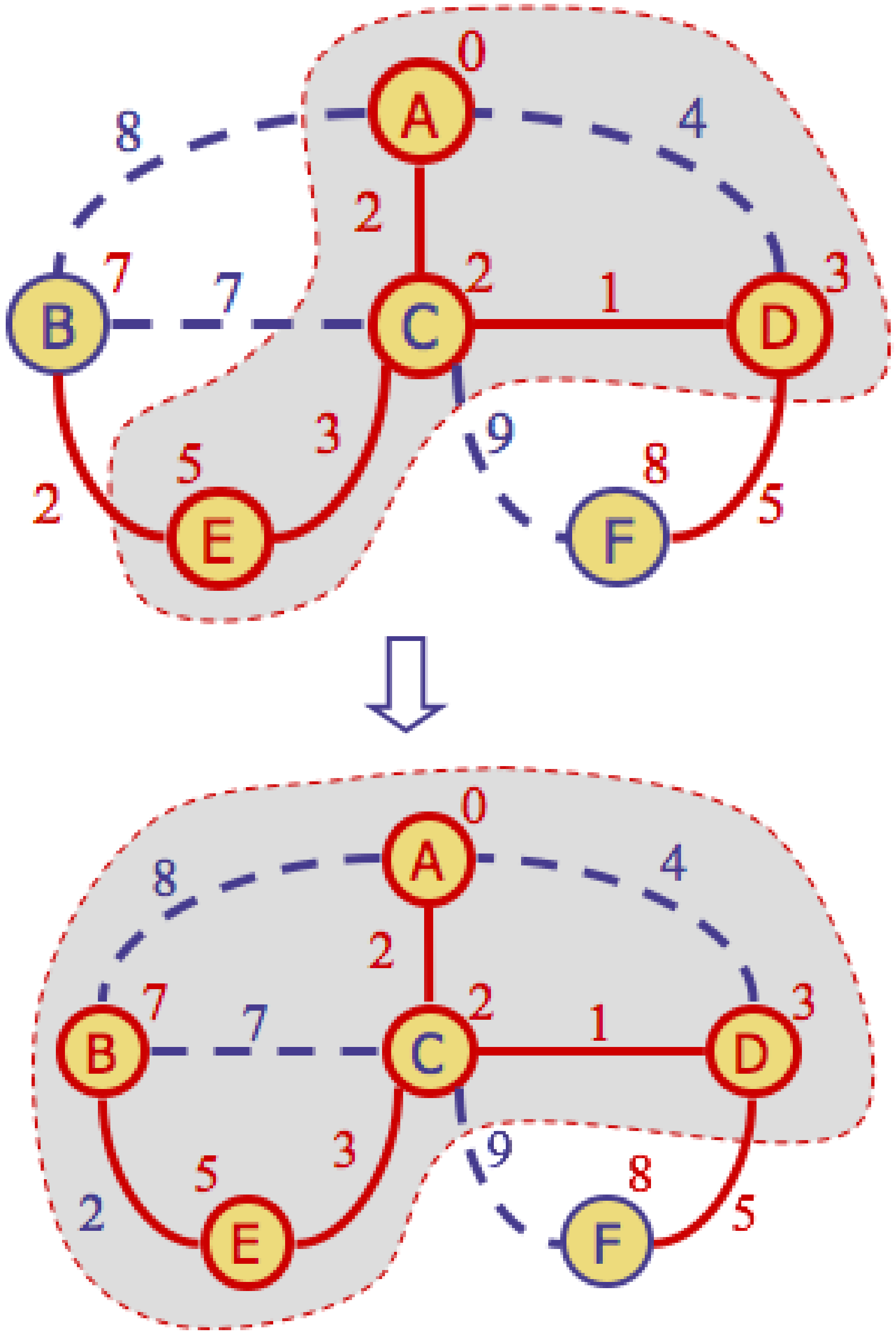
# Example


A 0 B 8 C 2 D 4 E ∞ F ∞
8 4 2 7 1 3 9 2 5
A 0 B 8 C 2 D 3 E 5 F 11
8 4 2 7 1 3 9 2 5
A 0 B 8 C 2 D 3 E 5 F 8
8 4 2 7 1 3 9 2 5
A 0 B 7 C 2 D 3 E 5 F 8
8 4 2 7 1 3 9 2 5

