# Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University
a.khan@innopolis.ru

# Recap

- BFS
- DFS
- $O(|V| + |E|)$
- BSF and DFS Trees

# Objectives

1. Optimization Problems
2. Greedy Algorithms
3. Minimum Spanning Tree
   - MST in Unweighted Graphs
   - MST in Weighted Graphs (Prim's Algorithm, and Krushkal's Algorithm)

# Optimization Problems

# Optimization Problems

- Many important problems that we deal in real-life are *optimization problems*

- Important thing to remember about such problems

  - *There exist many possible solutions*

  - *The goal is to find "an optimum" solution from amongst all possible solutions*

# Optimization Problems

- If $u$ and $v$ are two vertices in a graph $G$, and I ask you to find a path between them – ***not an optimization problem***

- But, if I ask you to find the/a shortest path between them – ***optimization problem***

- Clues: "smallest", "shortest" or "minimize" – "largest", "longest" or "maximize"

# Optimization Problems

- Examples
  - Rod cutting problem
  - Knapsack problem
  - Travelling salesman problem
  - Vehicle routing problem

# Greedy Algorithms

- Classic algorithmic paradigm for approaching optimization problems

- Basic structure:

  - Solve the problem as making a sequence of “moves”

  - The move that we take is the one that seems the best at the moment

  - Every time we make a move, we end up with a smaller version of the problem

# Greedy Algorithms

- Finding solutions to problem step-by-step
- A partial solution is incrementally expanded towards a 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.
- Thus, at each step the choice must be locally optimal – this is the central point of this technique

# Example: Activity Selection Problem

- Problem: Given a set $A = \{A_1, A_2, \cdots, A_n\}$ of $n$ activities with start and finish times $(s_i, f_i)$, $1 \leq i \leq n$, select maximal set $S$ of "non-overlapping" activities.

# Example: Activity Selection Problem

- Greedy solution:
    - Sort activity by finish time (let $A_1, A_2, \cdots, A_n$ be denote sorted sequence)
    - Pick first activity $A_1$
    - Remove all activities with start time before finish time of $A_1$
    - Recursively solve problem on remaining activities.

# Example: Activity Selection Problem

- Greedy solution:

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

# Example: Currency Counting

- For example, counting to a desired value using the least number of coins

- Let's say, we are given coins of value 1, 2, 5 and 10 of some currency. And the target value is 16 in that currency

- How will you proceed?

# Example: Currency Counting II

- Does not always give the optimal solution
- Let's say, a monetary system consists of only coins of worth 1, 7 and 10.
- How would a greedy approach count out the value of 15?

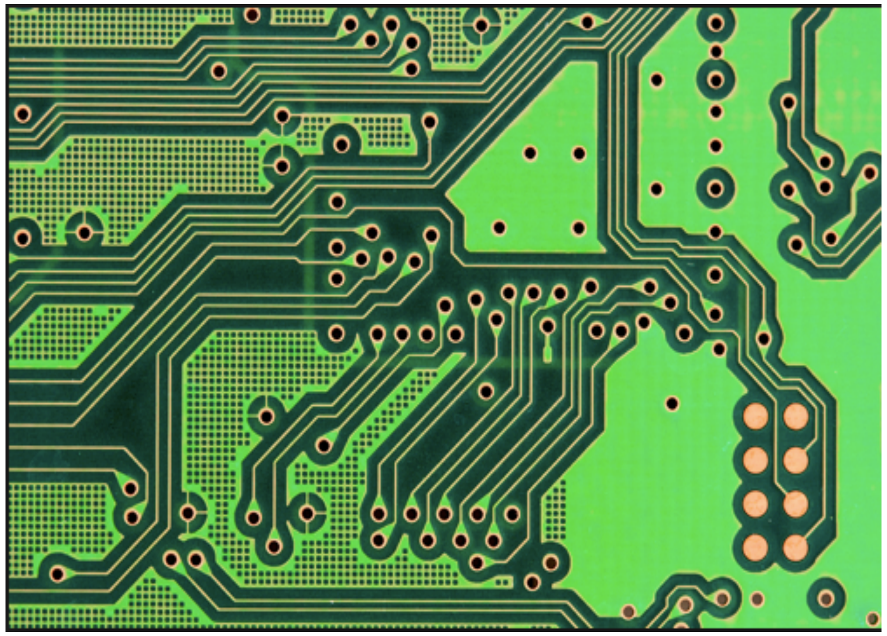
# More Details

- For more details on Greedy Approaches
- Read Chapter 16, "*Introduction to Algorithms*"

# 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

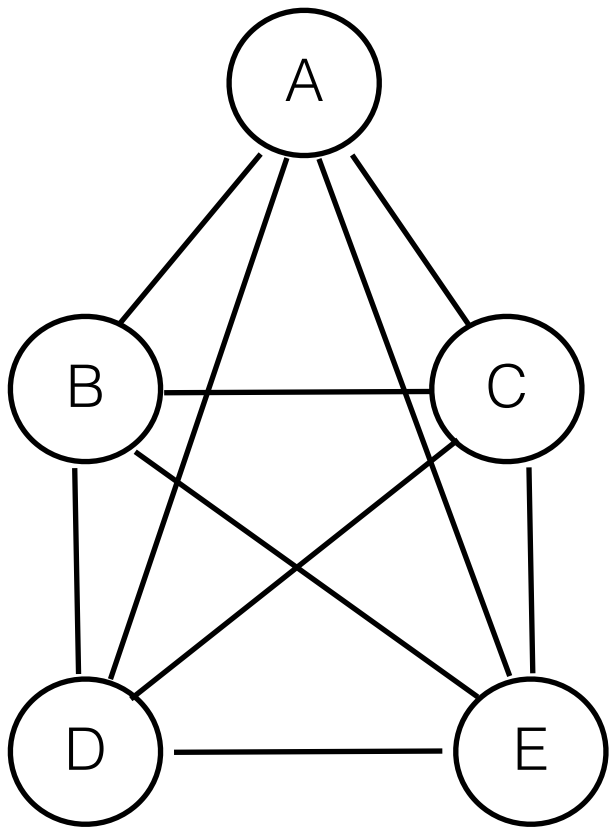
- Same question arises in many other situations:
  - Computer networks
  - Transportation networks
  - Water supply networks
  - Telecommunication networks

# Minimum Spanning Tree

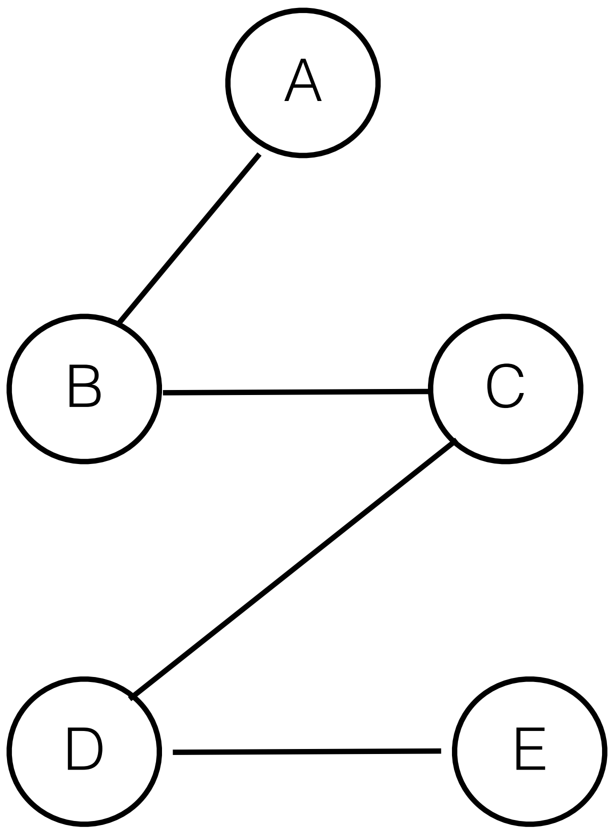
- The trick is to imagine such cases as a graph
  - ➢ Pins & Traces -> Vertices & Edges
- Then, reduce this graph such that it has the same vertices with the minimum number of edges to connect them

  **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 MST
- DFS and BFS can be used
- Execute, and record the discovery edges

# 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

- Finding MST 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

- Example:

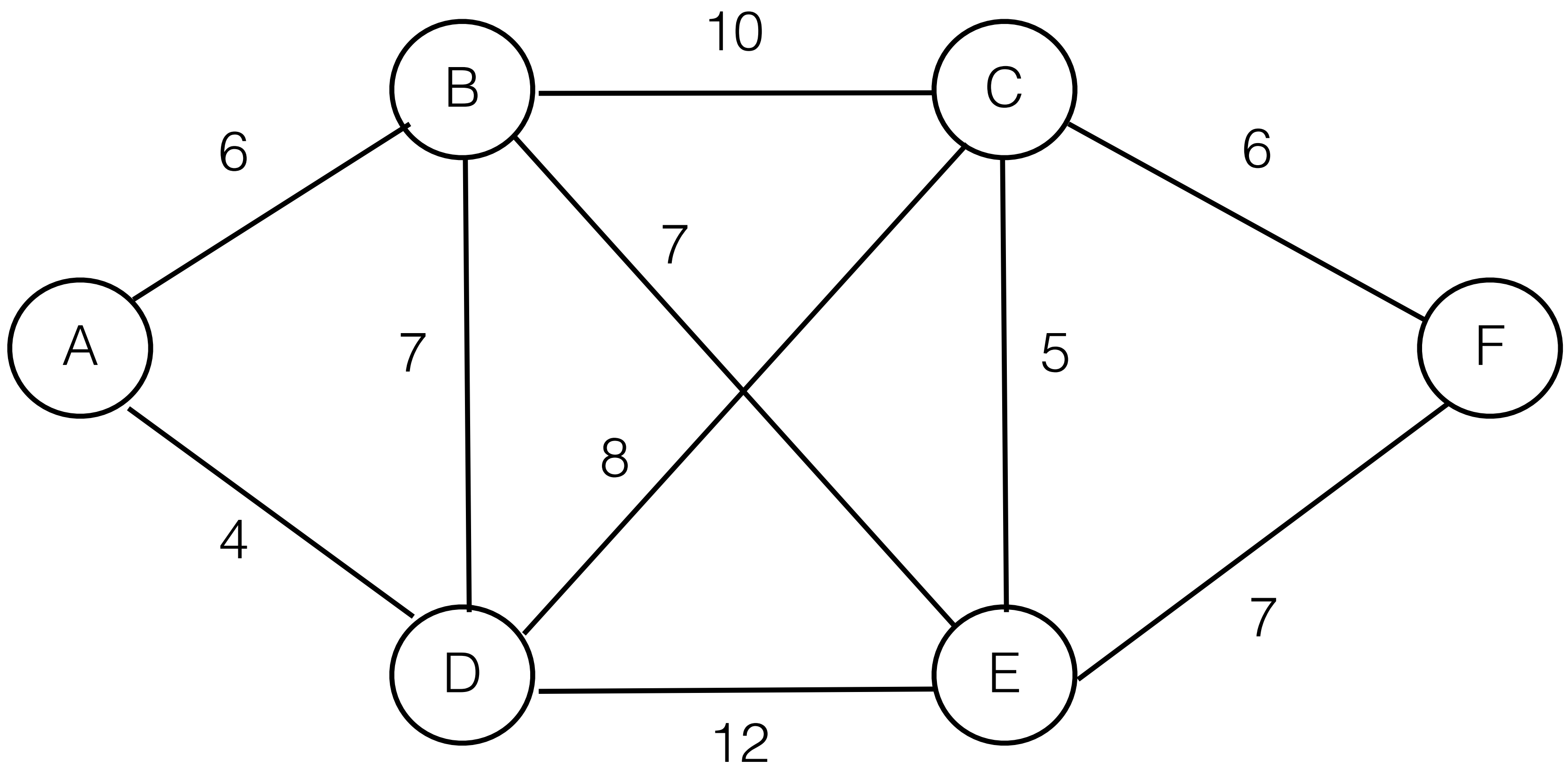
- We want to install a cable television line that connects six cities

