

CSCI 256 Lecture 9:

Shortest Paths &
Min Spanning Trees

Shikha Singh

Announcements

- Hand in **Exercise #7**, pick up **Exercise #8**
- **Assignment 2** due tonight at 10 pm
- First in-class test is on **Mon, Oct 5**
 - Will cover topics up until Assignment 2 and Exercise 5
 - Chapter 1.1, 2.1-2.3 and 3 in KT
 - Stable matching, Asymptotics, Graphs, BFS, DFS and their applications, directed graphs and topsort
 - All content up to but not including Dijkstra's
 - Practice test posted on GLOW
- Office hours: 1.30-3 pm today!

Last Time

- Moved on from simple traversal algorithms to traversal algorithms that **optimize an objective** (such as min distance)
- Covered Dijkstra's algorithm in detail
- Proved that it is correct: always computes shortest path
- Today: running time analysis
 - How to use data structures to improve time bound
- Simpler algorithm for DAGs

Dijkstra's Algorithm

```
procedure Dijkstra(G, s)      # G = (V,E) is connected
    S = {s}, d[s] ← 0        # initialize
    for all neighbors v of s:
        d[v] ← w(s, v)
    for all non-neighbors v of s:
        d[v] ← ∞
    while S not equal to V:
        select v from V - S with min d[v]
        Add v to S
        for each neighbor u of v in V - S:
            if d[v] + w(v, u) < d[u]: # better path
                d[u] = d[v] + w(v, u) # update
```

n iterations at most

Can do in $O(n)$ scan

$O(\text{outdegree}(v))$ time

Dijkstra's Algorithm: Runtime

```
procedure Dijkstra( $G, s$ )      #  $G = (V, E)$  is connected
     $S = \{s\}, d[s] \leftarrow 0$   # initialize
    for all neighbors  $v$  of  $s$ :
         $d[v] \leftarrow w(s, v)$ 
    for all non-neighbors  $v$  of  $s$ :
         $d[v] \leftarrow \infty$ 
    while  $S$  not equal to  $V$ :
        select  $v$  from  $V - S$  with min  $d[v]$ 
        Add  $v$  to  $S$ 
        for each neighbor  $u$  of  $v$  in  $V - S$ :
            if  $d[v] + w(v, u) < d[u]$ : # better path
                 $d[u] = d[v] + w(v, u)$  # update
```

Can implement in $O(n(n + m))$ time

Dijkstra's Algorithm: Runtime

```
procedure Dijkstra( $G, s$ )      #  $G = (V, E)$  is connected
     $S = \{s\}, d[s] \leftarrow 0$     # initialize
    for all neighbors  $v$  of  $s$ :
         $d[v] \leftarrow w(s, v)$ 
    for all non-neighbors  $v$  of  $s$ :
         $d[v] \leftarrow \infty$ 
    while  $S$  not equal to  $V$ :
        select  $v$  from  $V - S$  with min  $d[v]$ 
        Add  $v$  to  $S$ 
        for each neighbor  $u$  of  $v$  in  $V - S$ :
            if  $d[v] + w(v, u) < d[u]$ : # better path
                 $d[u] = d[v] + w(v, u)$  # update
```

Every time a node is added to S , update neighbors of shorter path

Choose min d node

Dijkstra's Algorithm: Runtime

```
procedure Dijkstra(G, s)      # G = (V,E) is connected
    S = {s}, d[s] ← 0        # initialize
    for all neighbors v of s:
        d[v] ← w(s, v)
    for all non-neighbors v of s:
        d[v] ← ∞
    while S not equal to V:
        select v from V - S with min d[v]
        Add v to S
        for each neighbor u of v in V - S:
            if d[v] + w(v, u) < d[u]: # better path
                d[u] = d[v] + w(v, u) # update
```

Update key values

ExtractMin key

Priority Queue: Using Heaps (CS 136 Review)