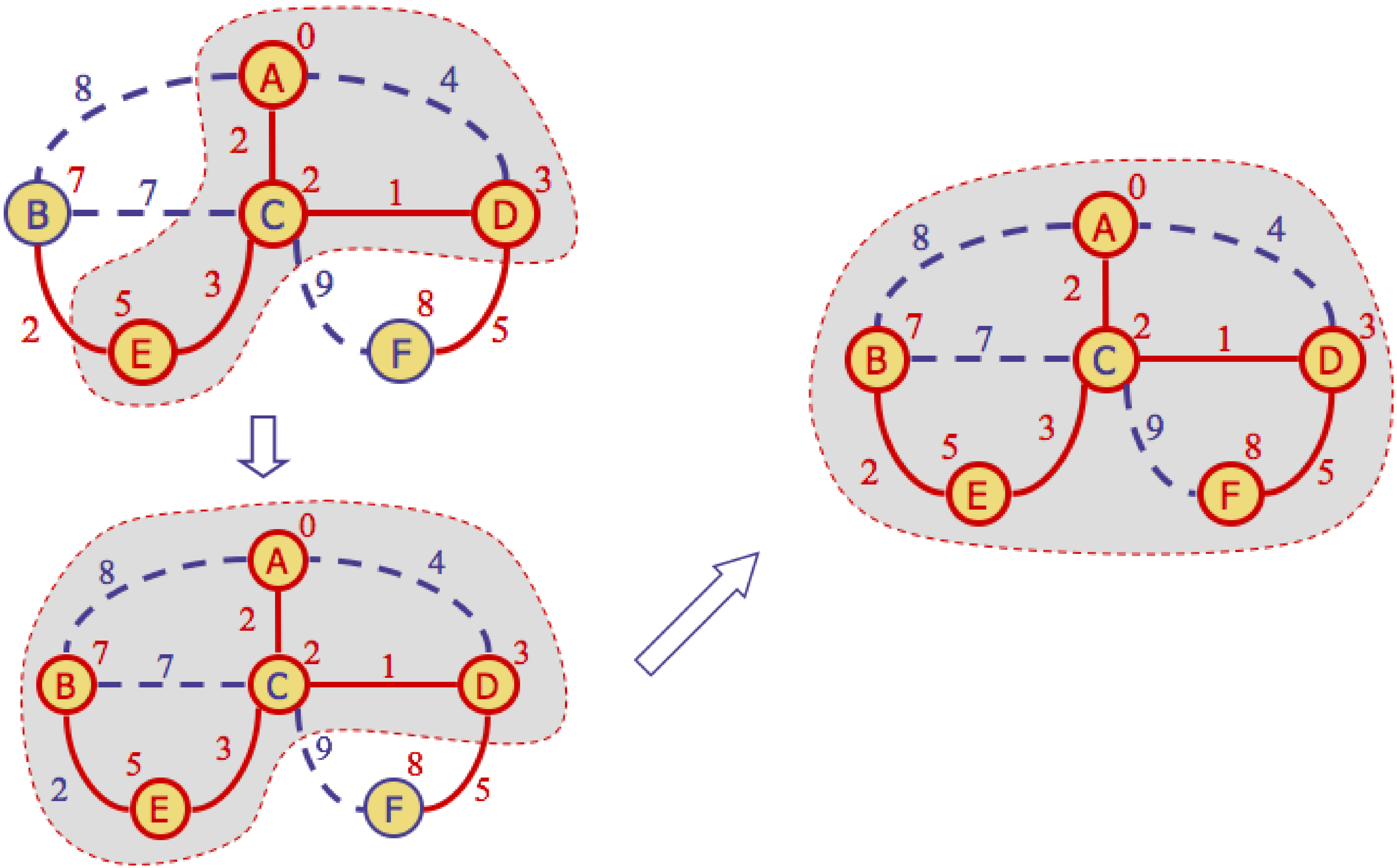
# Example



# Example



# Example



# Time Complexity

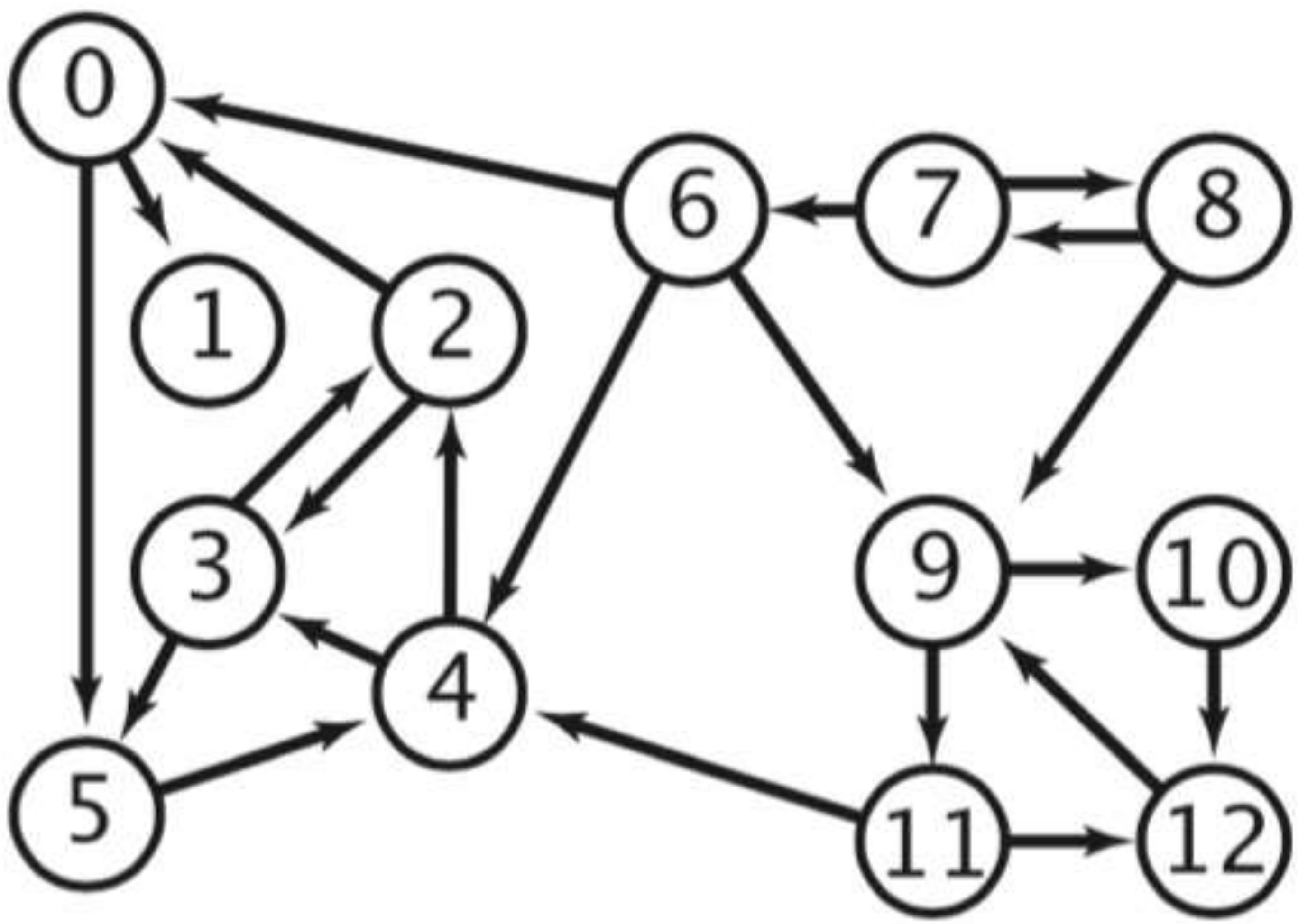
- Three main tasks

1. Creation of PQ – O(|V| log (|V|)
2. Emptying the PQ – O(|V| log (|V|)
3. Updating the PQ – O(|E| log (|V|)

- Thus, T: O(|E| log (|V|)