➢ Installing cable between any pair of cities has an associated cost

➢ We want to achieve our goal with smallest cost

# MST with Weighted Graphs

Let's say we are given the following information




Weight: cost of installing cable, in million USD

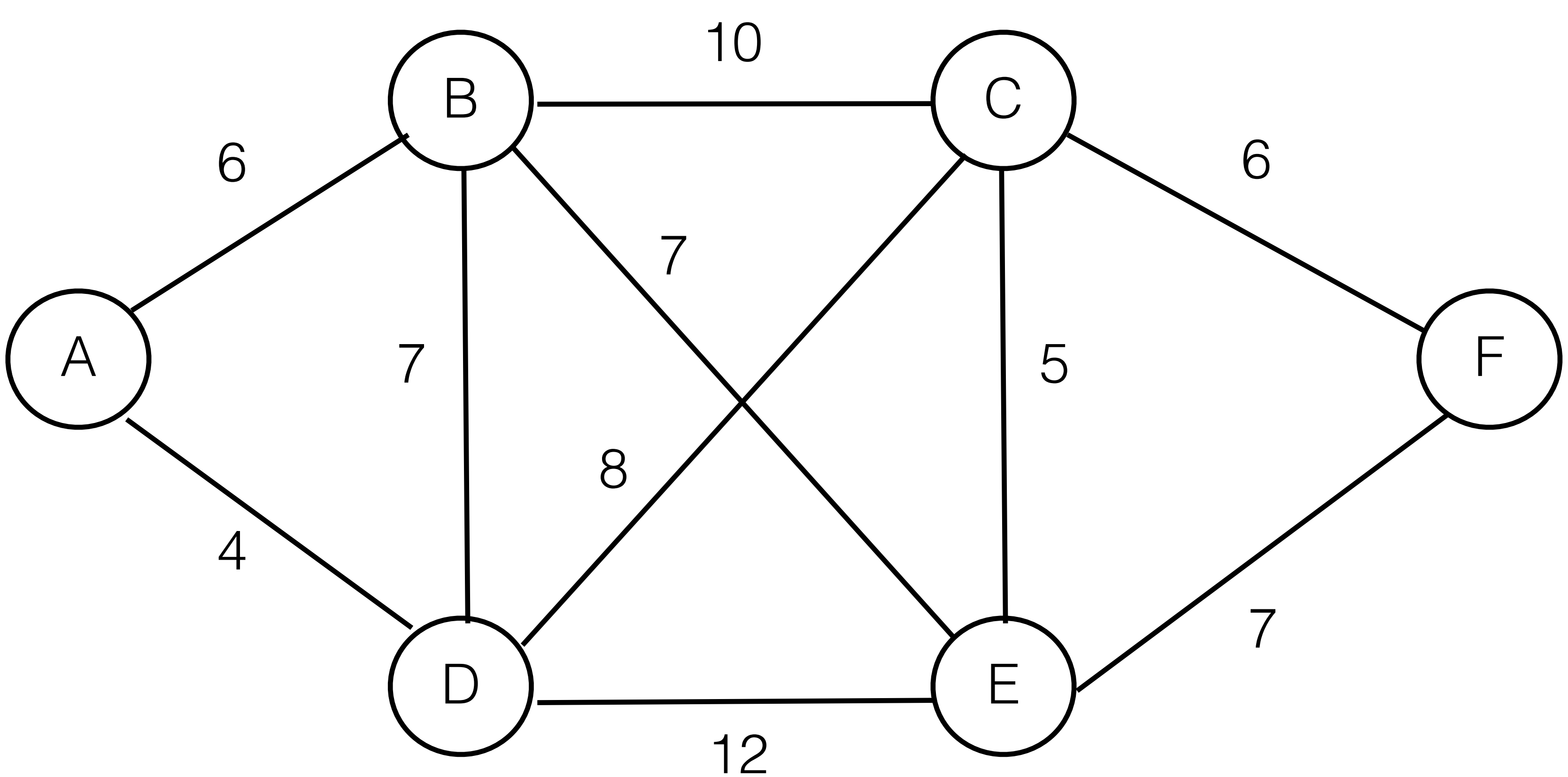
Some links are missing; let's assume that they are just too expensive that there is no point in considering them
But the algorithm will work fine even if they were present.

# 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.

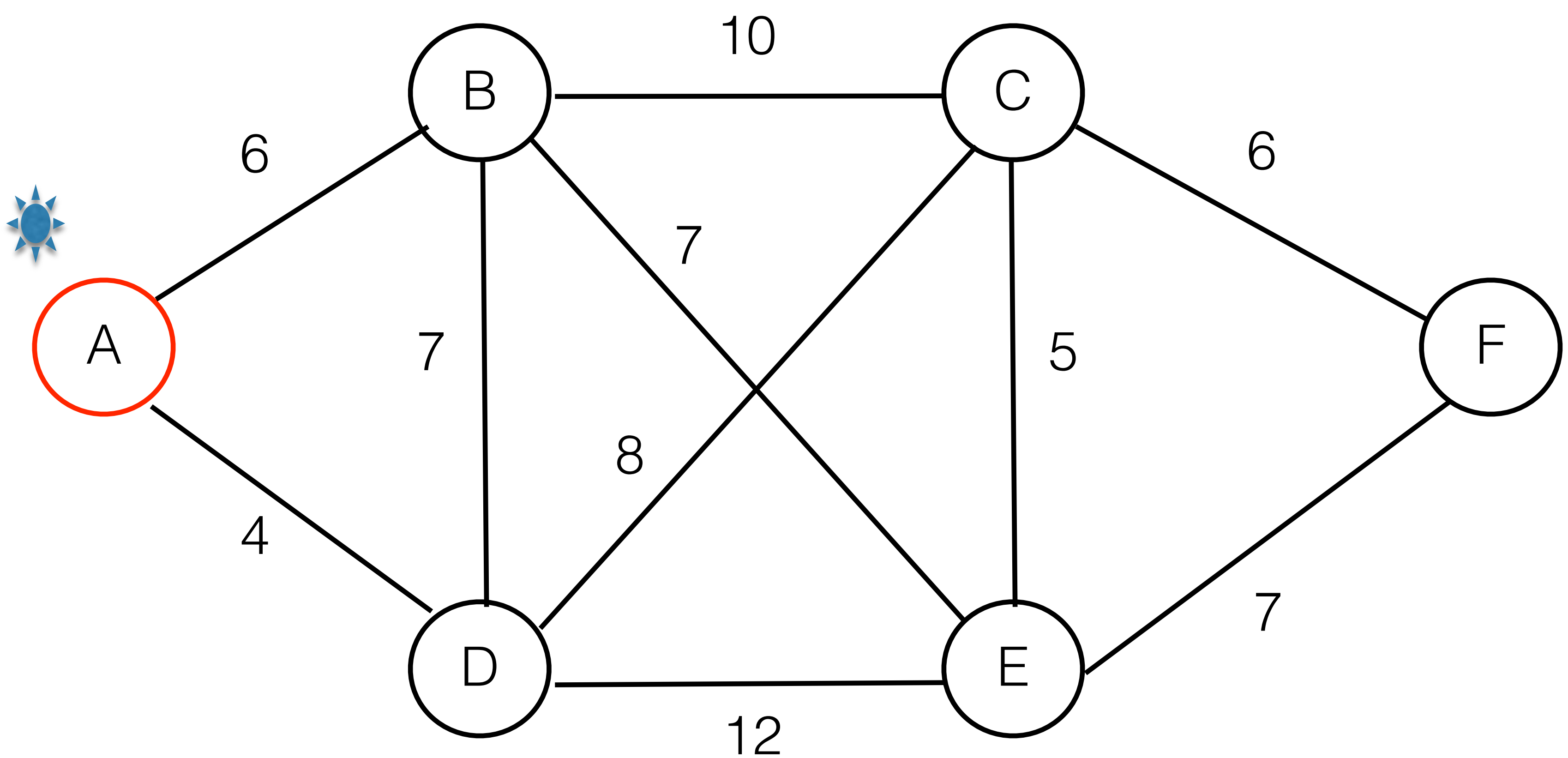
# MST with Weighted Graphs




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

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

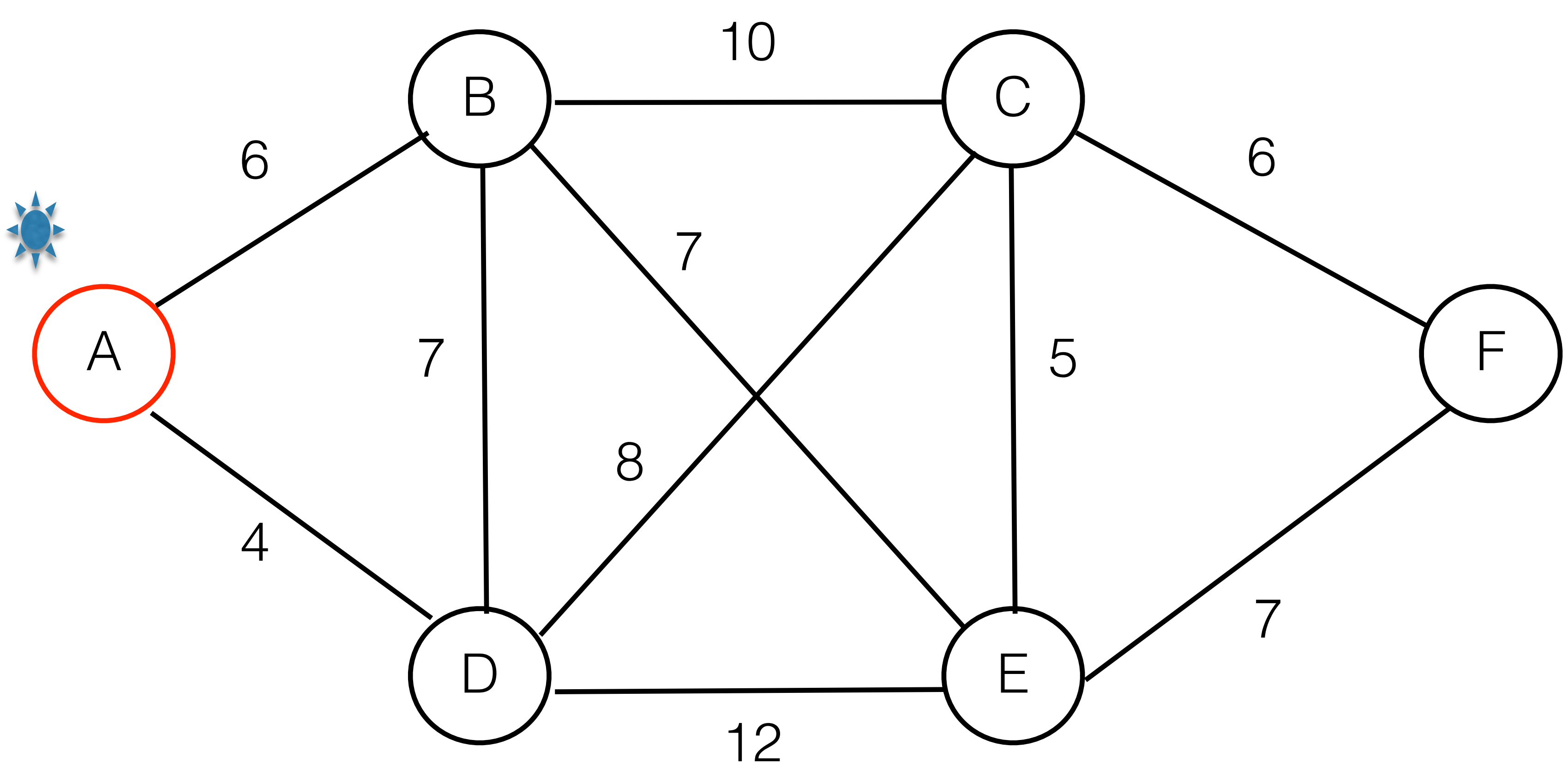
# MST with Weighted Graphs




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

Indicates where a new office has just been built!