Priority Queues

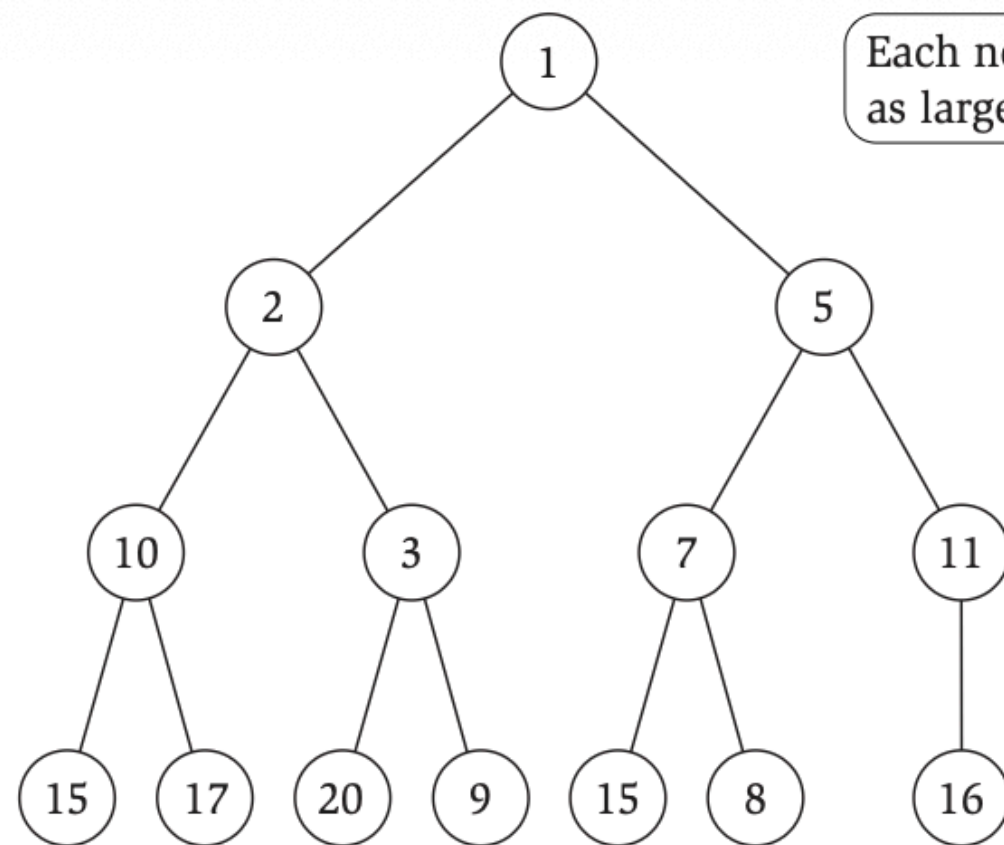
- Maintain a dynamic set S of items, each with a key (**priority** value), which typically supports
 - **Insert (x, k)**. Insert a new element x into S with key k
 - **ExtractMin**. Retrieve element in S with min key
 - **[Optional] ChangeKey**. update key of an item in S
- Implemented using a **binary heap** data structure
- Before we discuss heaps, let's discuss why existing data structures may not work as well
 - Linked list? Sorted array?

Heaps

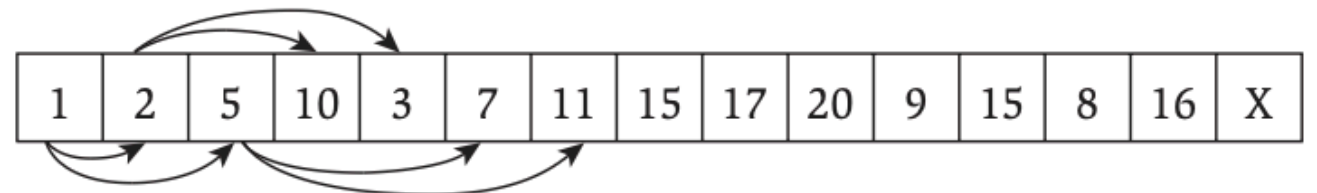
- Combines benefits of a tree structure but stored as an array
- Let's pretend we have a binary tree first
- Heap invariants:
 - Each element in the heap is smaller than its children
 - It is a **complete** binary tree: all levels but the last are "full"

Heap Example

Heap property: For every element v , at node i , the element w at i 's parent satisfies $\text{key}(w) \leq \text{key}(v)$



Each node's key is at least as large as its parent's.

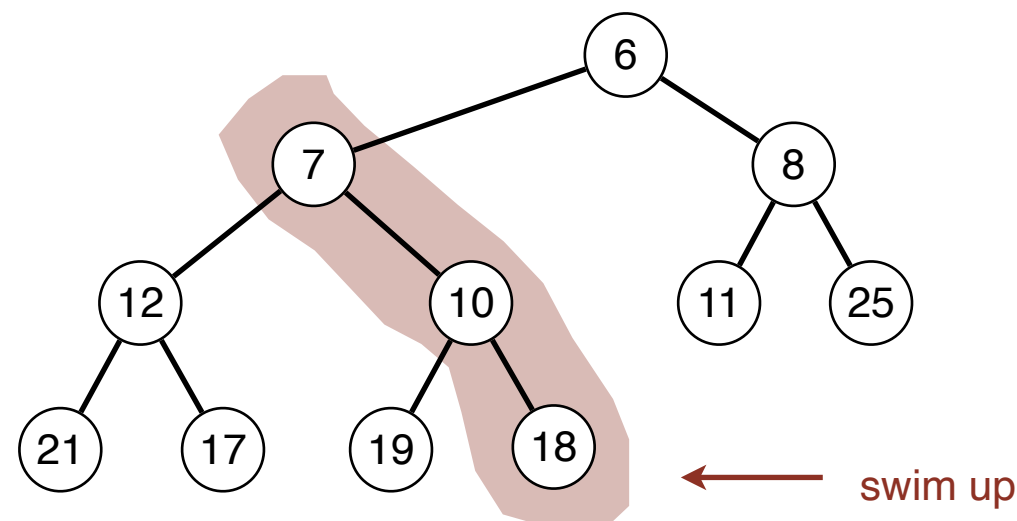
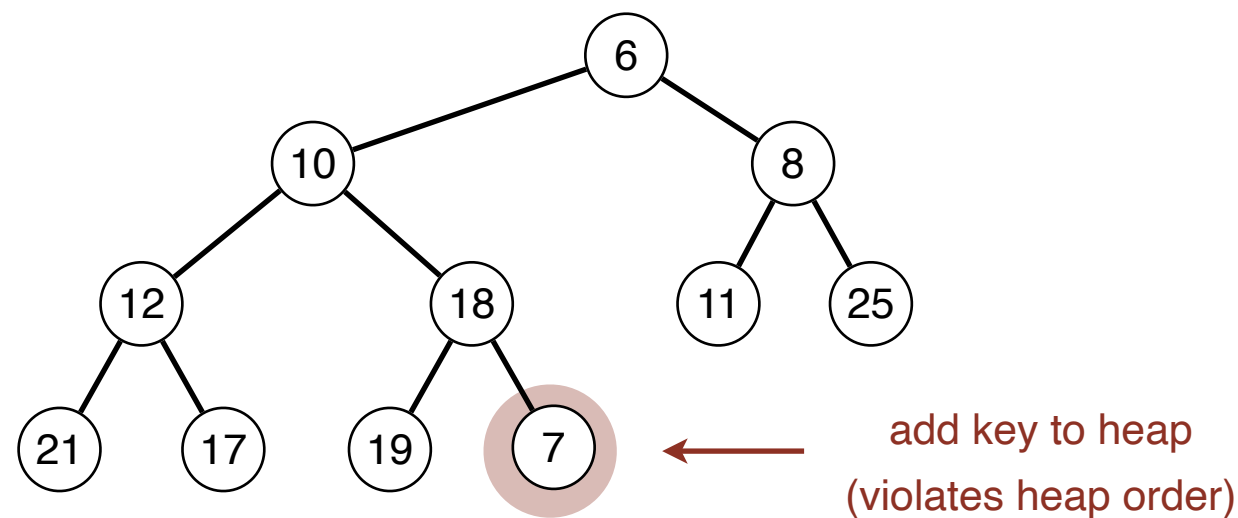