# Topological Sorting with Directed Graphs

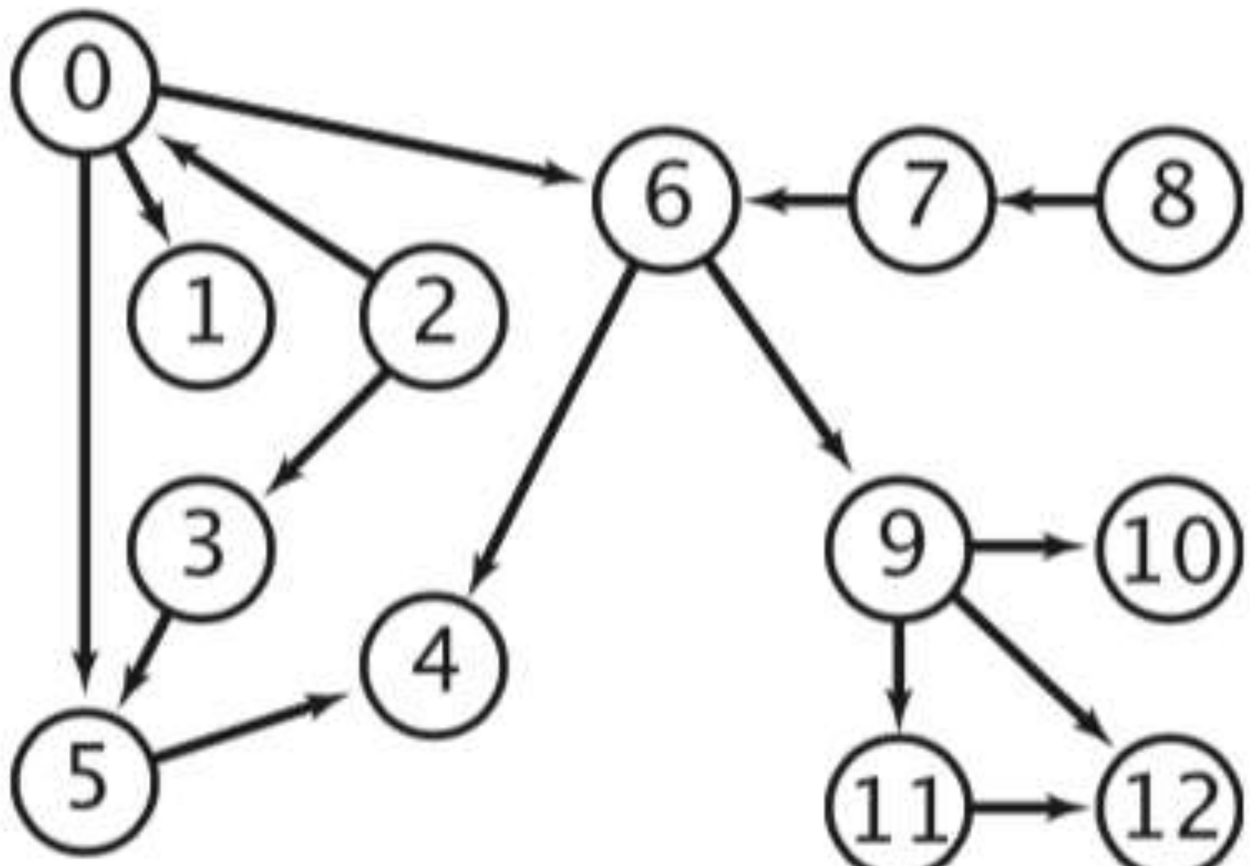
# Directed Graphs




outdegree, indegree, directed path, directed cycle

# Directed Acyclic Graphs

- A digraph with no directed cycle



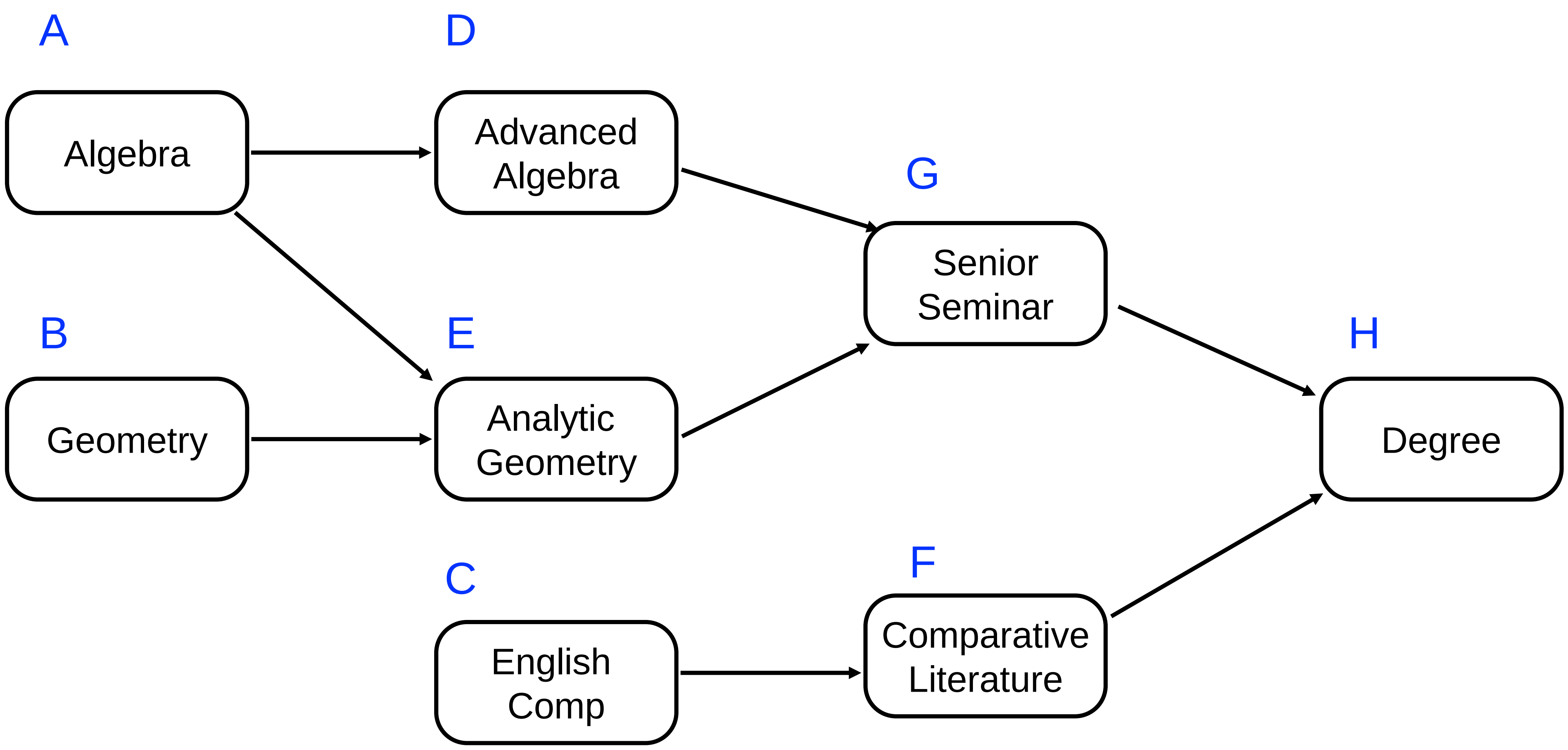
# Topological Sorting

- Most important operation on directed acyclic graphs (DAGs)
- It orders vertices on a line such that all directed edges go from left to right
- Why is it important?

# Topological Sorting

- Sometimes, students cannot just take any course they want
- Some courses have prerequisites
- Look at the example on the next slide

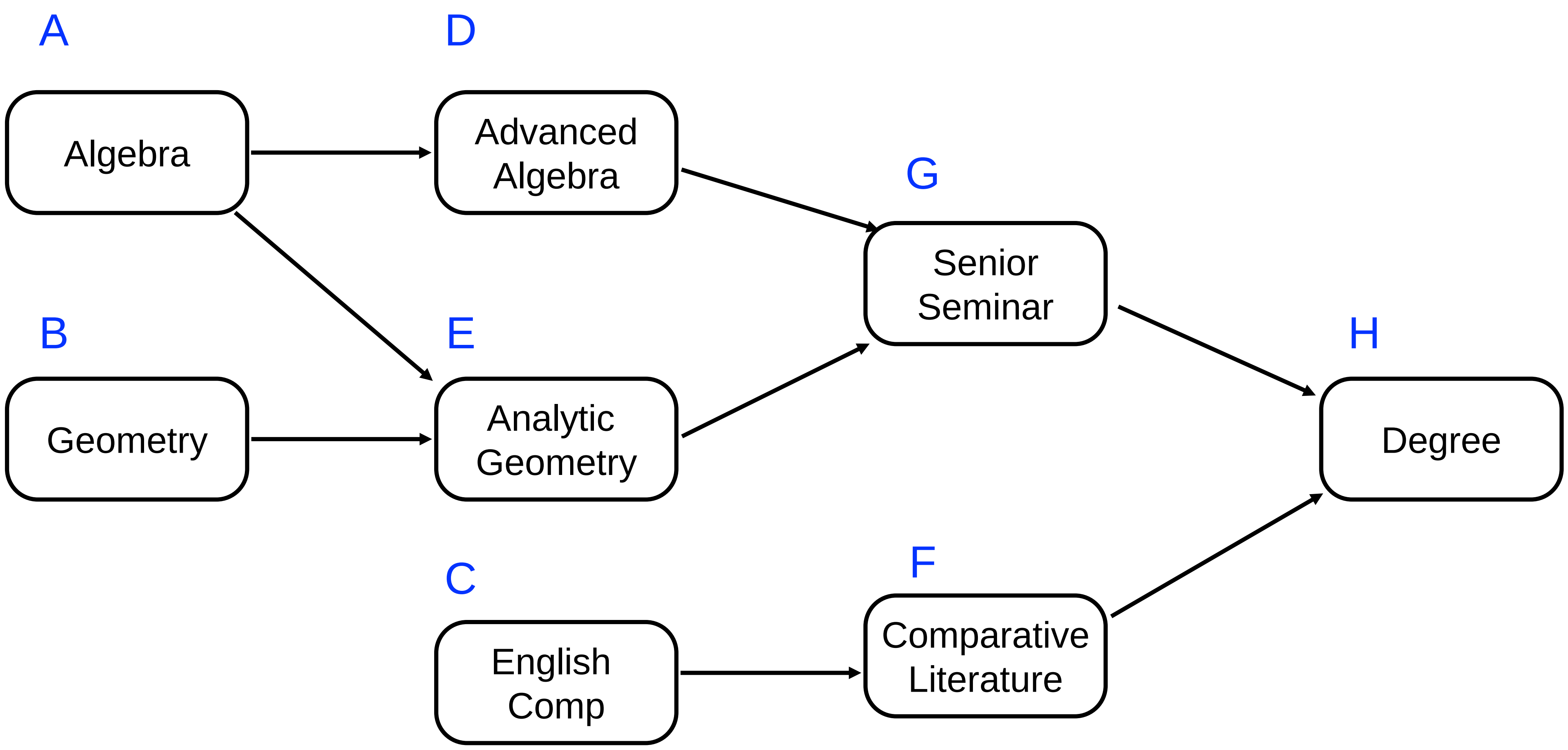
# Topological Sorting



# Topological Sorting

- You can use topological sorting to list the courses in the order you need to take them
- Arranged this way, the graph is called **topological sorted**
- For example,