# MST with Weighted Graphs



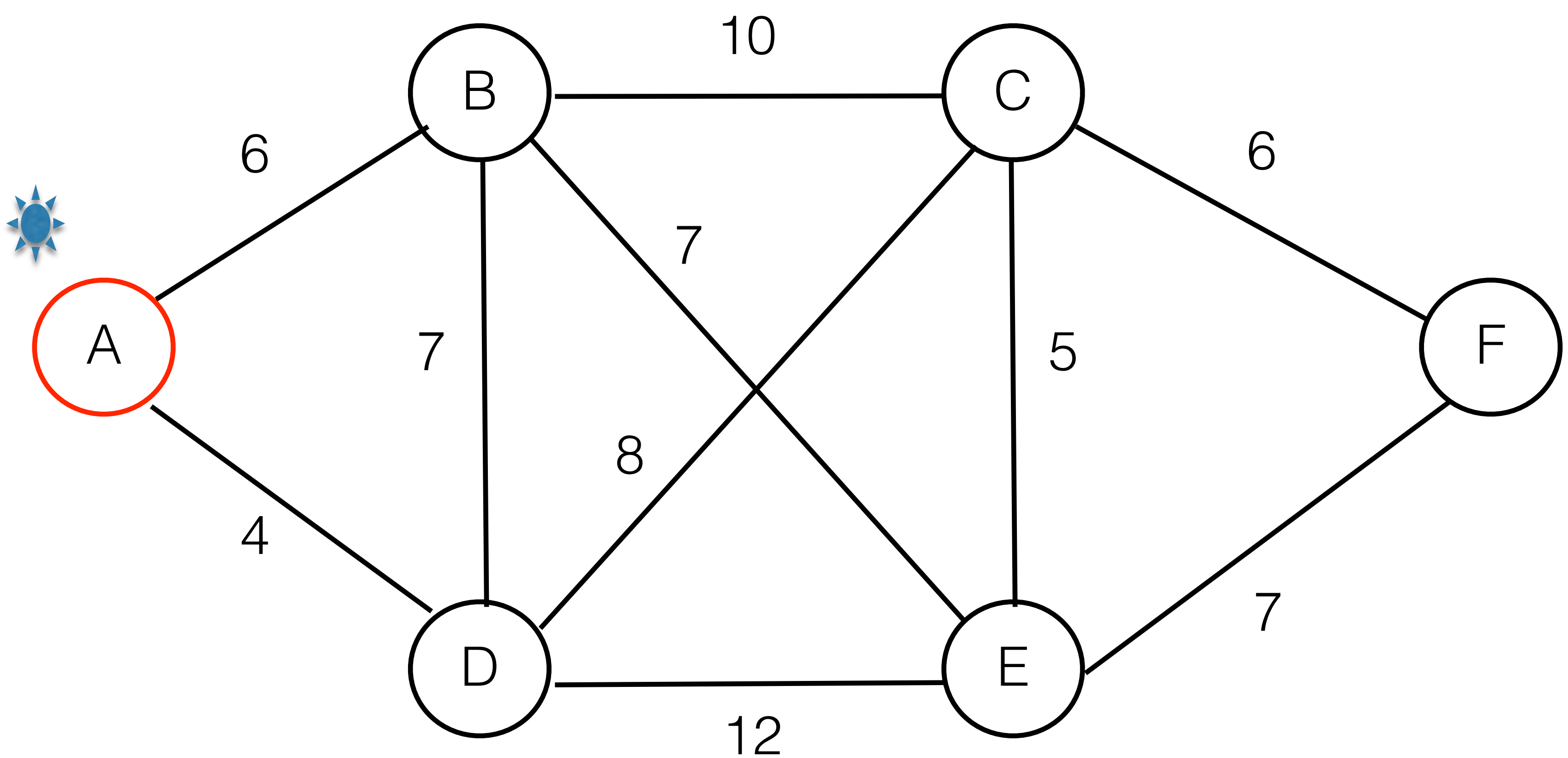

| Edge | Weight |
|---|---|
| AB | 6 |
| **AD** | **4** |
| | |
| | |

1. From the list, pick the cheapest link
2. Install it
3. Remove it from the list
4. Create an office in the new city.

# 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

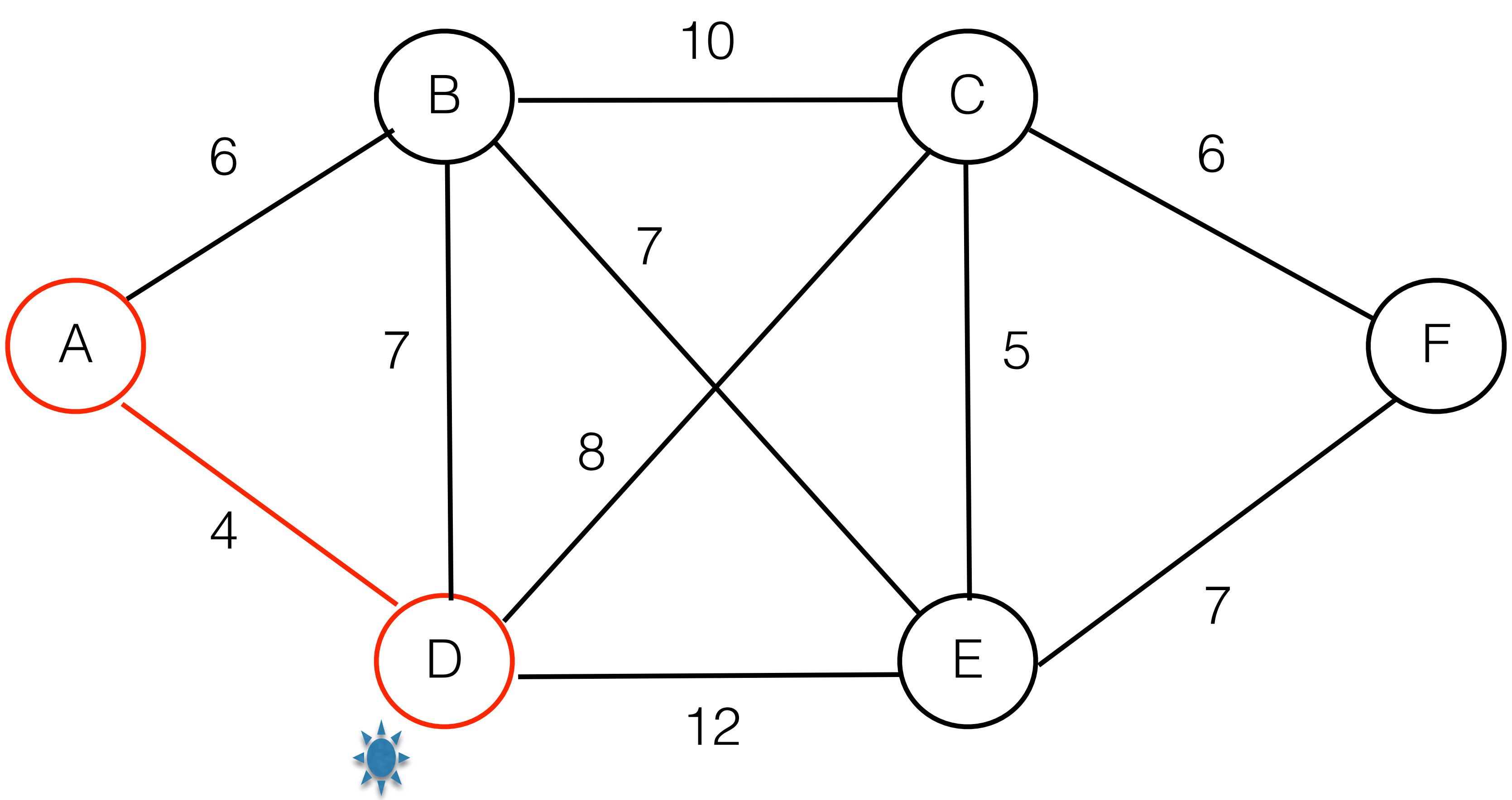



| Edge | Weight |
|---|---|
| AB | 6 |
| **AD** | **4** |
| | |
| | |

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

Greedy Algorithm

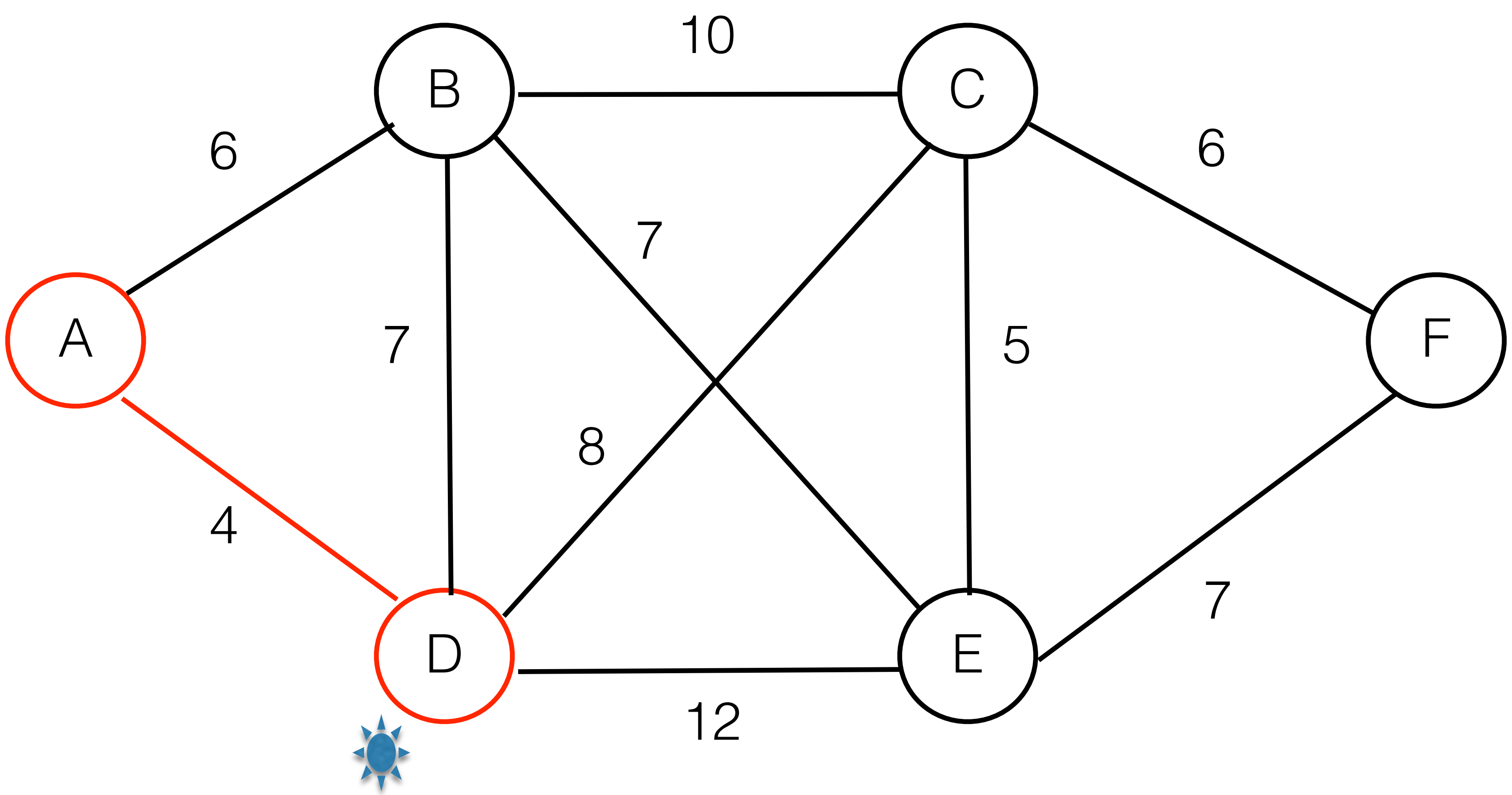
# MST with Weighted Graphs




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

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

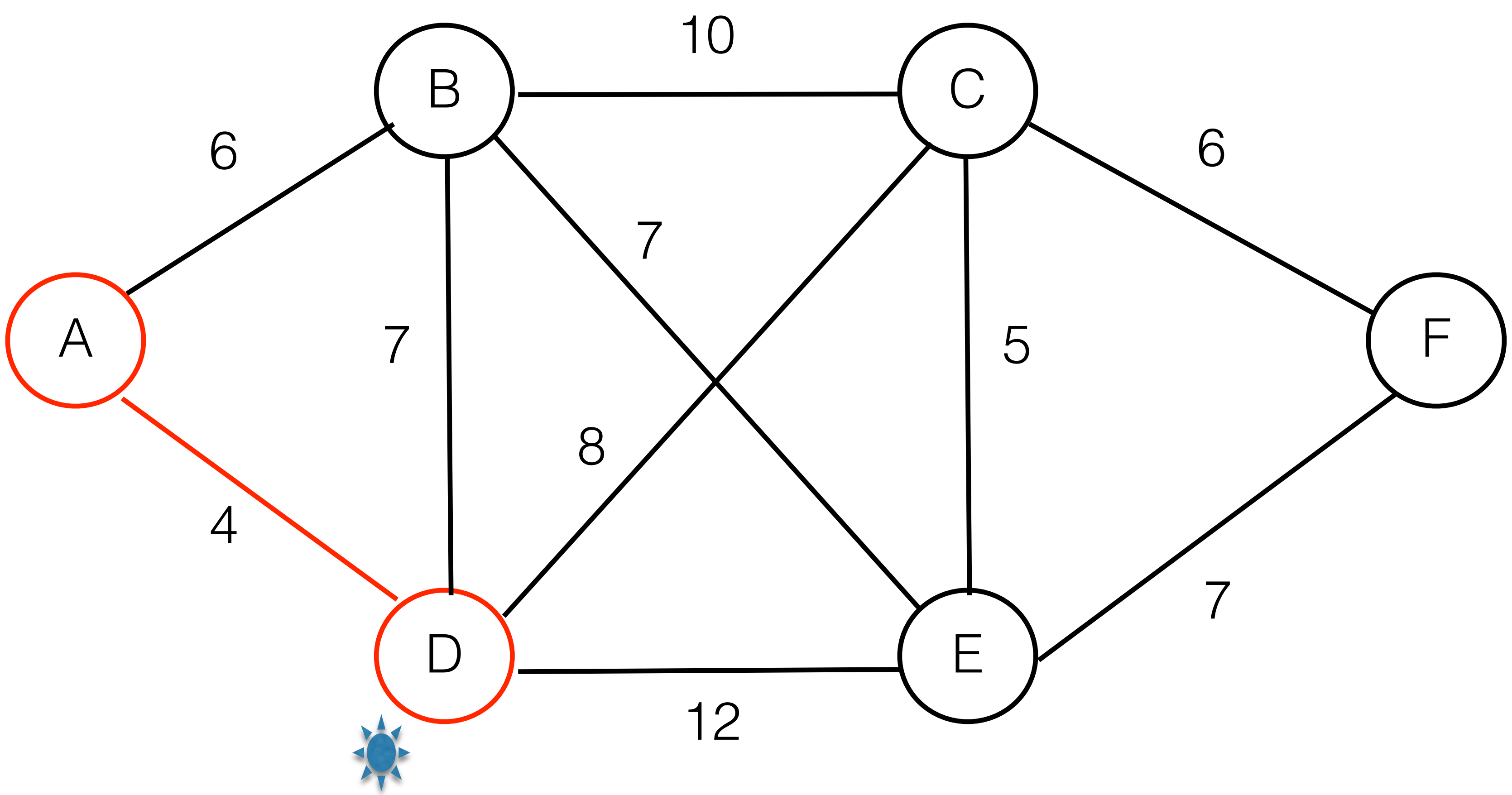
# MST with Weighted Graphs




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

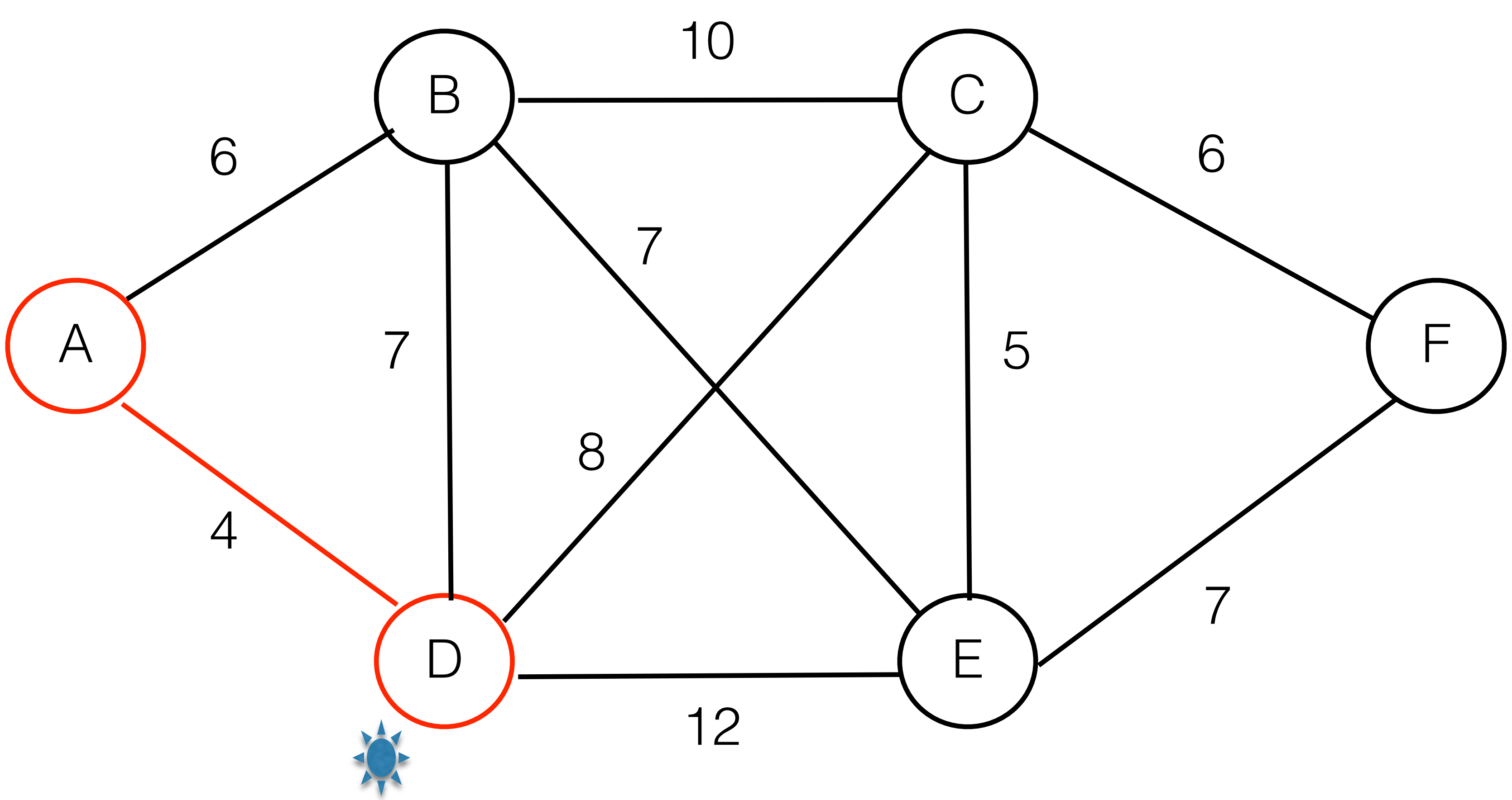
- Repeat the process for D.