$\text{parent}(i) = \lfloor i/2 \rfloor$.
 $\text{Children}(i) = 2i, 2i + 1$

Mapping to an array

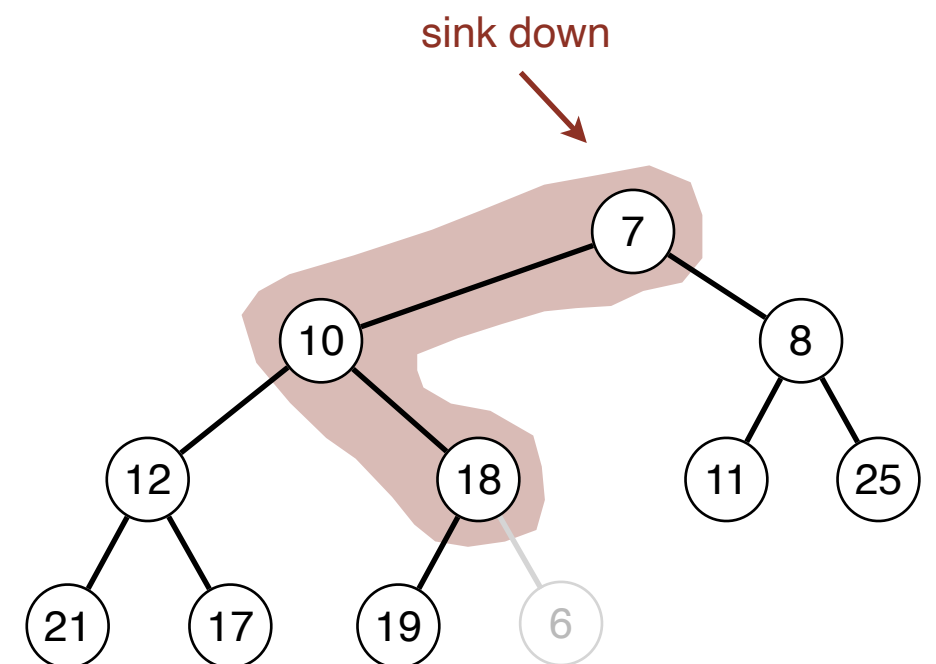
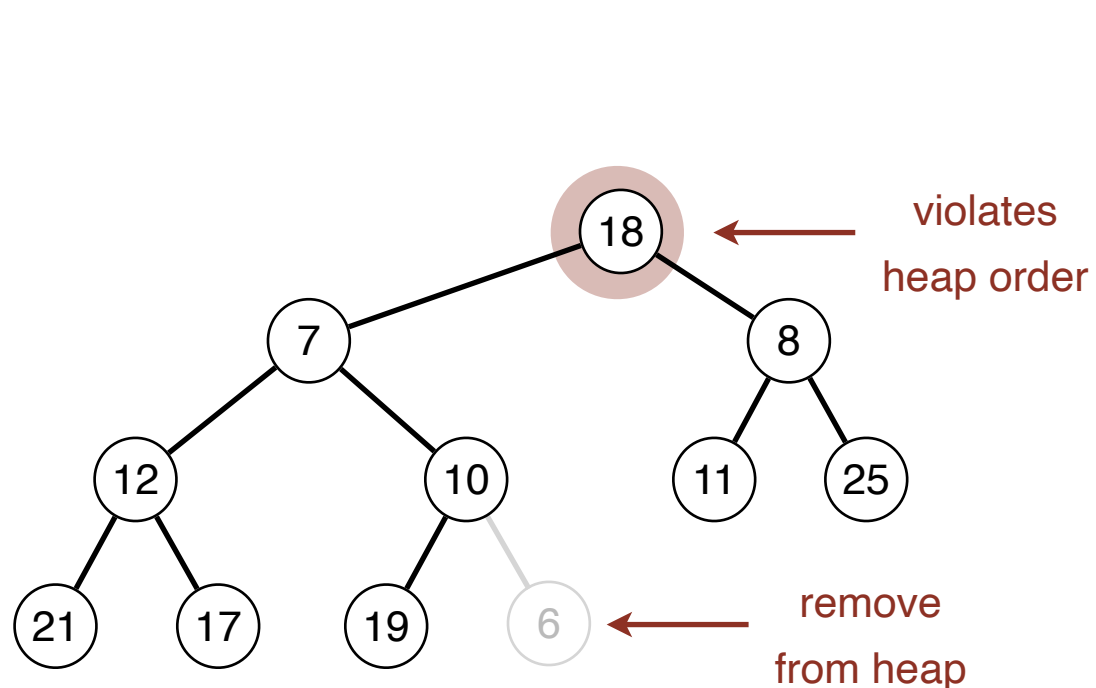
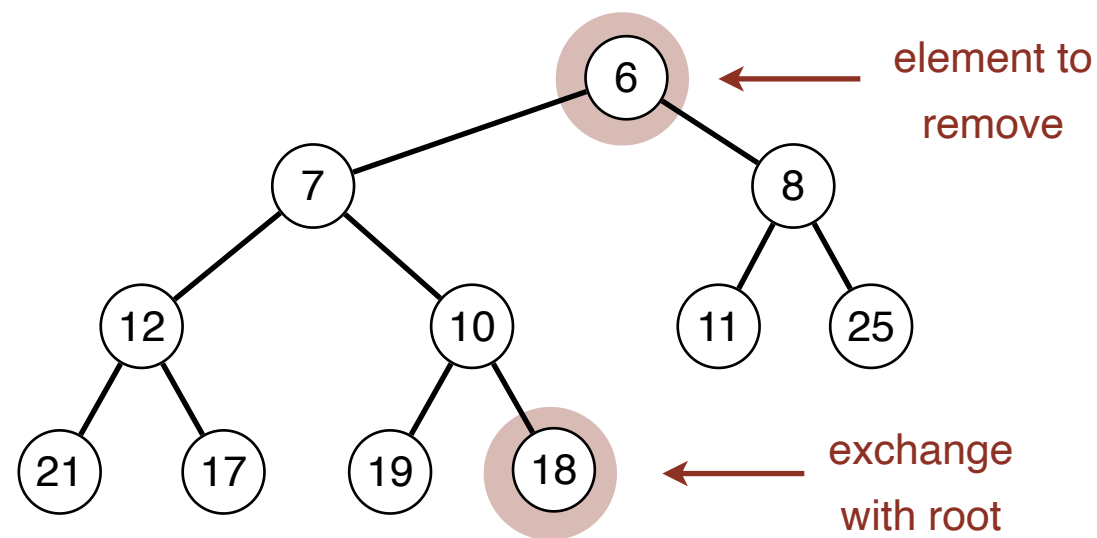
Binary Heap: Insert

Insert. Add element in new node at end; repeatedly exchange new element with element in its parent until heap order is restored.



Binary Heap: Extract Min

Extract min. Exchange element in root with last node; repeatedly exchange element with its smaller child until heap order is restored.



Decrease/Change Key

- Book talks about implementing this for Dijkstra
- Most standard priority queue implementations do not support it
- Any ideas how to avoid it for Dijkstra?
 - Instead of decreasing the distance explicitly, we can just insert a duplicate node with new (shorter) distance
 - Does not effect correctness which depends on min value
- Downside?
 - Max size of heap can grow, but by how much?
 - $O(m)$ instead of $O(n)$

Priority Queue Summary

Given a priority with n items in it, we can perform the following operations efficiently:

- Insert: $O(\log n)$
- ExtractMin: $O(\log n)$
 - $O(1)$ to just find minimum
 - $O(\log n)$ to delete and maintain heap property

Dijkstra's Algorithm: Runtime

```
procedure Dijkstra( $G, s$ )      #  $G = (V, E)$  is connected
```

```
   $S = \{s\}, d[s] \leftarrow 0$       # initialize
```

```
  for all neighbors  $v$  of  $s$ :
```

```
     $d[v] \leftarrow w(s, v)$ 
```

```
  for all non-neighbors  $v$  of  $s$ :
```

```
     $d[v] \leftarrow \infty$ 
```

```
  while  $S$  not equal to  $V$ :
```

```
    select  $v$  from  $V - S$  with min  $d[v]$ 
```

```
    Add  $v$  to  $S$ 
```

```
    for each neighbor  $u$  of  $v$  in  $V - S$ :
```

```
      if  $d[v] + w(v, u) < d[u]$ : # better path
```

```
         $d[u] = d[v] + w(v, u)$  # update
```

Extract Node
with Min $d[u]$

Insert
duplicate
 $d[u]$

Performance Analysis

Running time breakdown.