# Topological Sorting



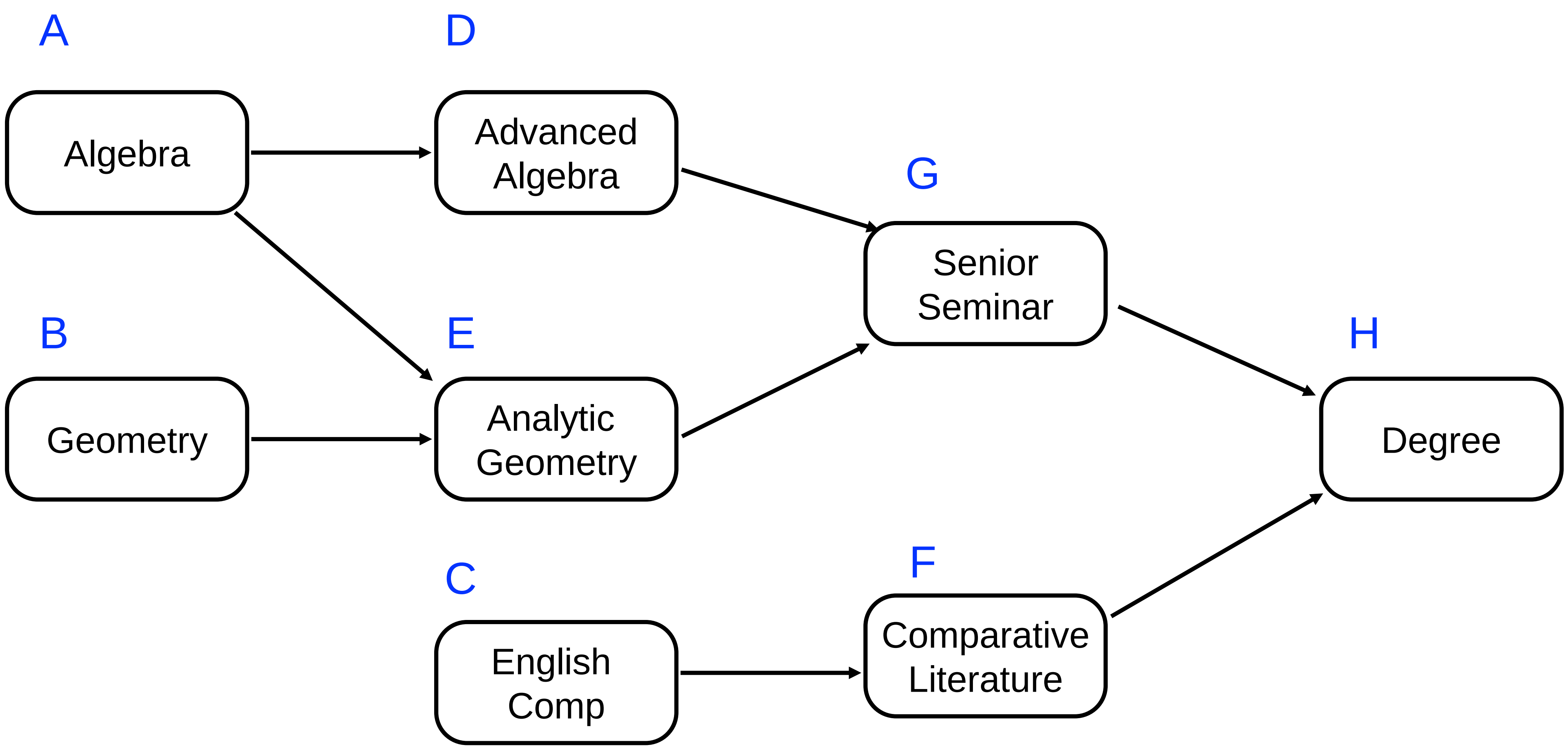

BAEDGCFH

# Topological Sorting

- Many possible ordering would satisfy the course prerequisites

- For example,

# Topological Sorting




CFBAEDGH

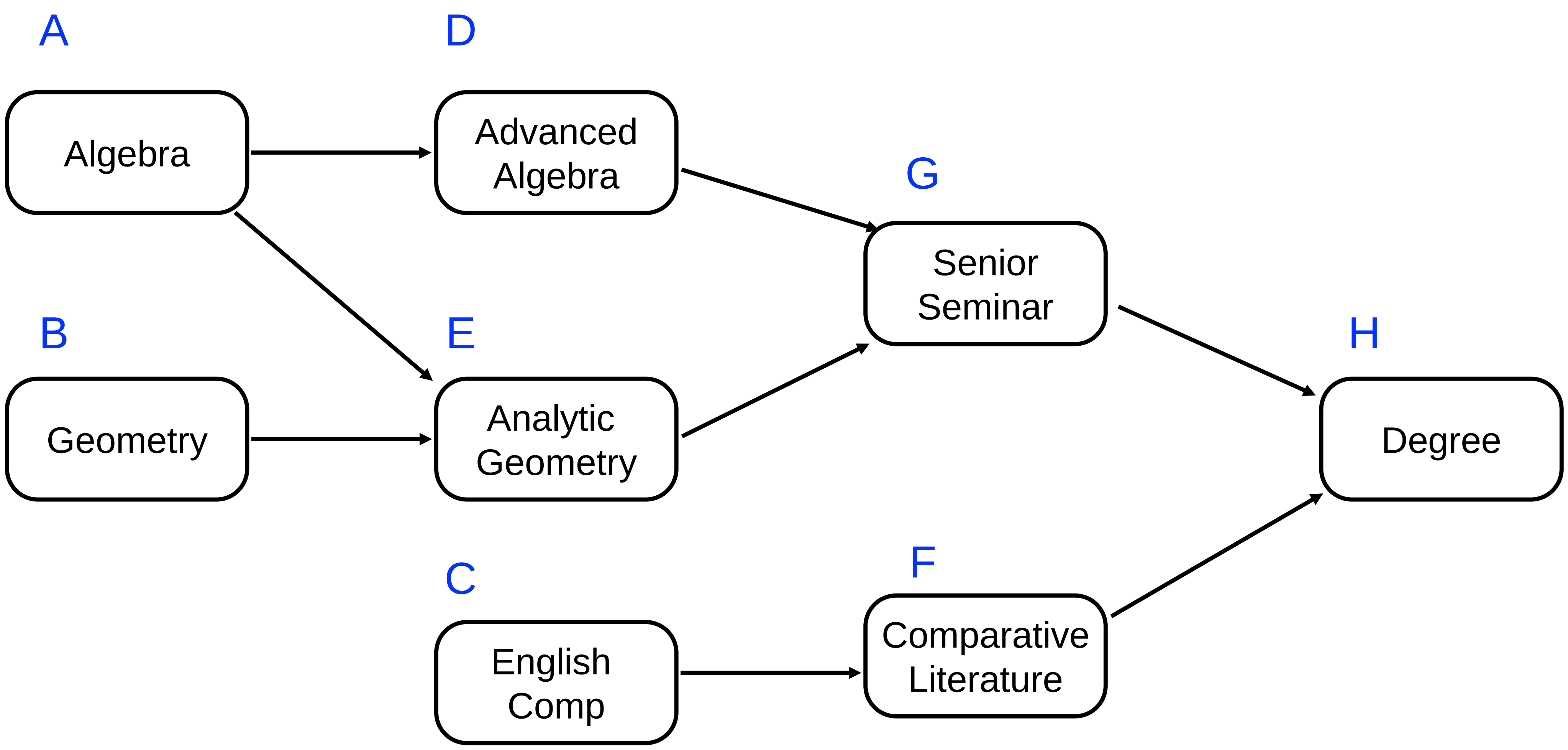
# Topological Sorting

- There are many other possible orderings as well
- When you use an algorithm to generate a topological sort, the approach you take and the details of the code determine which of various valid sorting are generated
- Topological sorting can be used to model job scheduling as well

# Topological Sorting

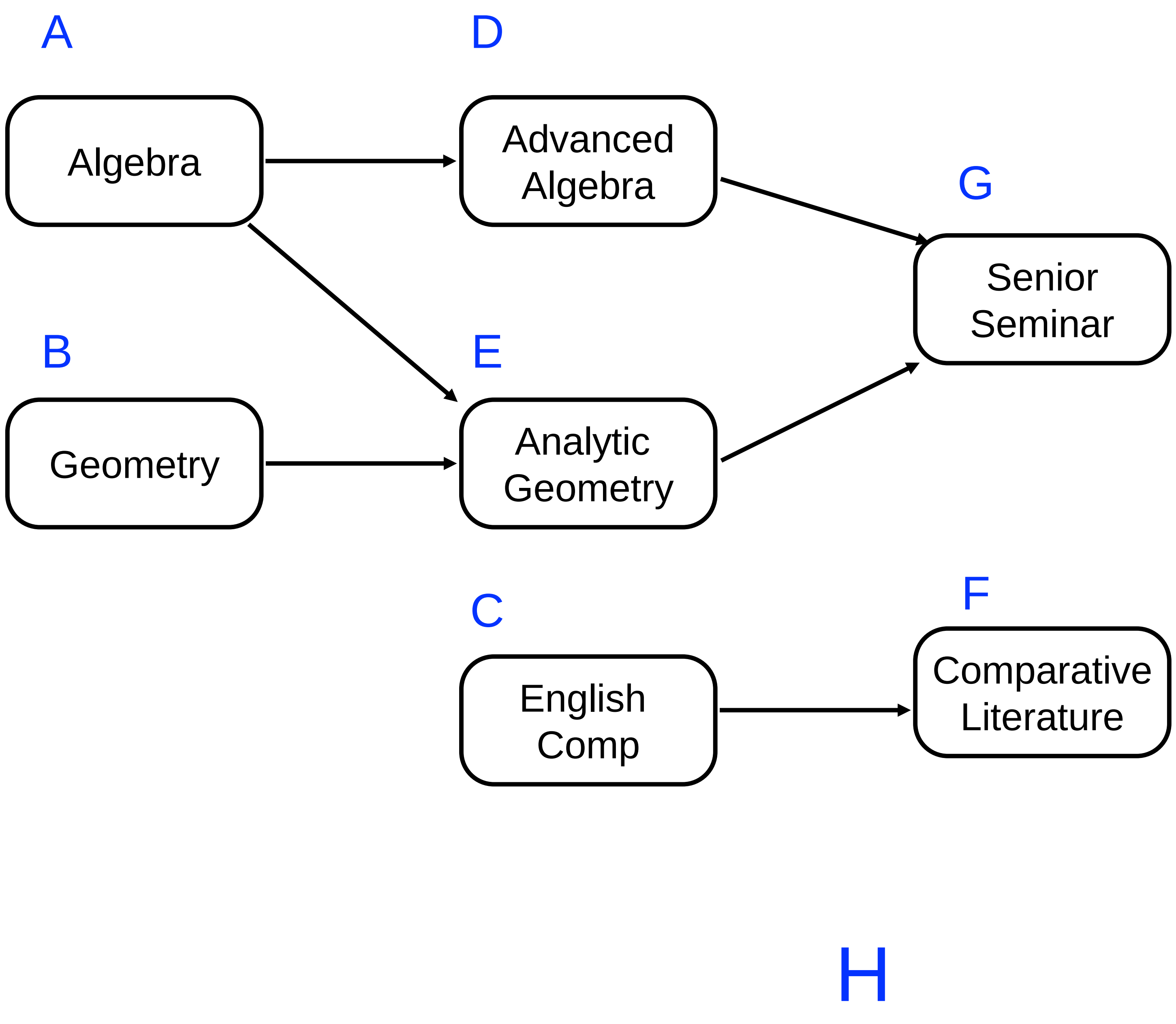
- A simple algorithm
- Uses a list
- Given a DAG, take the following steps
  - Step 1: Find a vertex that has no successors
  - Step 2; Delete this vertex from the graph, and insert its label at the beginning of the list
- Repeat Step 1 and Step 2, until all the vertices are gone
- The list shows the vertices arranged in topological order