# MST with Weighted Graphs




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

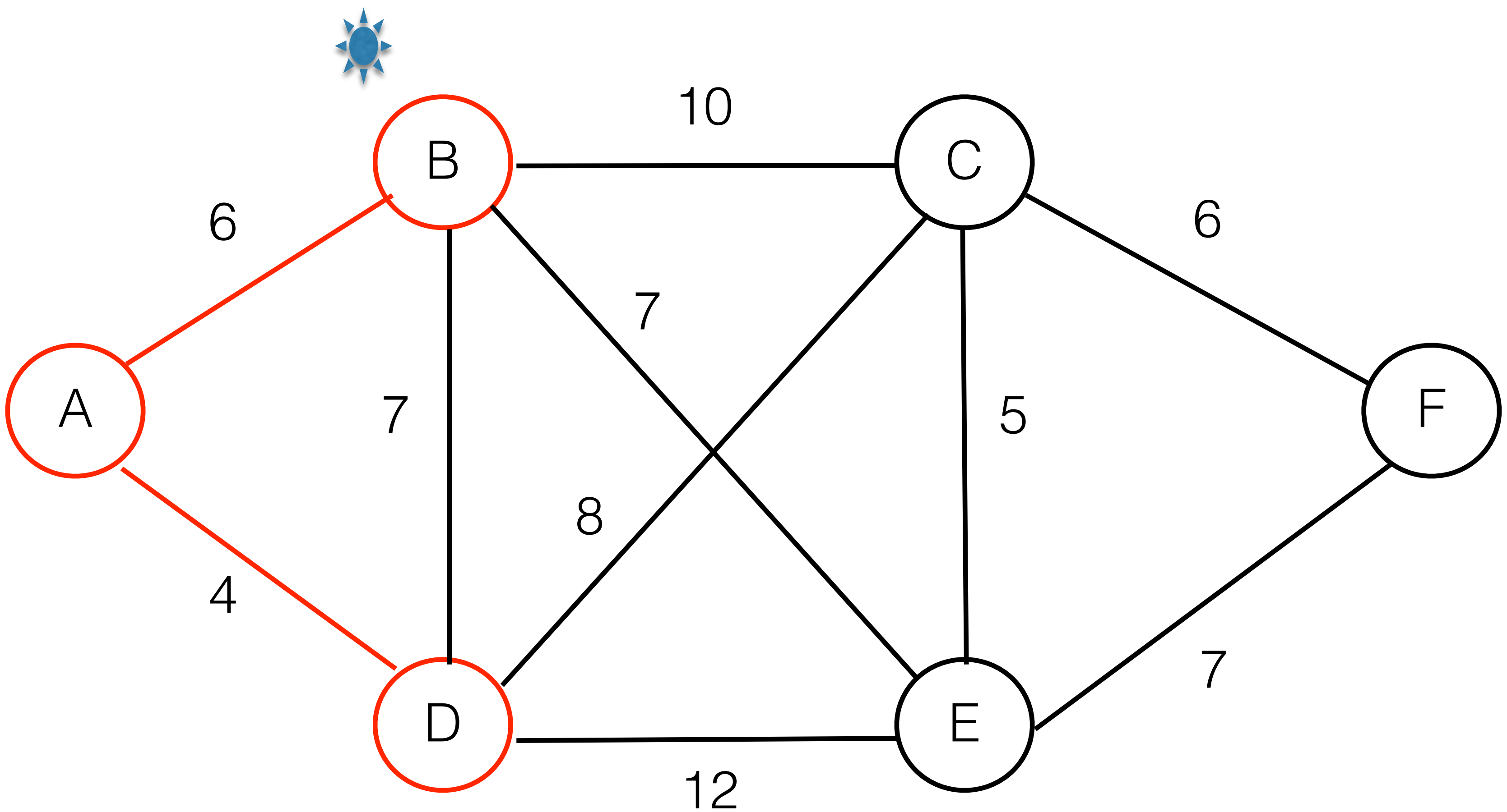
# MST with Weighted Graphs




| 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

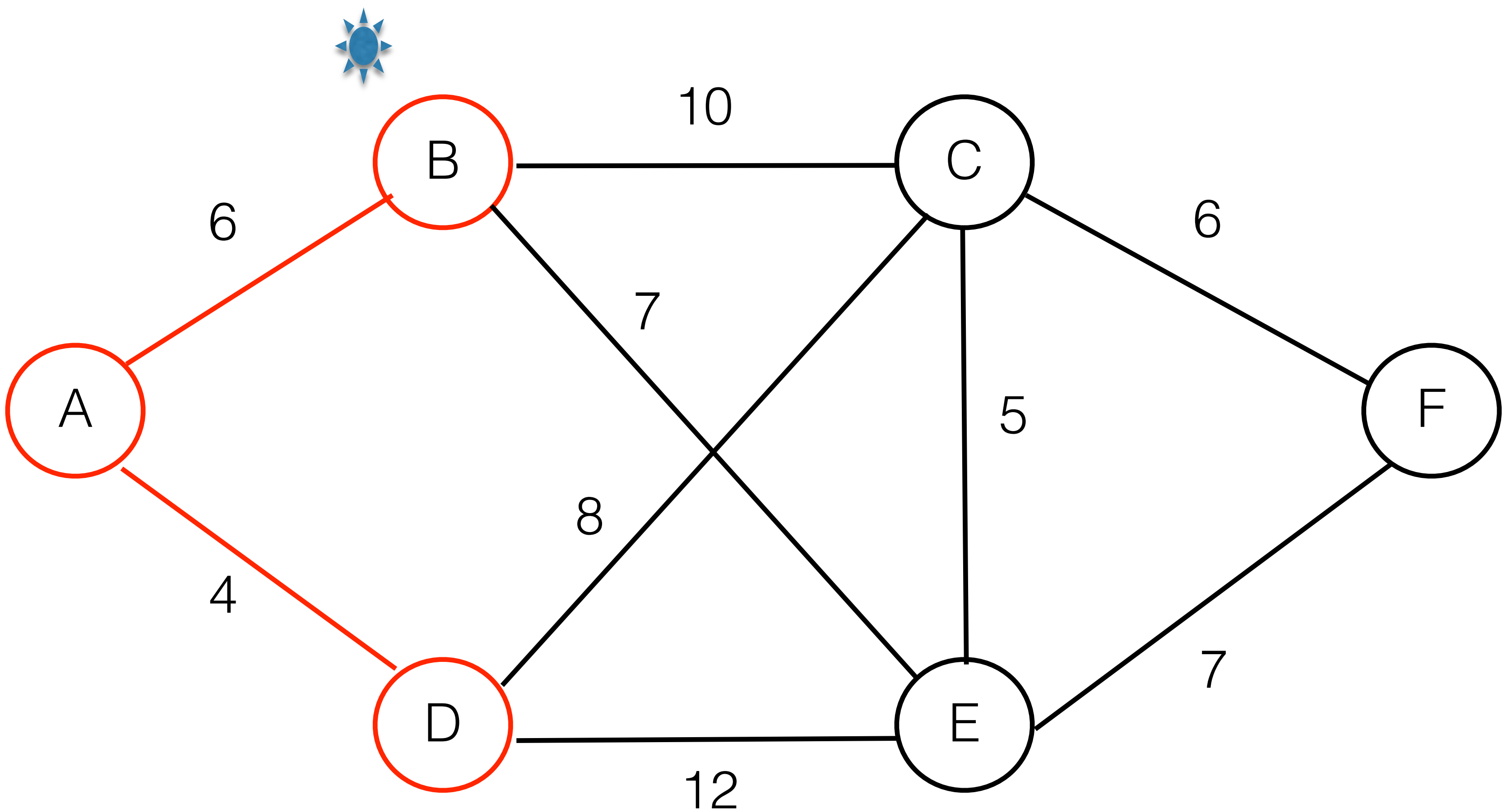
# MST with Weighted Graphs




| 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

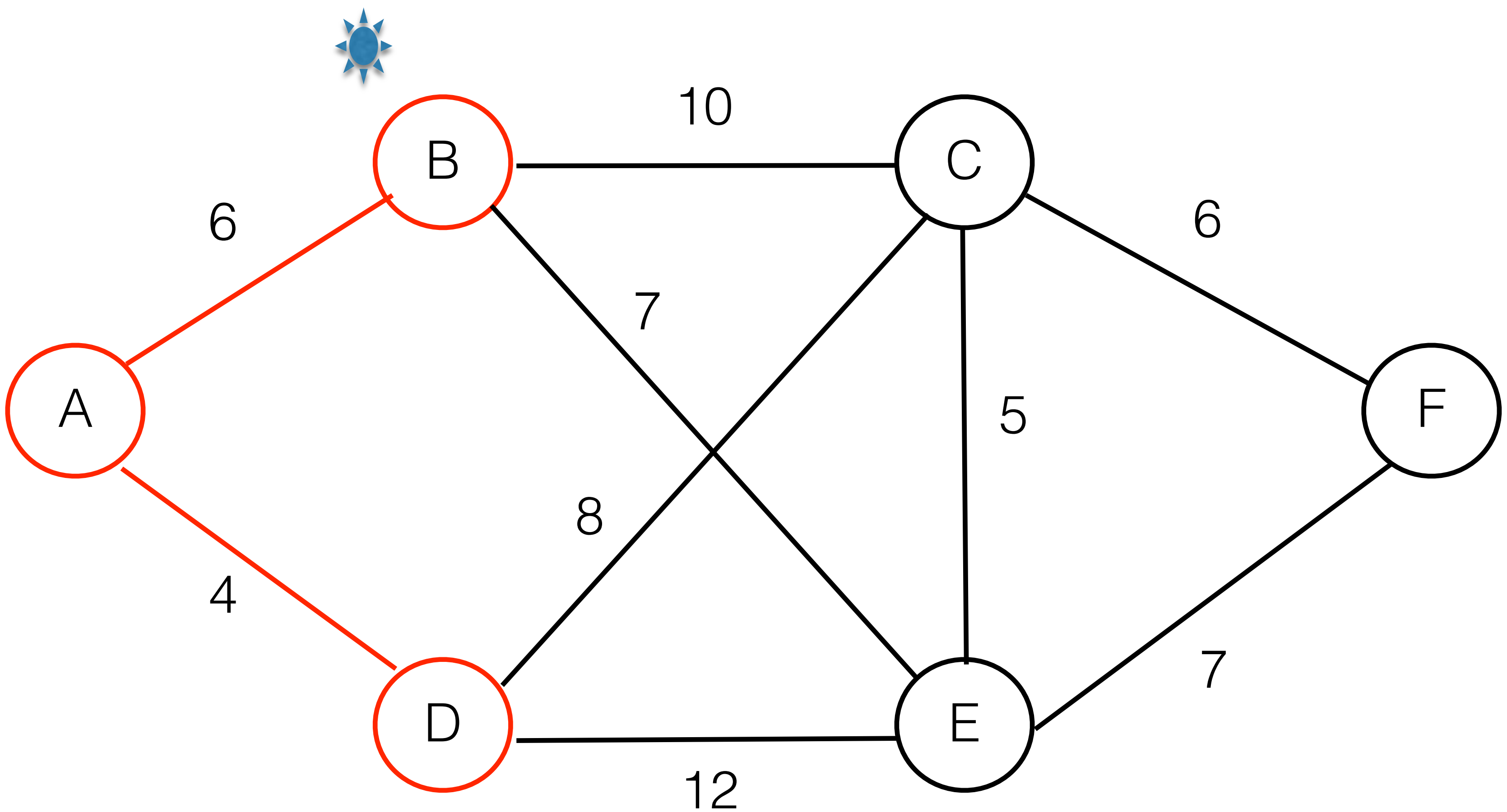
# MST with Weighted Graphs




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