- n deleteMin operations from PQ to select next vertex $O(n \log n)$
- At most m inserts to PQ: $O(m \log n)$
- Traversal of the graph: $O(n + m)$

Overall running time:

- $O((n + m) \log n)$. This is $O(m \log n)$ if G is connected
- Space: $O(n + m)$
- **Question.** Is this optimal? Or is it possible to do better?
- Was an open problem since 1956 (original algorithm)

Breaking the Sorting Barrier for Directed Single-Source Shortest Paths

Ran Duan

Tsinghua University
Beijing, China
duanran@mail.tsinghua.edu.cn

Jiayi Mao

Tsinghua University
Beijing, China
mgy22@mails.tsinghua.edu.cn

Xiao Mao

Stanford University
Stanford, USA
matthew99a@gmail.com

Xinkai Shu

Max Planck Institute for Informatics
Saarbrücken, Germany
xshu@mpi-inf.mpg.de

Longhui Yin

Tsinghua University
Beijing, China
ylh21@mails.tsinghua.edu.cn

Abstract

We give a deterministic $O(m \log^{2/3} n)$ -time algorithm for single-source shortest paths (SSSP) on directed graphs with real non-negative edge weights in the comparison-addition model. This is the first result to break the $O(m + n \log n)$ time bound of Dijkstra's algorithm on sparse graphs, showing that Dijkstra's algorithm is not optimal for SSSP.

Breakthrough last year: STOC 2025

Shortest Path on a DAG

- Simpler algorithm that runs in $O(n + m)$ time
- Why do DAGs provide a nice structure?
- Traverse vertices in topological order (starting from source s)
 - Each vertex u propagates shortest path from s to neighbors that goes through u
 - When you arrive a new node v , all paths leading to it from s have already been considered, why is that?

for each u (in top order):

 for each neighbor v :

 if $d[v] > d[u] + w(u, v)$:

 set $d[v] = d[u] + w(u, v)$

Connection to A^* Search

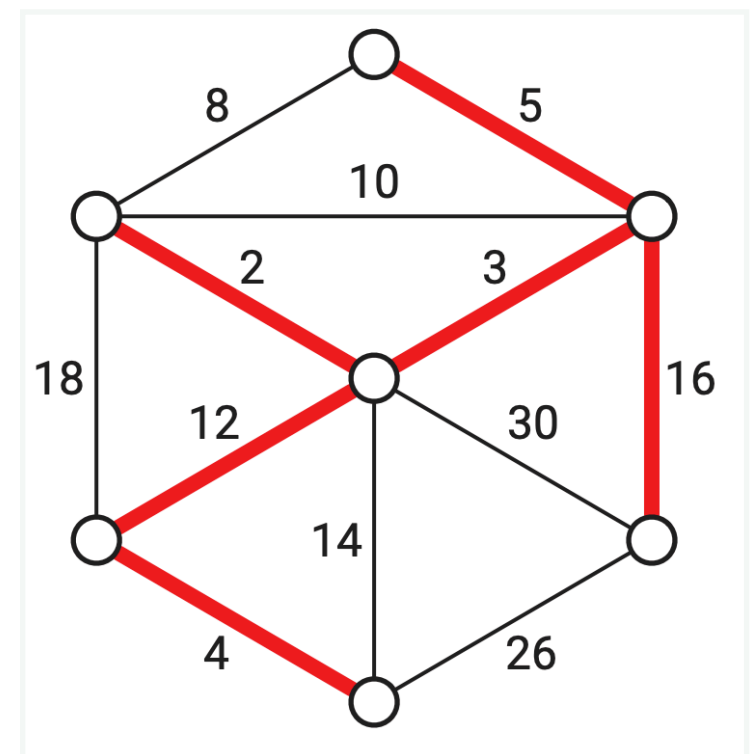
- Essentially Dijkstra's algorithm with a heuristic that guides the search based on estimated distance to a particular target t
- Let $d[v]$ is the estimate of the shortest path distance discovered so far from s and let $h[v]$ be an estimate of the remaining distance to the target t
- Dijkstra's algorithm processes vertices based on min d value
- A^* search is the same algorithm but processes vertices based on min sum of h and d values
 - Goal to get to target faster heuristically
 - Correct if estimate h does not overestimate distance

Minimum Spanning Trees

Minimum Spanning Trees

Problem. Given a connected, undirected graph $G = (V, E)$ with edge weights w_e , output a minimum spanning tree T , i.e., set of edges $T \subseteq E$ s.t.

- T is a spanning tree of G : tree T connects all vertices
- T is a spanning tree of minimum weight: for any other spanning tree $\hat{T}$ of G , we have $\sum_{e \in T} w_e \leq \sum_{e \in \hat{T}} w_e$
- **Question.** How would we compute a min spanning tree in a graph with all edge weights equal to 1?