# Topological Sorting

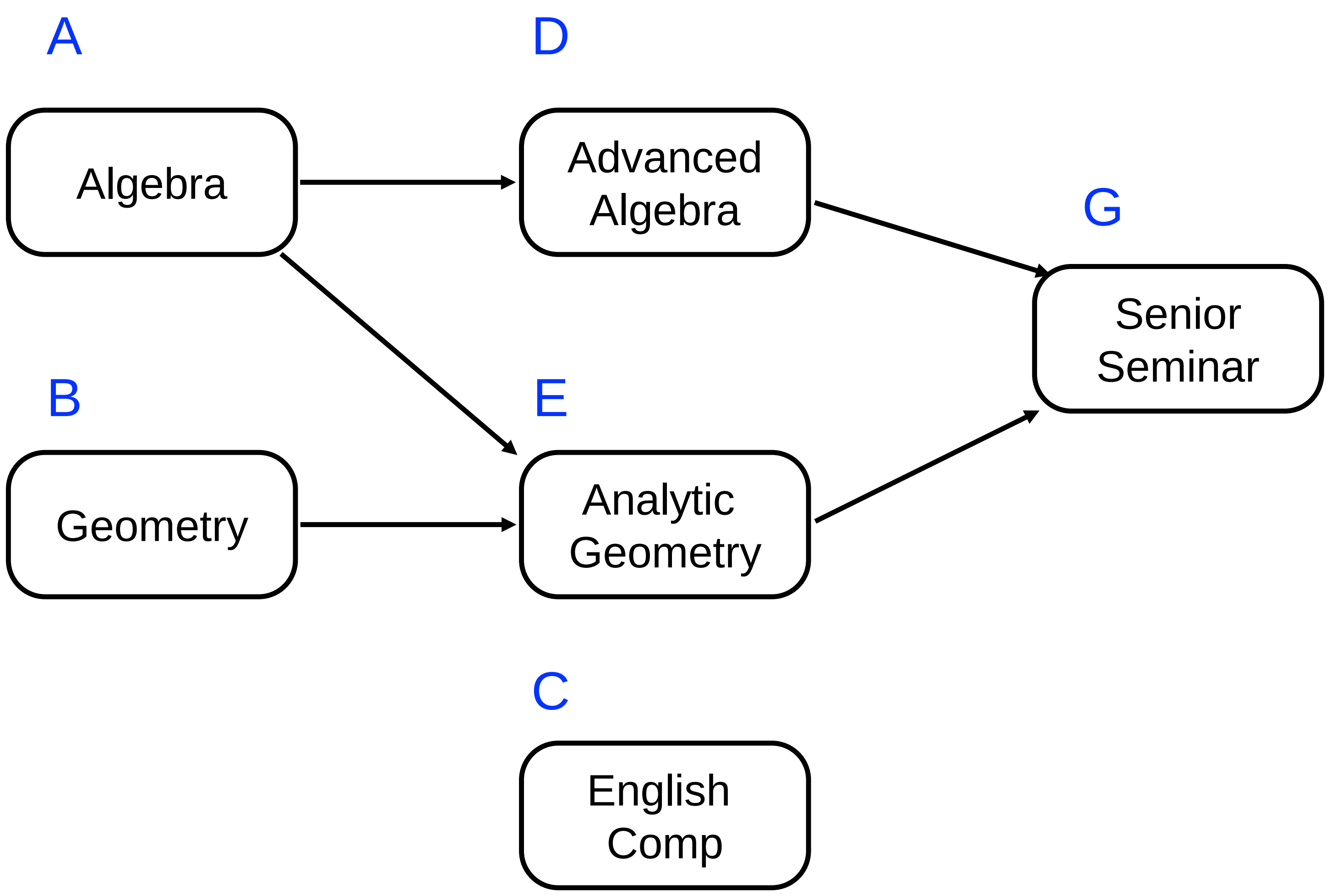



Which vertex has no successors?