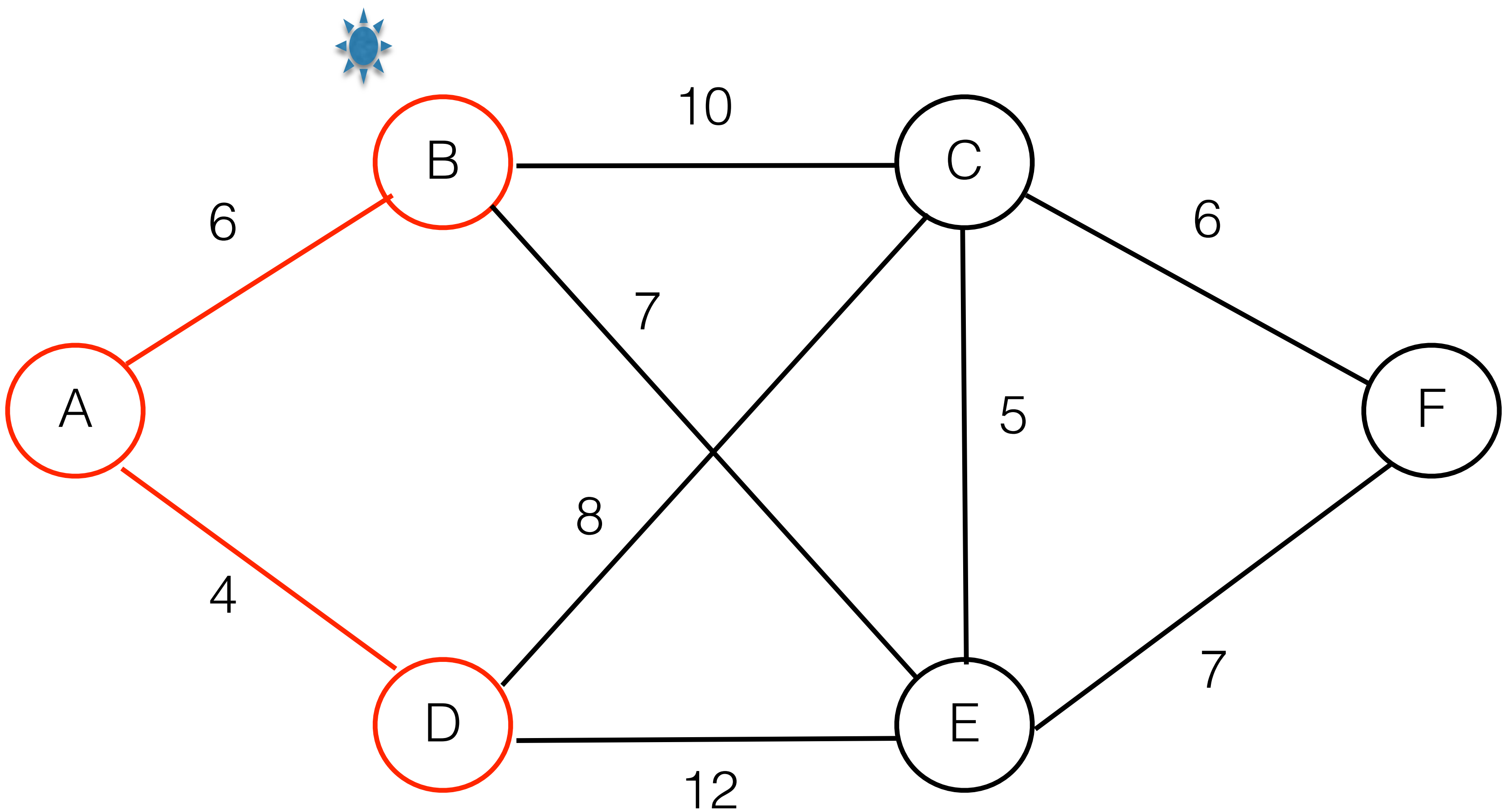
- Now B is the newest office
- Repeat the same for B

# MST with Weighted Graphs



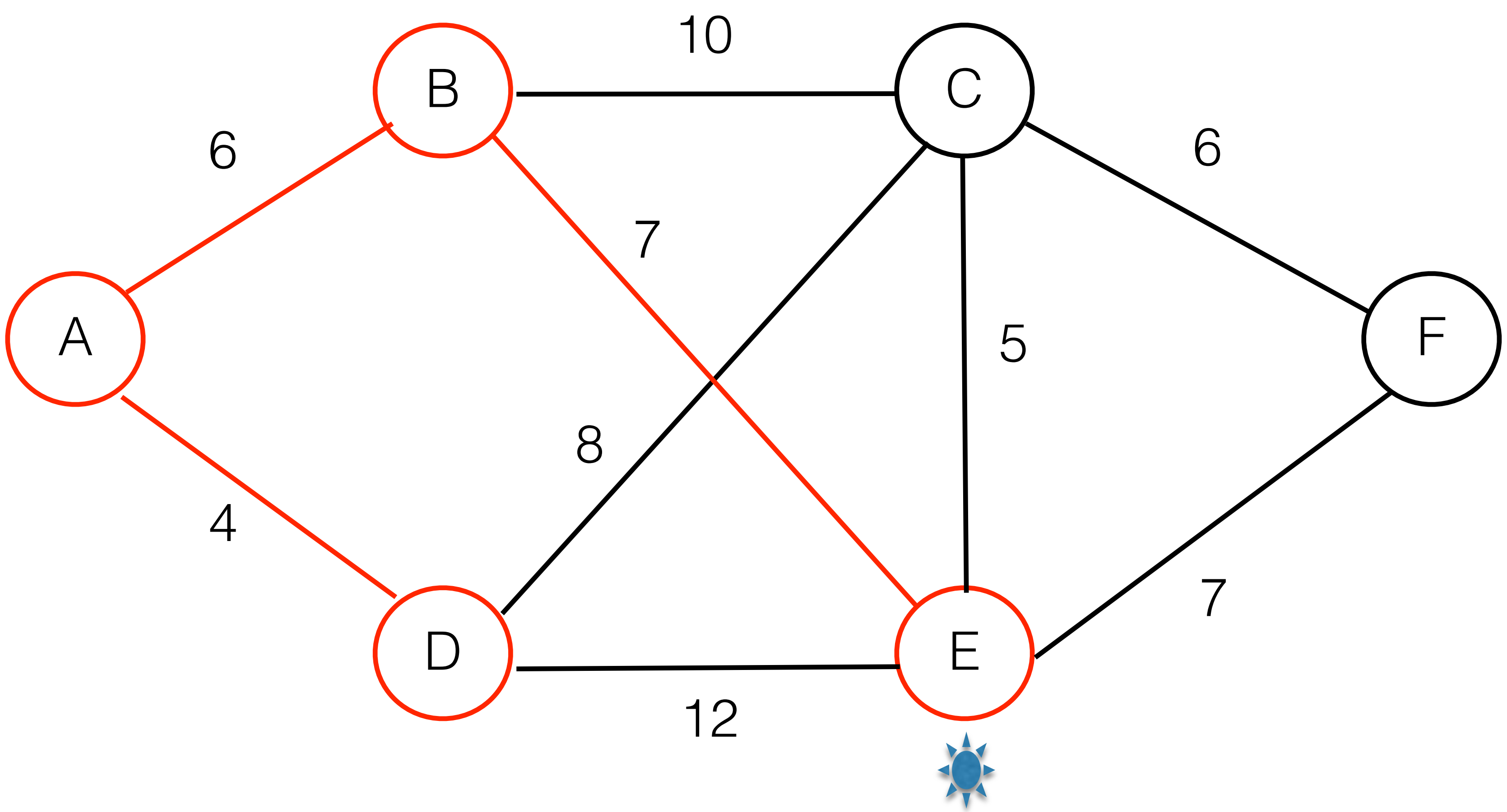

| Edge | Weight |
|---|---|
| DC | 8 |
| DE | 12 |
| BC | 10 |
| BE | 7 |

# MST with Weighted Graphs



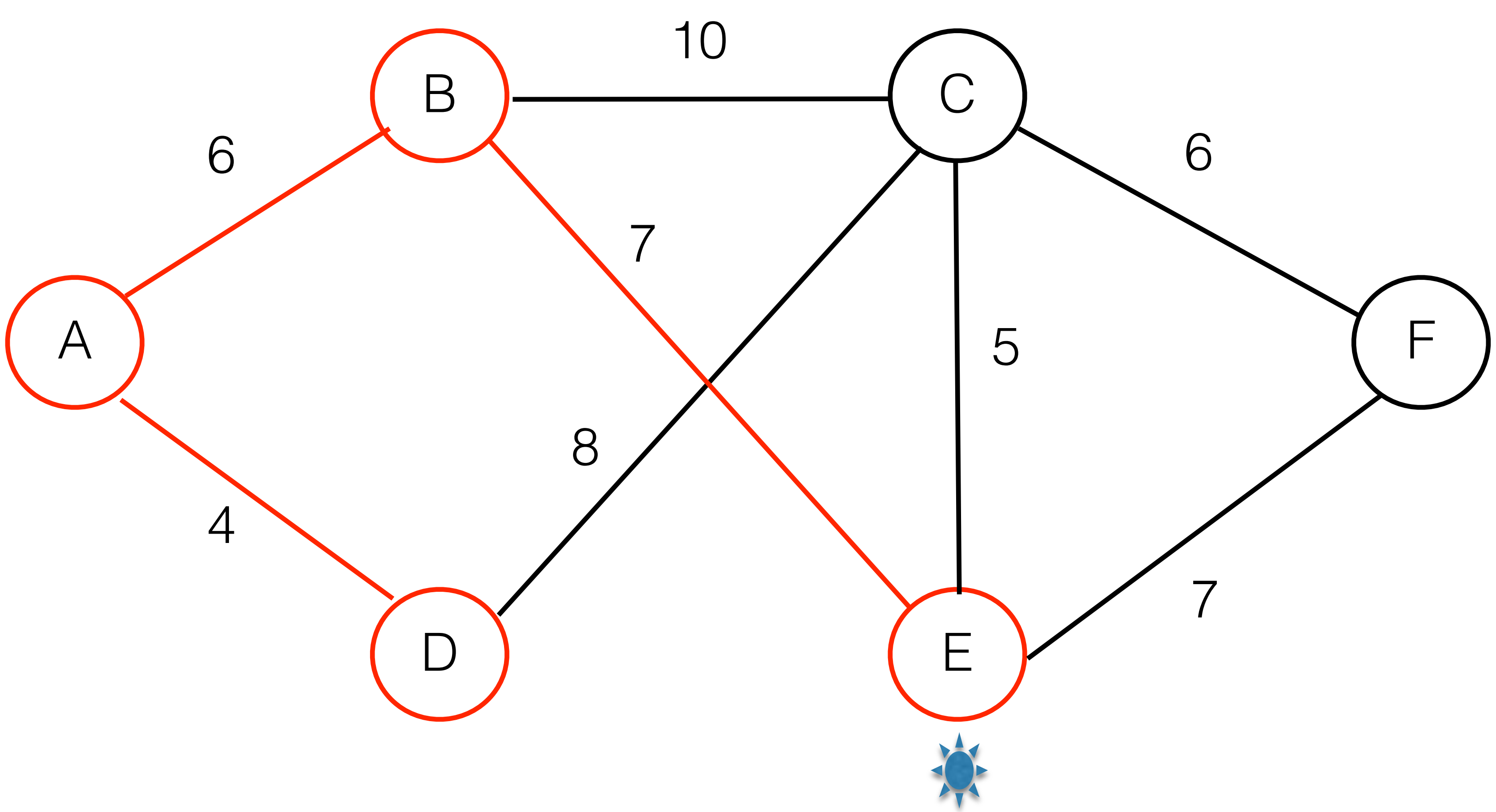

| Edge | Weight |
|---|---|
| DC | 8 |
| DE | 12 |
| BC | 10 |
| **BE** | **7** |