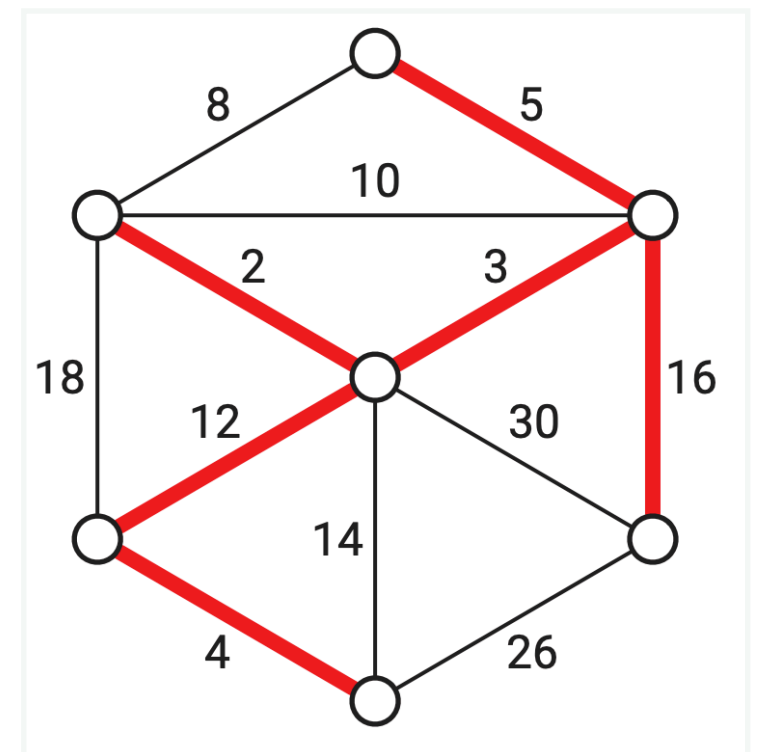


MST: Applications

- Many applications!
 - Classic application:
 - Connectivity in telecom or road network
 - Efficient broadcasting on a computer network
 - Approximate solutions to harder problems, such as TSP
 - Real-time face verification
 - Clustering and data analysis

Ideas of Algorithms?

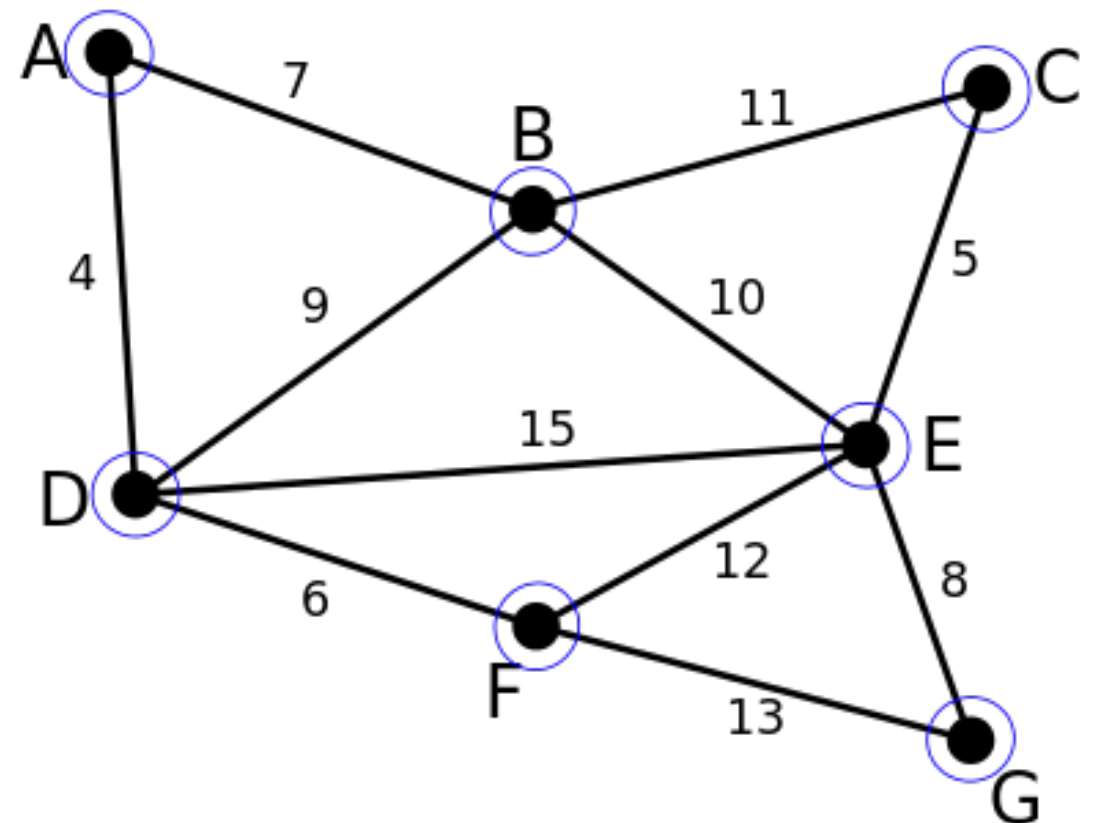
- How to construct a spanning tree?
- Constraints:
 - Need to ensure graph is not disconnected
 - Need to do so with lowest cost possible