# Topological Sorting

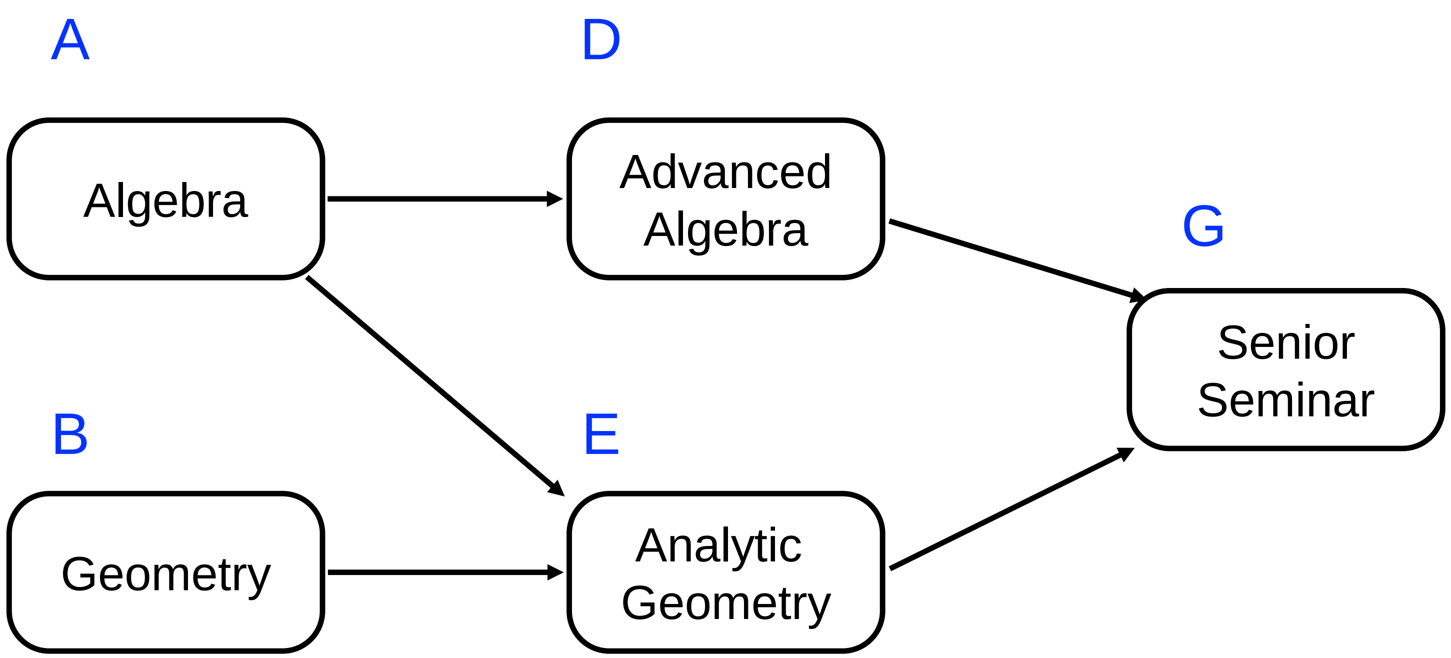


# Topological Sorting



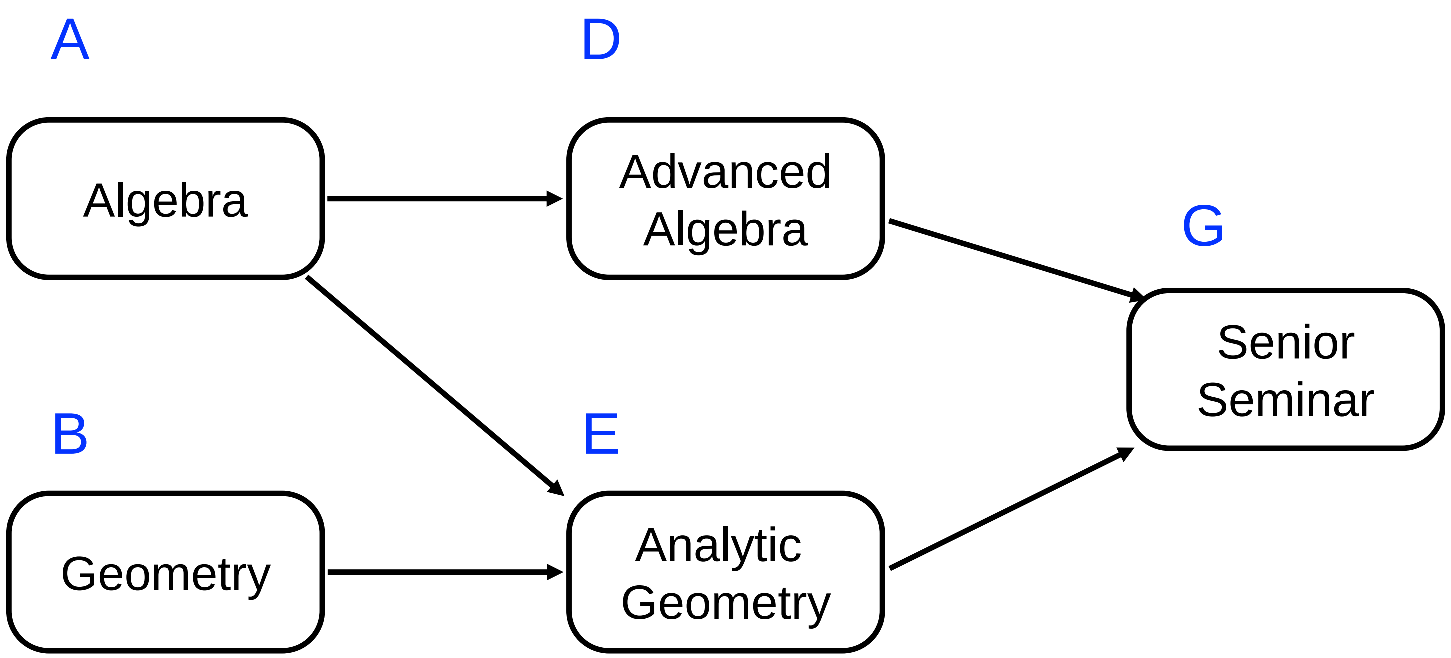

FH

# Topological Sorting



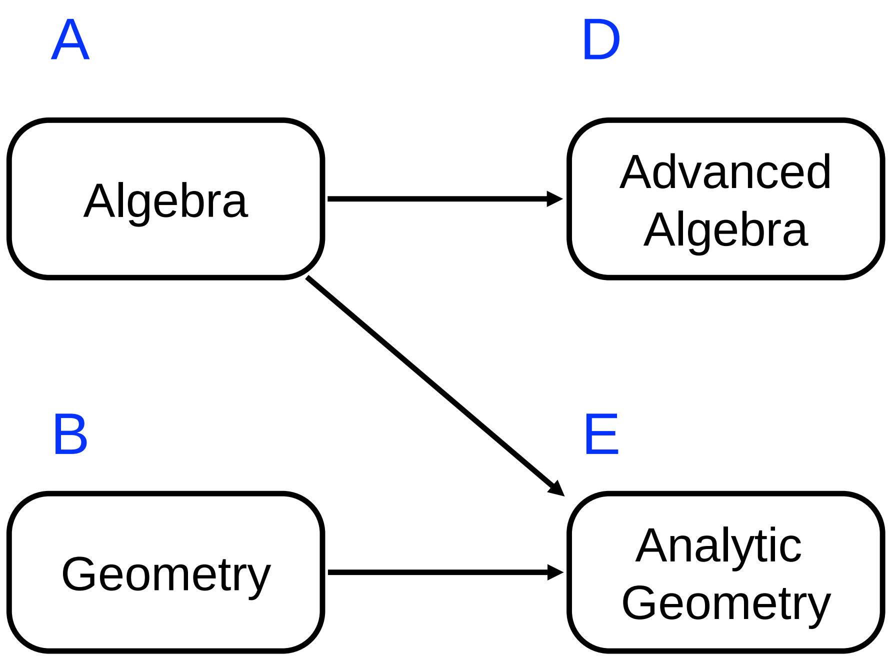

CFH

# Topological Sorting



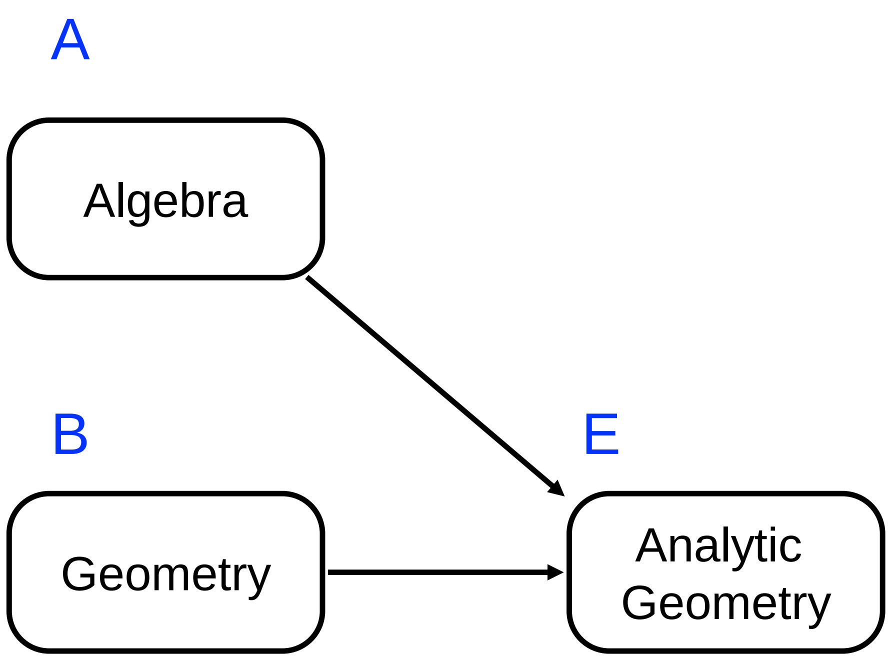

CFH

# Topological Sorting




GCFH

# Topological Sorting




DGCFH

# Topological Sorting

A

Algebra

B

Geometry

EDGCFH

# Topological Sorting

B

Geometry

AEDGCFH

# Topological Sorting

BAEDGCFH

Topological Sorted Graph

# Topological Sorting

- Note: one kind of graph that the topological sort algorithm cannot handle is a graph with **cycles**

  - A path that ends where it started

# Topological Sorting

```
public void topo()                  // toplogical sort
   {

      int orig_nVerts = nVerts;  // remember how many verts

      while(nVerts > 0)          // while vertices remain,
         {
         // get a vertex with no successors, or -1
         int currentVertex = noSuccessors();
         if(currentVertex == -1)       // must be a cycle
            {
            System.out.println("ERROR: Graph has cycles");
            return;
            }
```

# Topological Sorting

```
   // insert vertex label in sorted array (start at end)
   sortedArray[nVerts-1] = vertexList[currentVertex].label;

   deleteVertex(currentVertex);  // delete vertex
   }  // end while

// vertices all gone; display sortedArray
System.out.print("Topologically sorted order: ");
for(int j=0; j<orig_nVerts; j++)
   System.out.print( sortedArray[j] );
System.out.println("");

}  // end topo
```

# Topological Sorting

```
public int noSuccessors()  // returns vert with no successors
   {                       // (or -1 if no such verts)
   boolean isEdge;  // edge from row to column in adjMat

   for(int row=0; row<nVerts; row++)  // for each vertex,
      {
      isEdge = false;                 // check edges
      for(int col=0; col<nVerts; col++)
         {
         if( adjMat[row][col] > 0 )   // if edge to
            {                         // another,
            isEdge = true;
            break;                    // this vertex
            }                         //    has a successor
         }                            //    try another
      if( !isEdge )                   // if no edges,
         return row;                  //    has no successors
      }
   return -1;                         // no such vertex

   }  // end noSuccessors()
```