# MST with Weighted Graphs



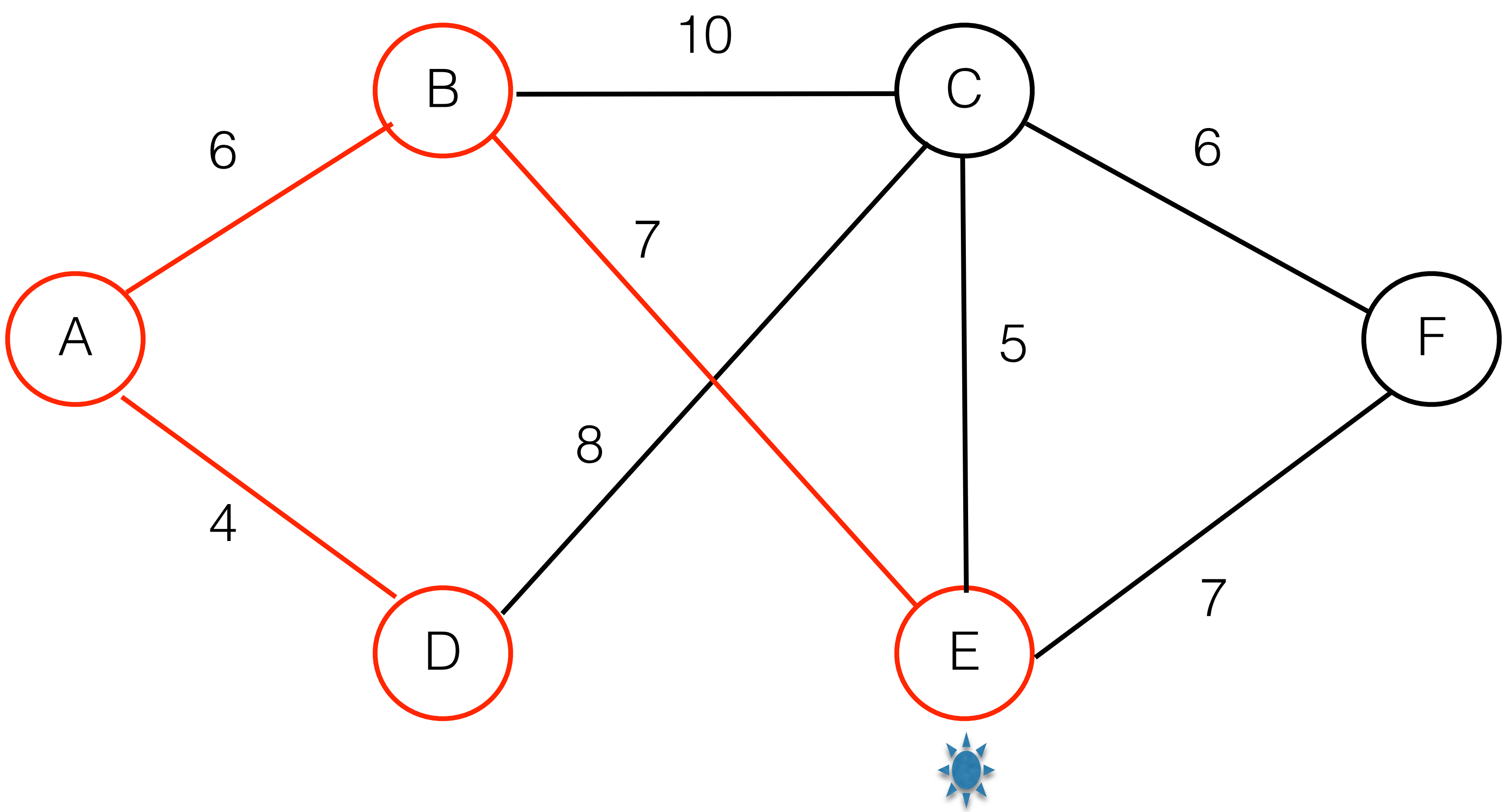

| Edge | Weight |
|---|---|
| DC | 8 |
| **DE** | **12** |
| BC | 10 |
| | |

# MST with Weighted Graphs



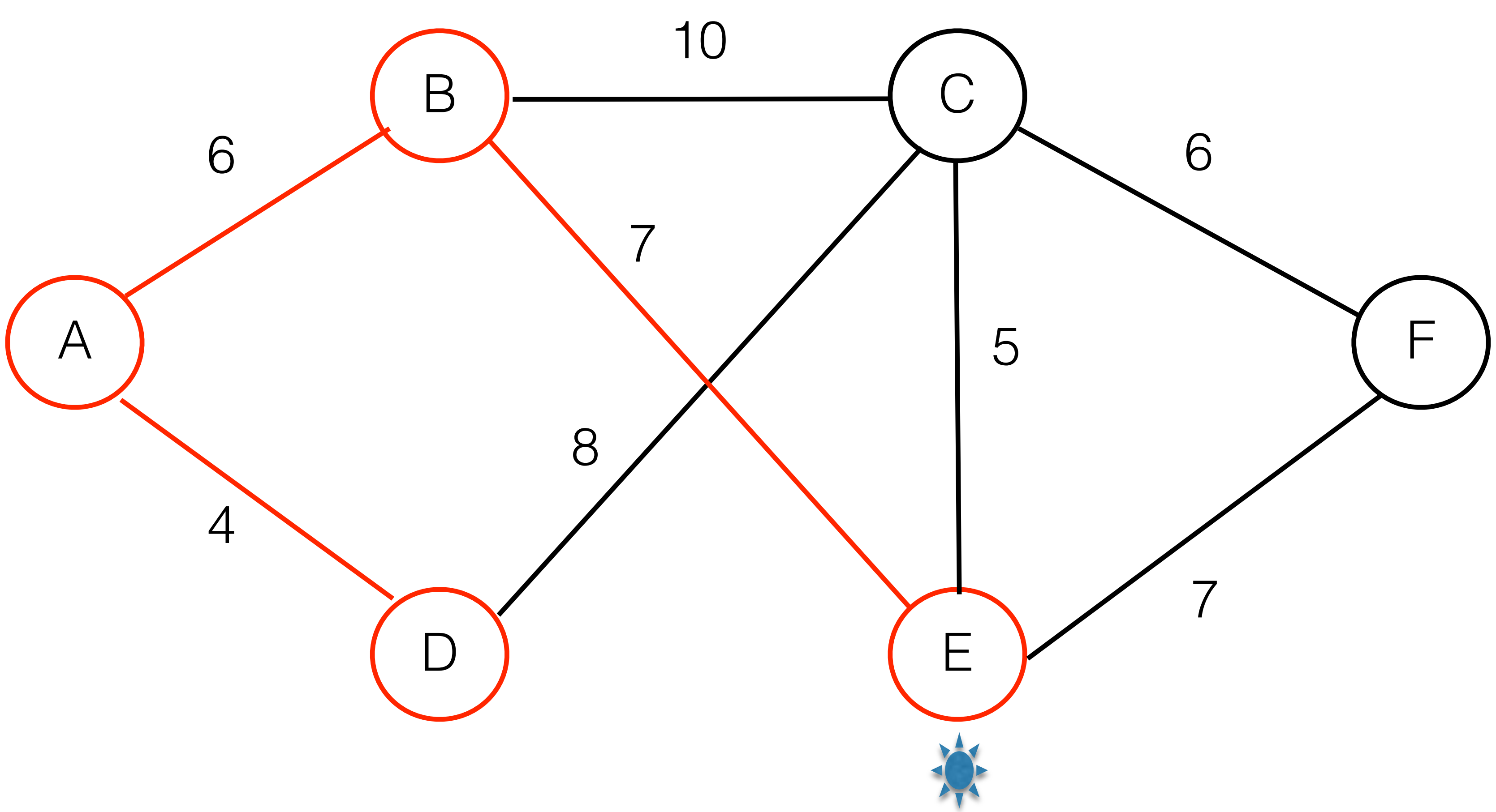

| Edge | Weight |
| --- | --- |
| DC | 8 |
| BC | 10 |
| | |

# MST with Weighted Graphs



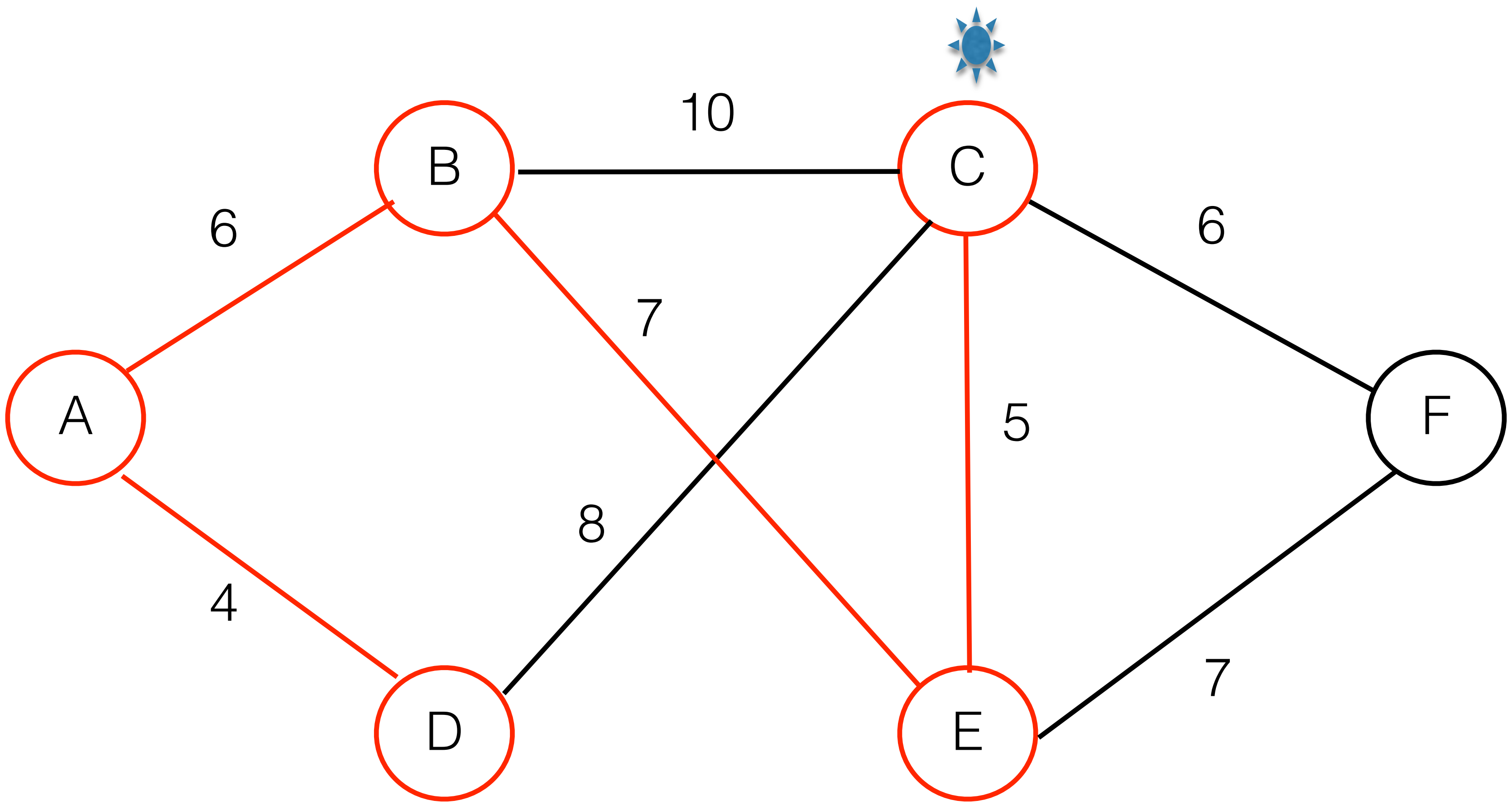

| Edge | Weight |
|---|---|
| DC | 8 |
| BC | 10 |
| EC | 5 |
| EF | 7 |

# MST with Weighted Graphs



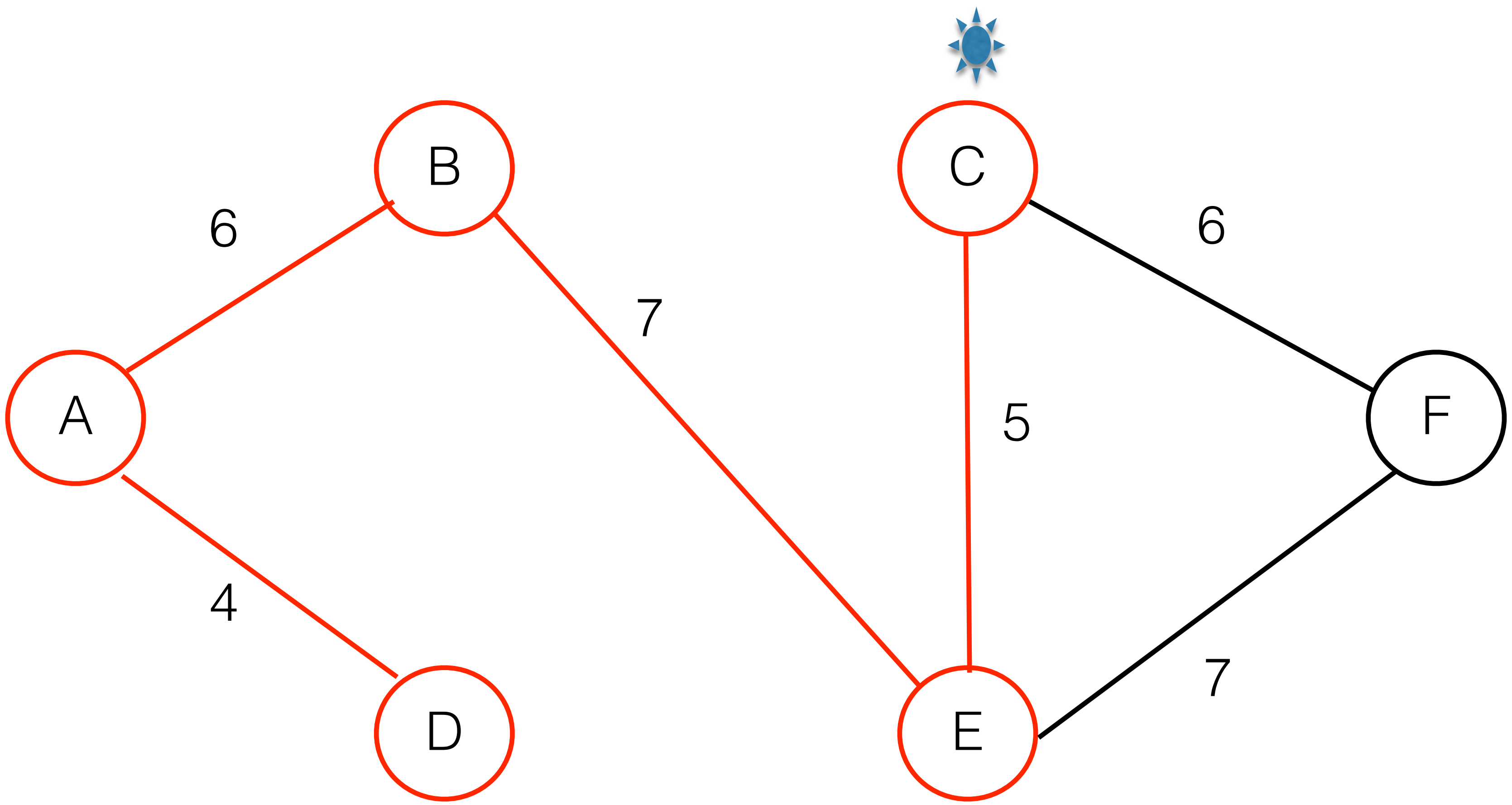

| Edge | Weight |
|---|---|
| DC | 8 |
| BC | 10 |
| **EC** | **5** |
| EF | 7 |

# MST with Weighted Graphs



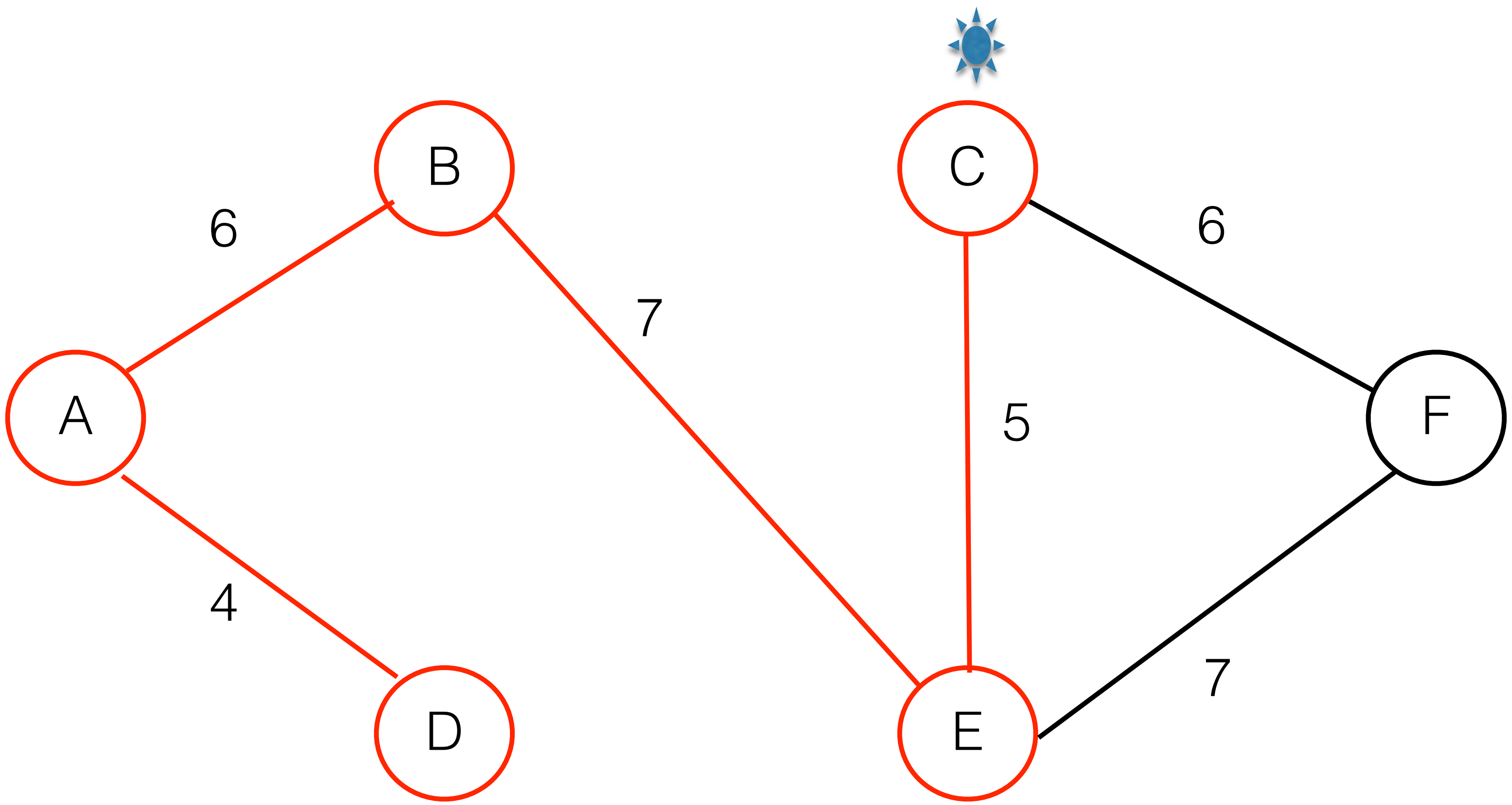

| Edge | Weight |
|---|---|
| **DC** | **8** |
| **BC** | **10** |
| EF | 7 |
| | |