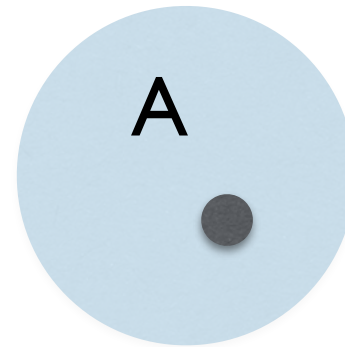
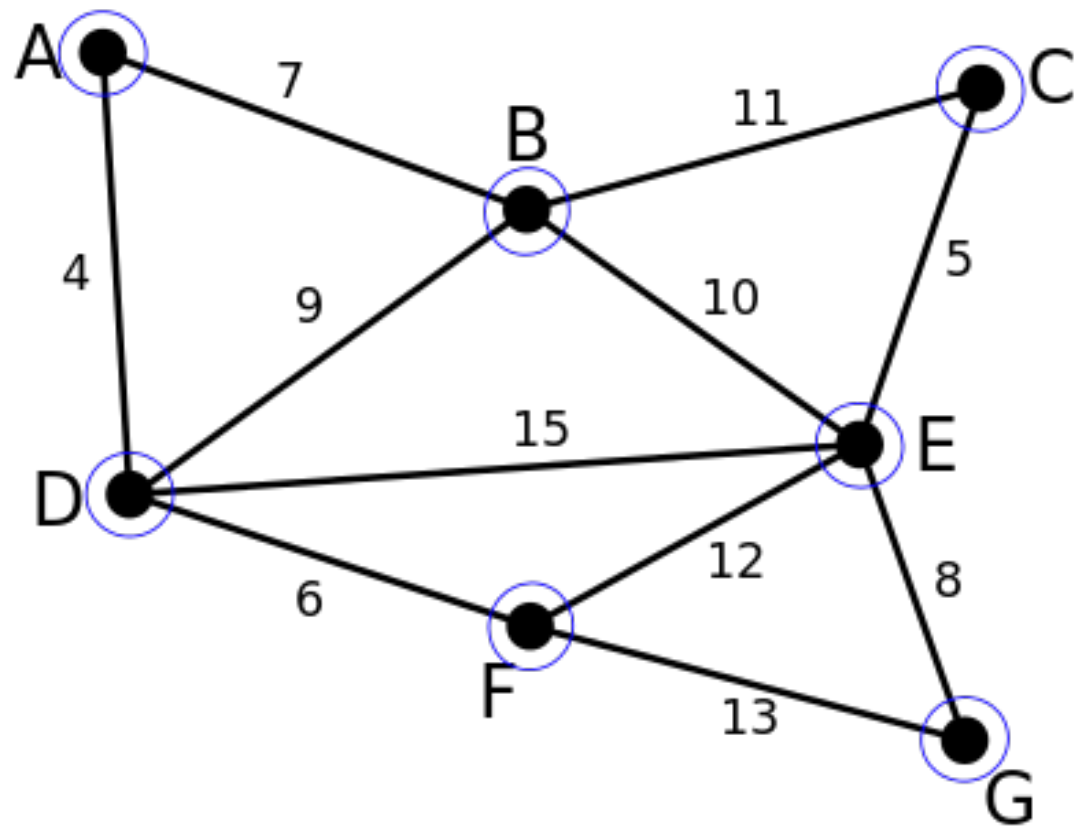


“Prims” Algorithm

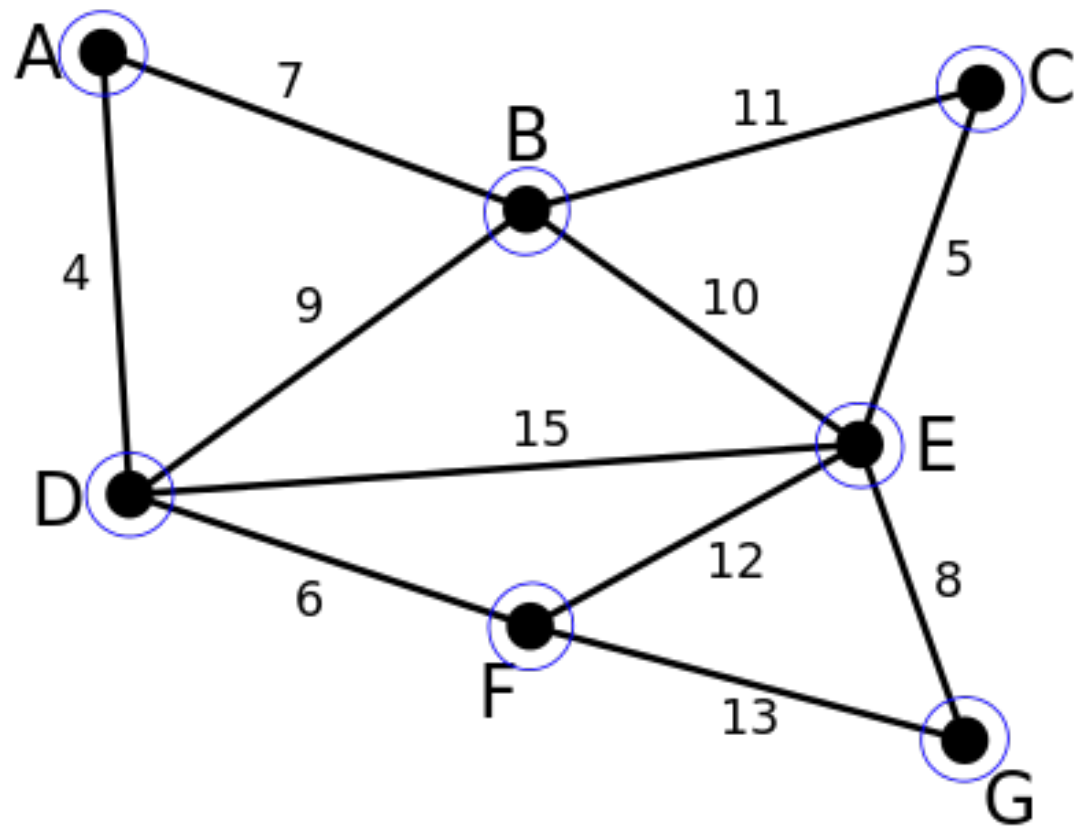
Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$