# P vs. NP

# Problem Complexity

- Some problems are easy to solve, whereas other are extremely hard to solve

- For example: Multiplication vs. Factoring

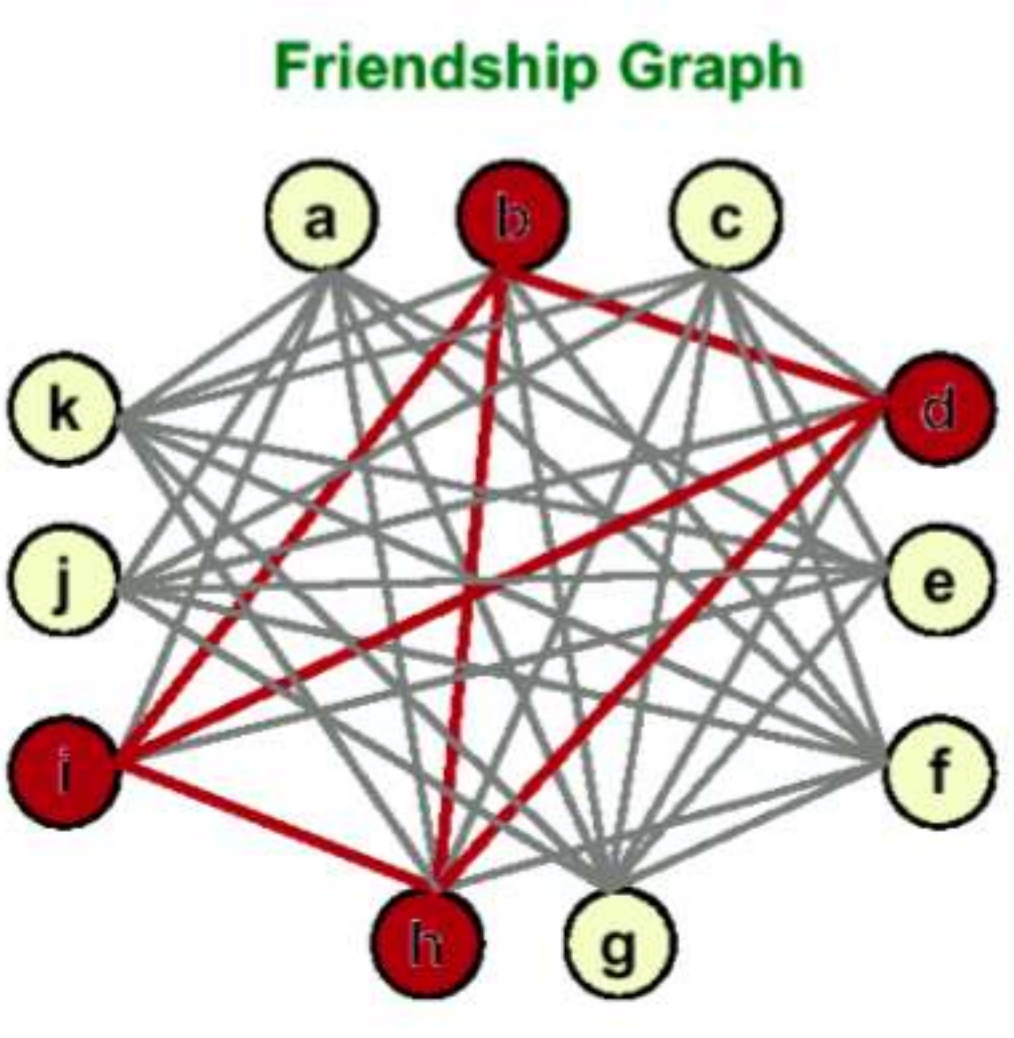
9 x 13 = ?  vs.  ? x ? = 91

# Why is factoring a hard problem?

- Unlike multiplication, where we can zoom in onto the answer
- All we can do for factoring is to search for the answer
- Brute Force Search
  - Divide by 2, 3, 5, 7, … until find a factor
  - Very slow when the search space is huge

# Clique Example

- Given n people and their pairwise relationships, is there a group of s people such that every pair in the group knows each other
  - people: a, b, c, ..., k
  - friendships: (a,e), (a,f),...
  - clique size: s = 4
  - YES,{b,d,i,h} – a **certificate**



# Bigger Clique Problems

- Finding the largest cliques in 100s of nodes
- can take centuries by searching

# Other Problem that Might Require Searching

- Scheduling
- Map coloring
- Travelling salesman
- Rule generation
- Tons of more examples…

# Is Searching Necessary?

- Michael Sipser (MIT) refers to it as the “Needle in a Haystack Problem”

Is searching necessary?

# Is Searching Necessary?

- Michael Sipser (MIT) refers to it as the “Needle in a Haystack Problem”

Is searching necessary?

Not, if you have a magnet!

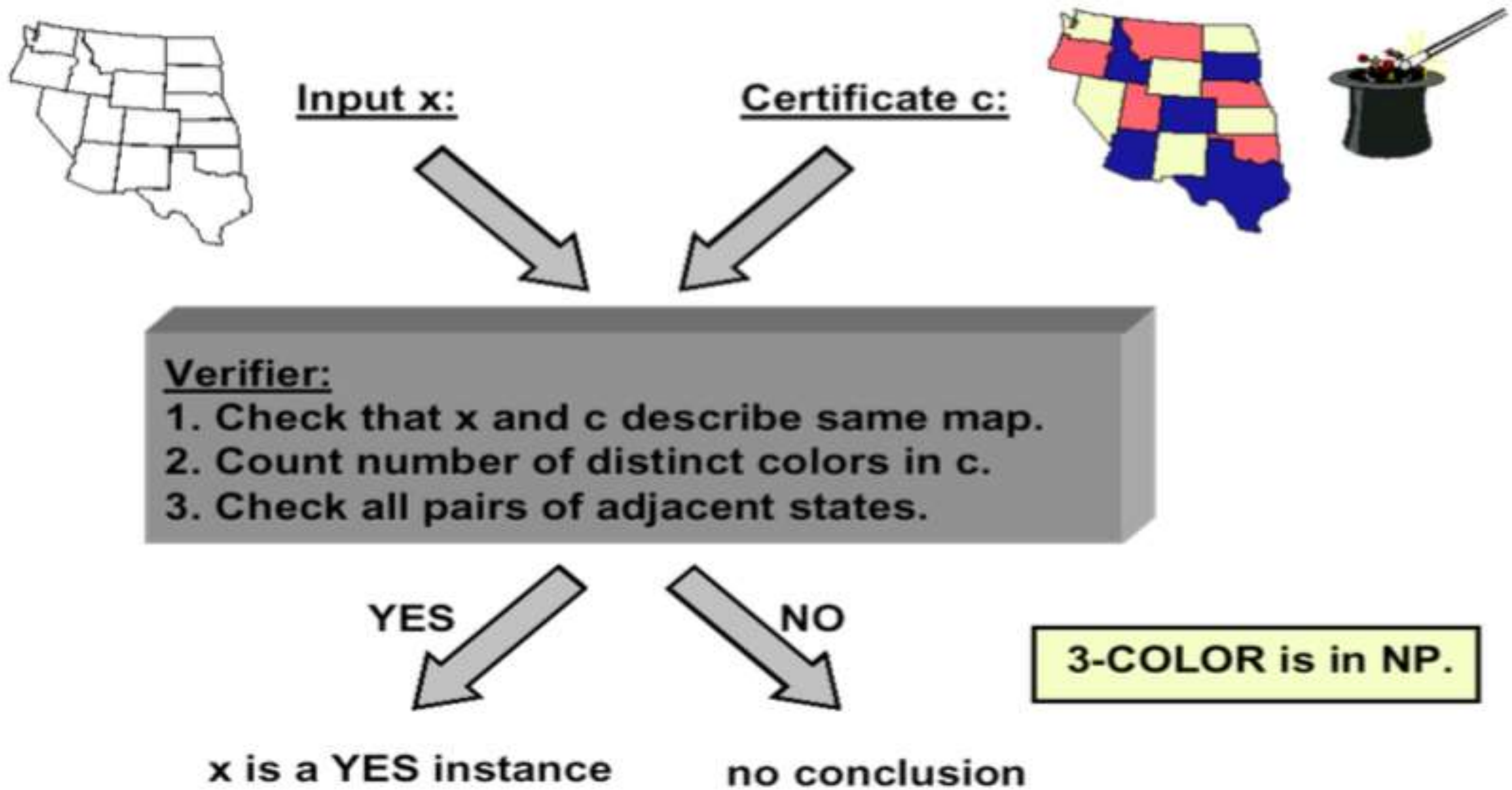
# P vs. NP Question

- Can we solve search problems without searching?