# MST with Weighted Graphs



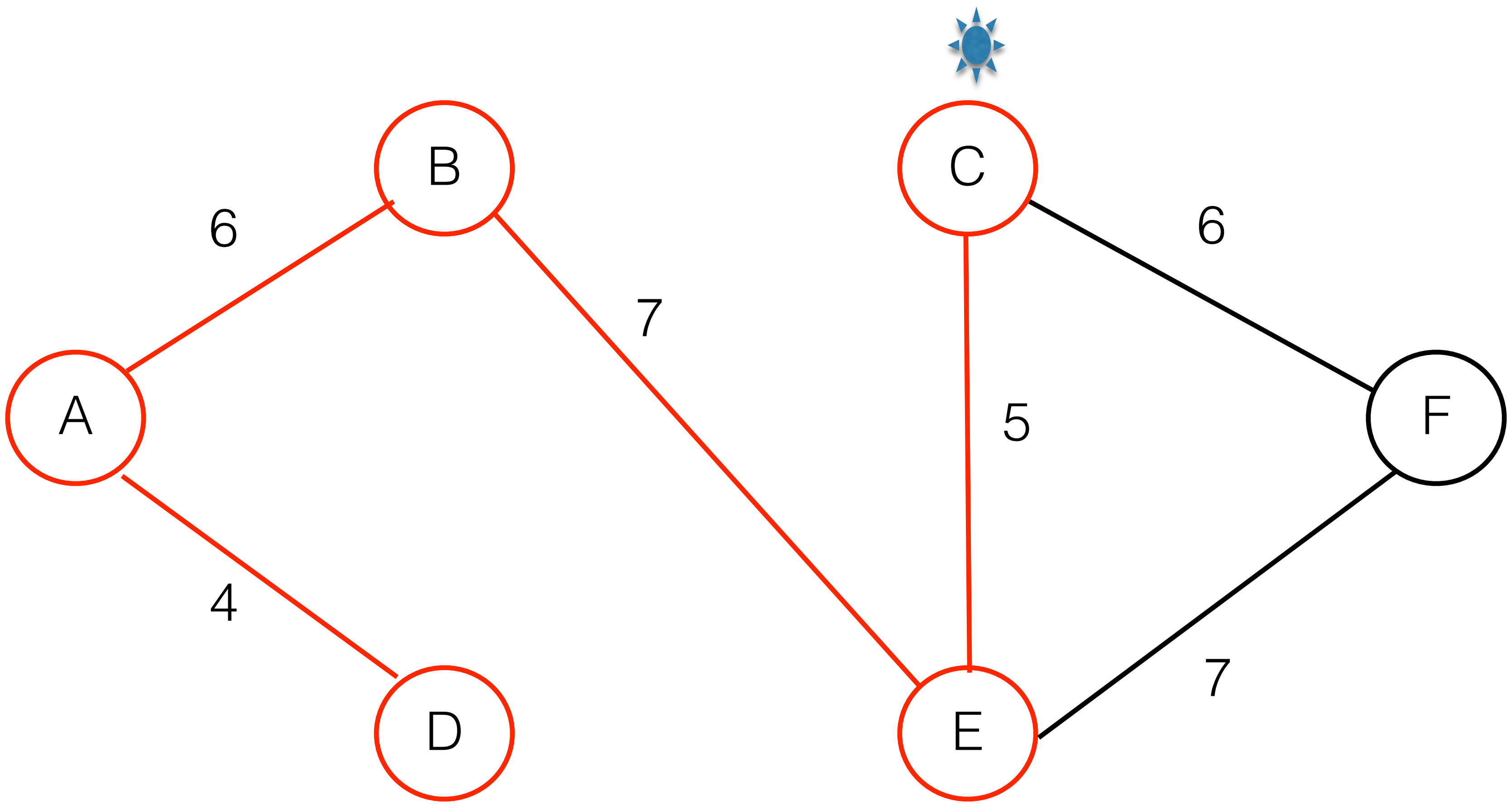

| Edge | Weight |
|---|---|
| EF | 7 |

# MST with Weighted Graphs



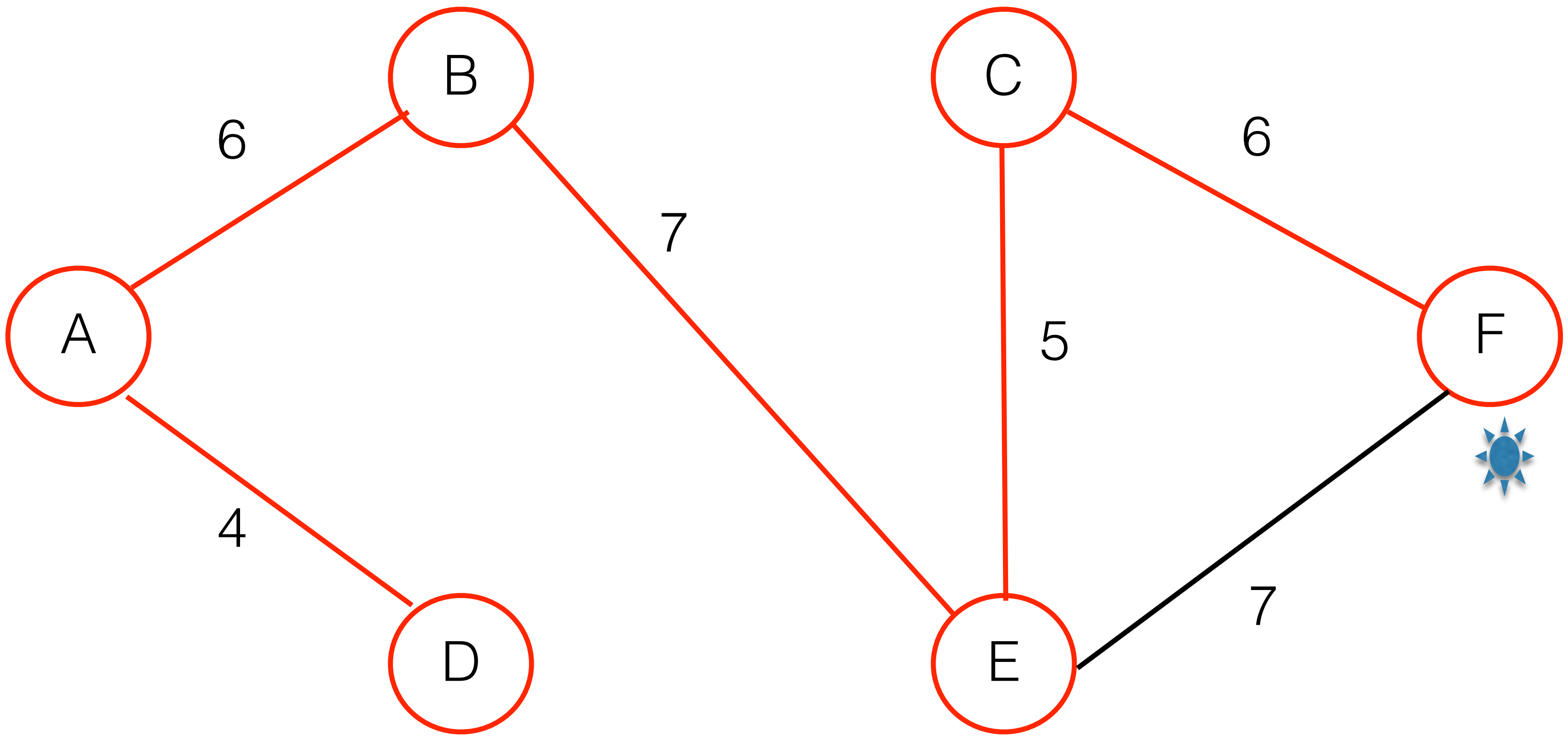

| Edge | Weight |
|---|---|
| EF | 7 |
| CF | 6 |

# MST with Weighted Graphs



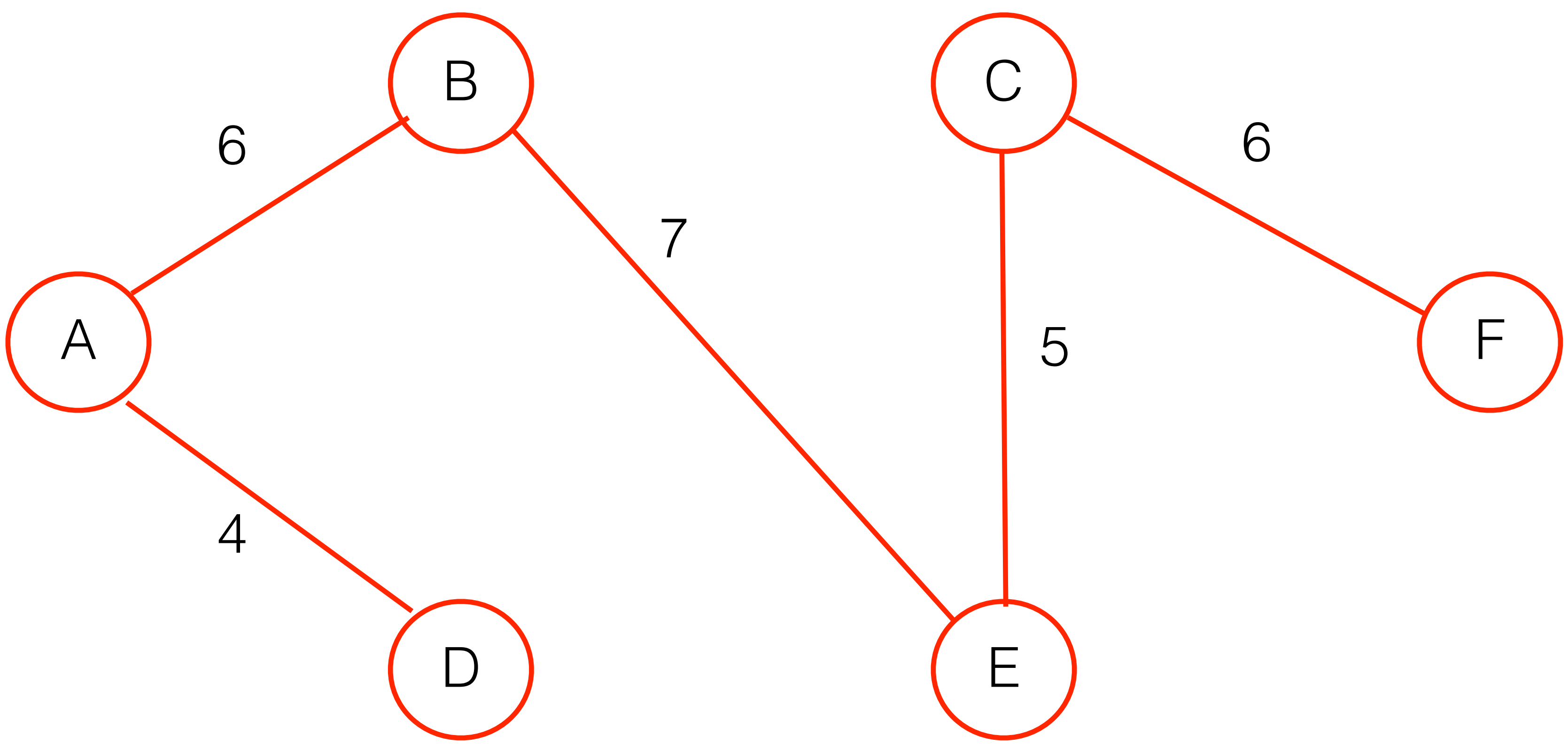

| Edge | Weight |
|---|---|
| EF | 7 |
| **CF** | **6** |

# MST with Weighted Graphs




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

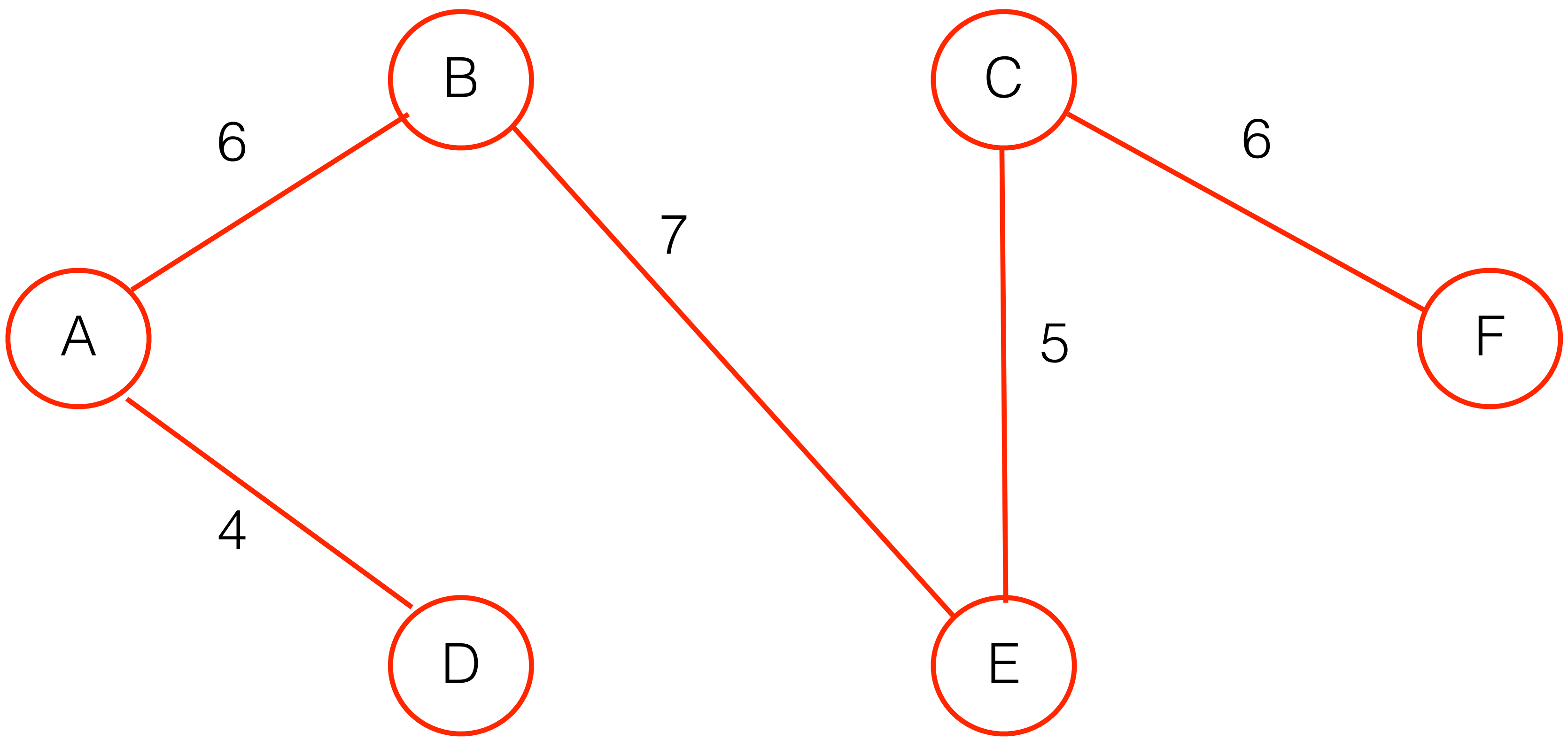
# MST with Weighted Graphs




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

The Resultant Minimum Spanning Tree

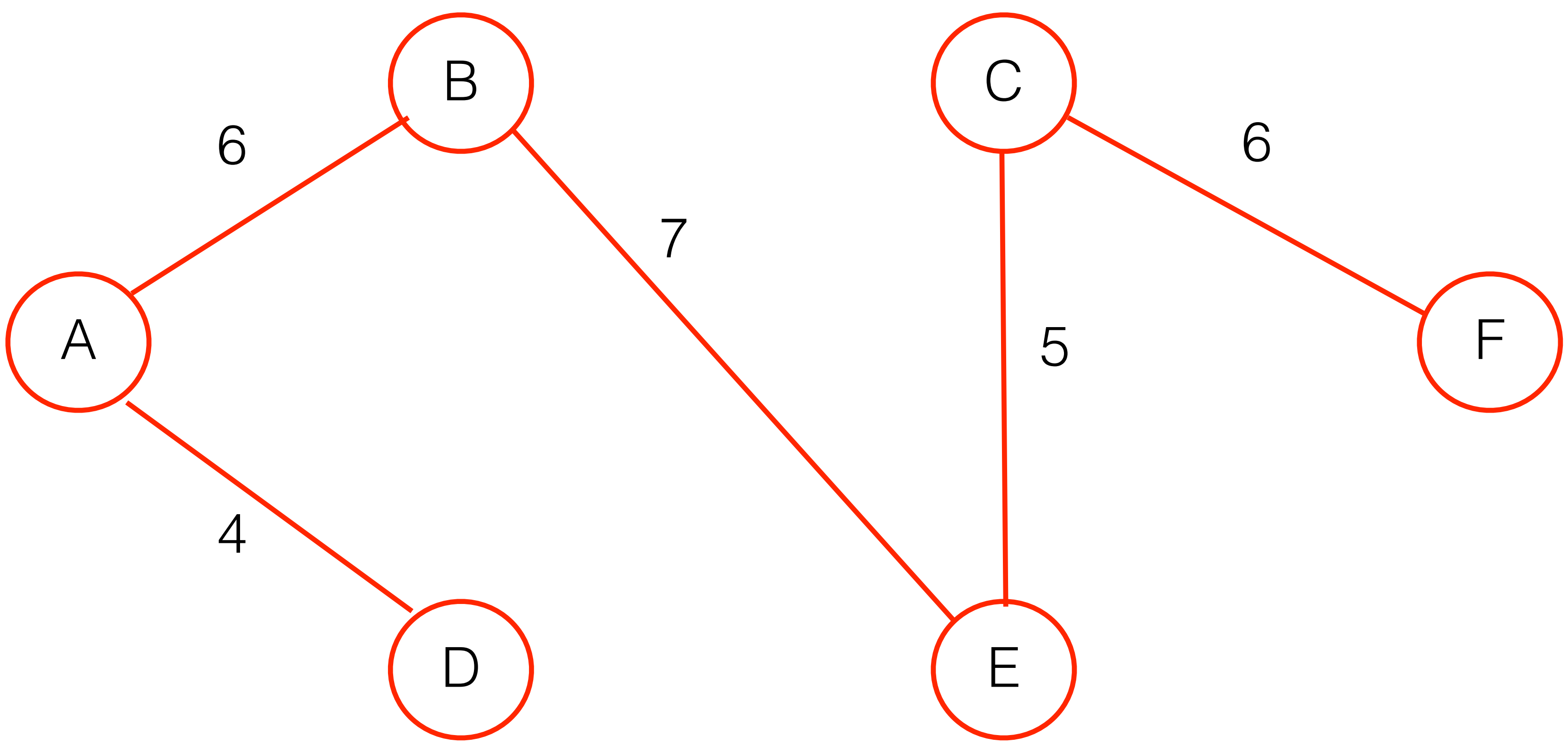
# MST with Weighted Graphs




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

Which data structure would you choose to implement this list?

# MST with Weighted Graphs

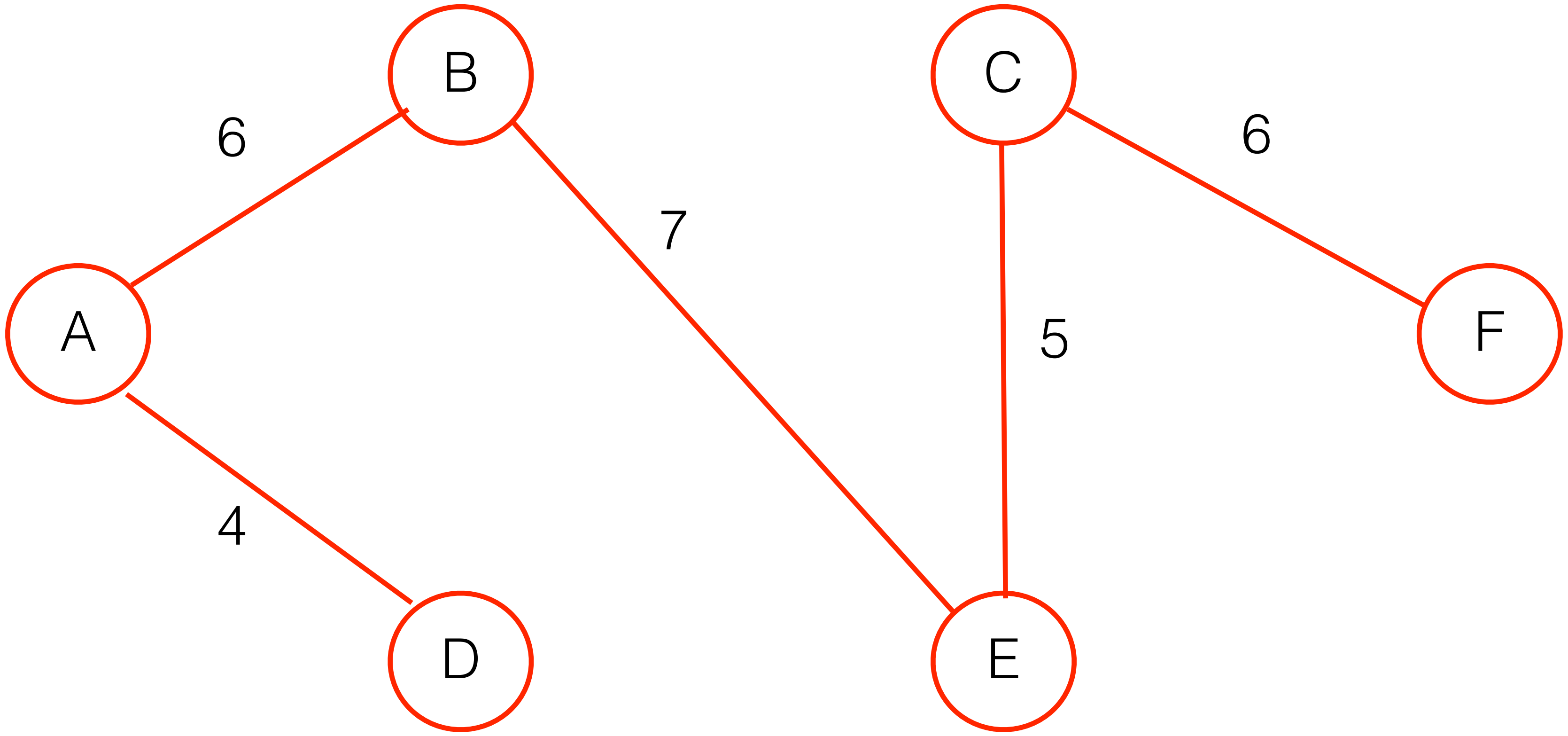



| Edge | Weight |
|---|---|
| | |

Which data structure would you choose to implement this list?

Priority Queue

# Important Points




- We will need a PQ.
- At any point in time, let $\boldsymbol{A}$ be the set of vertices in $\boldsymbol{T}$, and $\boldsymbol{V} - \boldsymbol{A}$ be the set of vertices in PQ.
- Each step adds to $\boldsymbol{T}$ a "light edge" connecting it with a vertex $\boldsymbol{v}$ in PQ (moving $\boldsymbol{v}$ from $\boldsymbol{V} - \boldsymbol{A}$ to $\boldsymbol{A}$).

# Prim-Jarnik's Algorithm

- This idea (which you just learned) of finding the MST is the basis of Prim's algorithm

- But, there is a slight modification 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)}

Key Value

- Where 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)}

Key Value

- Where 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 $(D[v], (v, \mathsf{None}))$ 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 $T$

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 $(D[v], (v, \text{None}))$ 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 using 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 using e

2. And, do the following

“Follow each outgoing edge to every adjacent vertex v of u, replace its key D[v] with the weight w(u,v), only if it is 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: $O(|E| \log(|V|))$

# Summary

1. Optimization Problems

2. Greedy Algorithms

3. Minimum Spanning Tree

- MST in Unweighted Graphs

- MST in Weighted Graphs (Prim's Algorithm, and Krushkal's Algorithm)