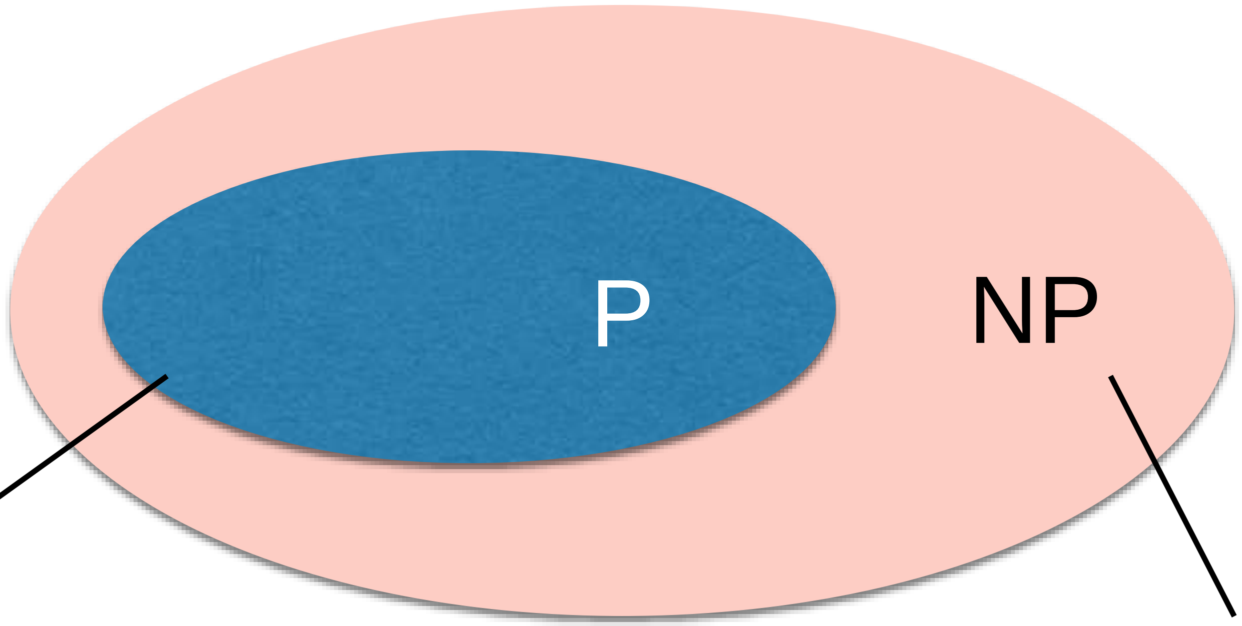
# P and NP

- P – Polynomial Time
- Quickly solvable problems
- NP – Nondeterministic Polynomial Time
- Quickly checkable problems

# Certificates



# The P and NP Classes



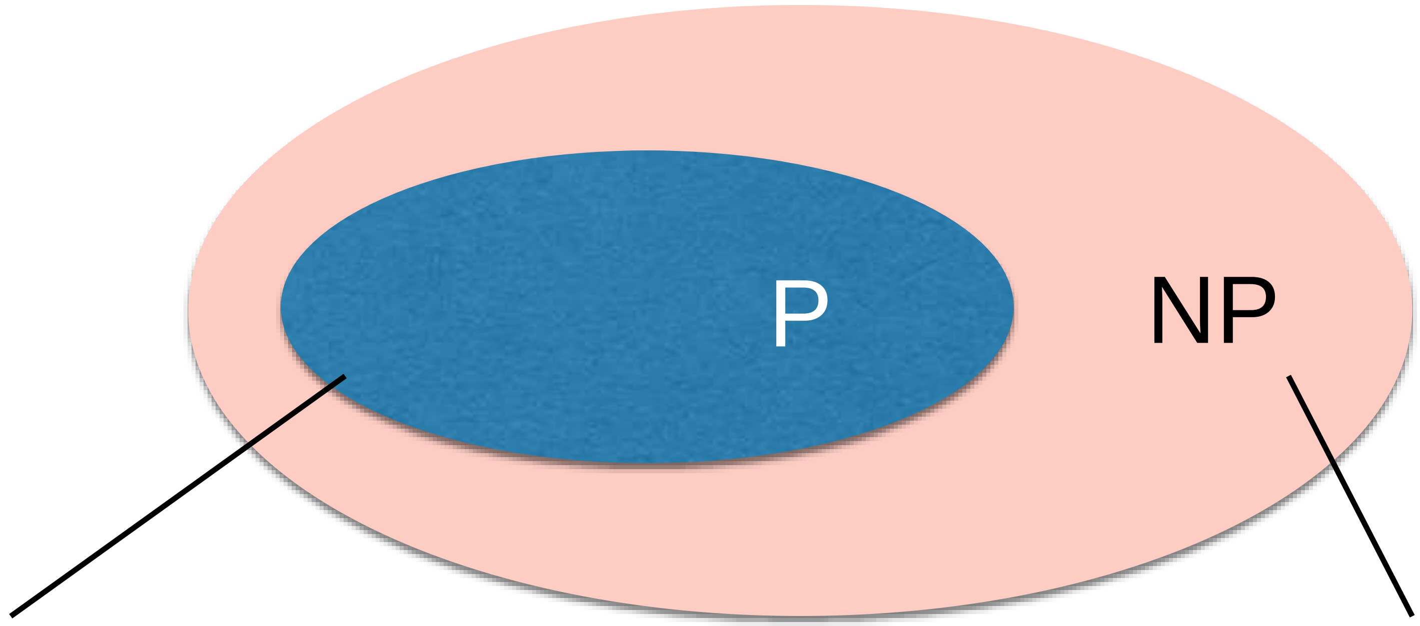

Easily solvable problems:

- Multiplication
- Sorting

Easily checkable problems:

- Factoring
- Clique

# The P and NP Classes




Easily solvable problems:

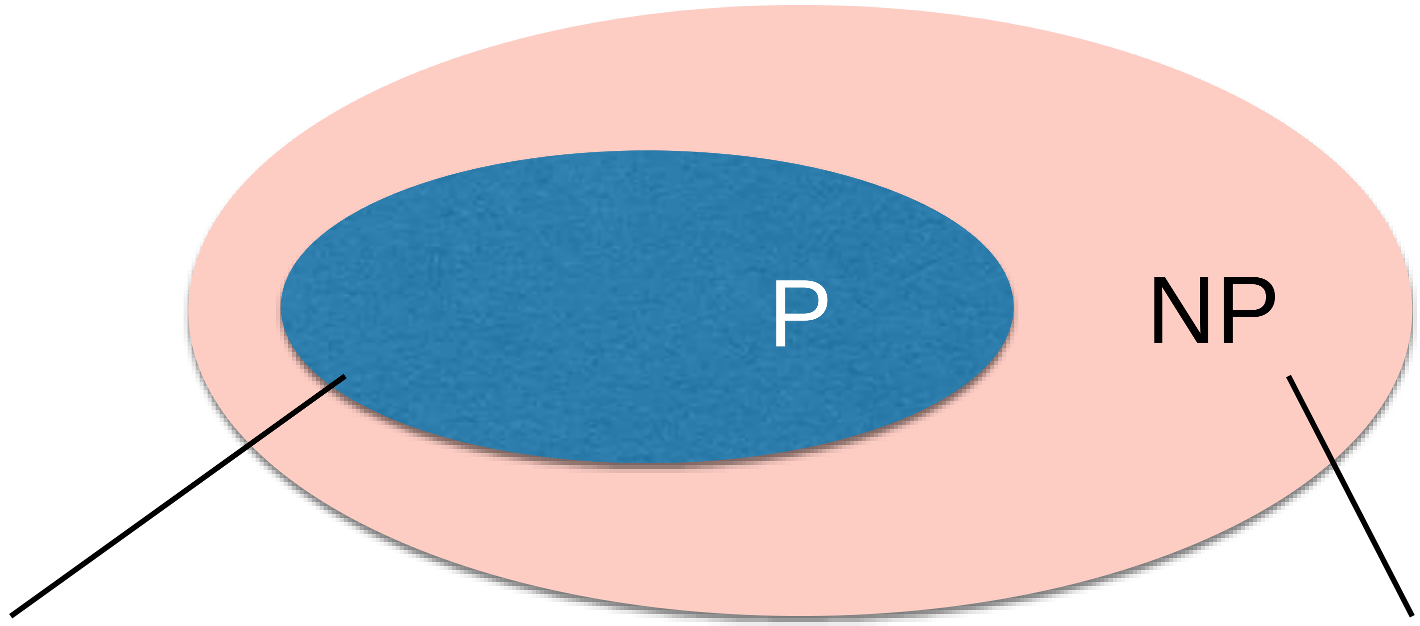
- Multiplication
- Sorting

Easily checkable problems:

- Factoring
- Clique

P = NP
or
P != NP

# The P and NP Classes




Easily solvable problems:

- Multiplication
- Sorting

Easily checkable problems:

- Factoring
- Clique

P = NP
or
P != NP

Unknown!

# Final Remark

- NP Complete – “Hardest Computational Problems in NP”

  NP → Transform → NP- Complete

  For example:

  Factoring Problem → Transform → Clique

  “Thus if it can be solved efficiently, then any other problem in NP can be solved efficiently”