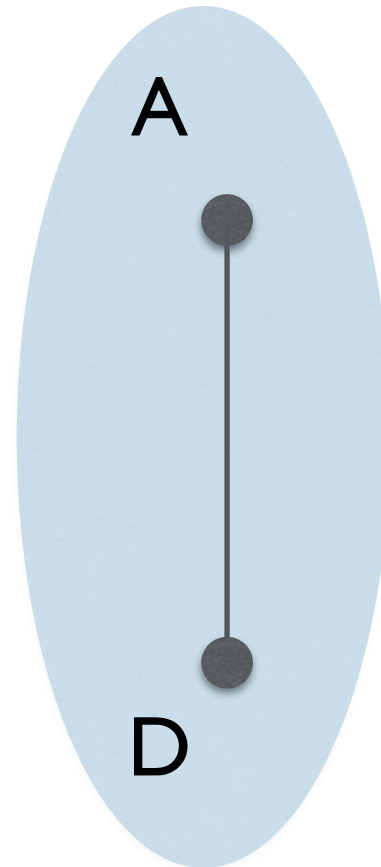


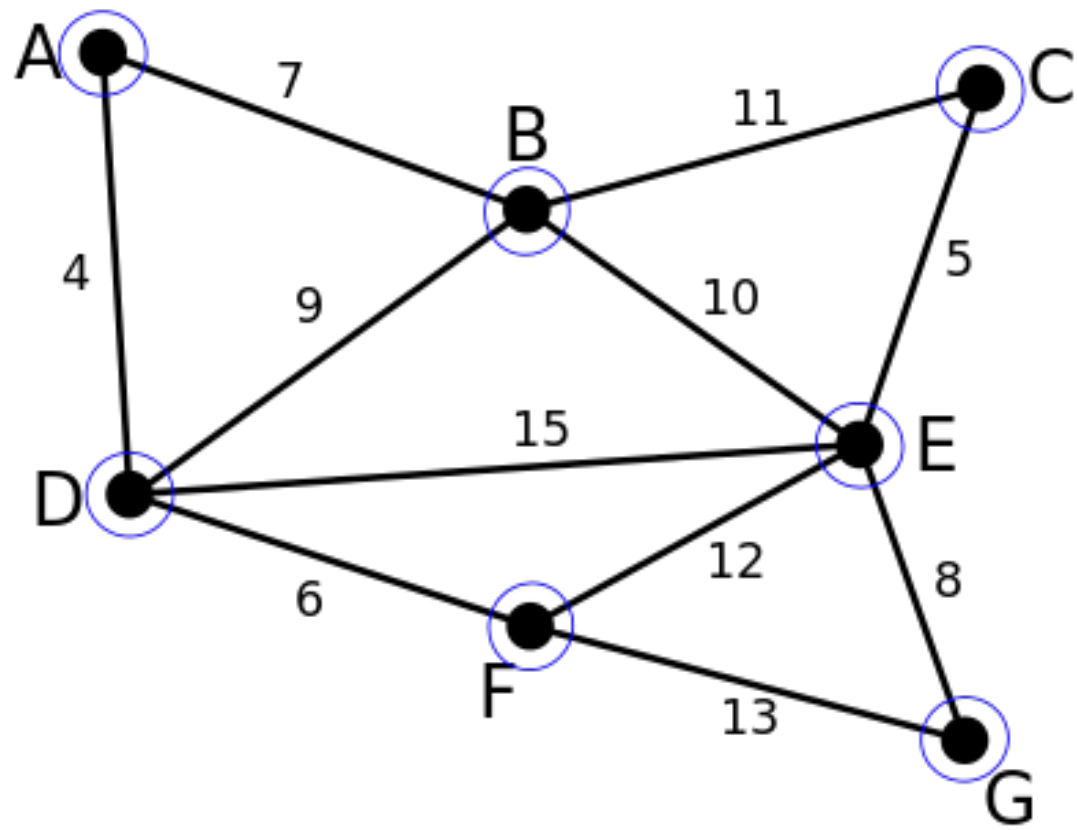


- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$

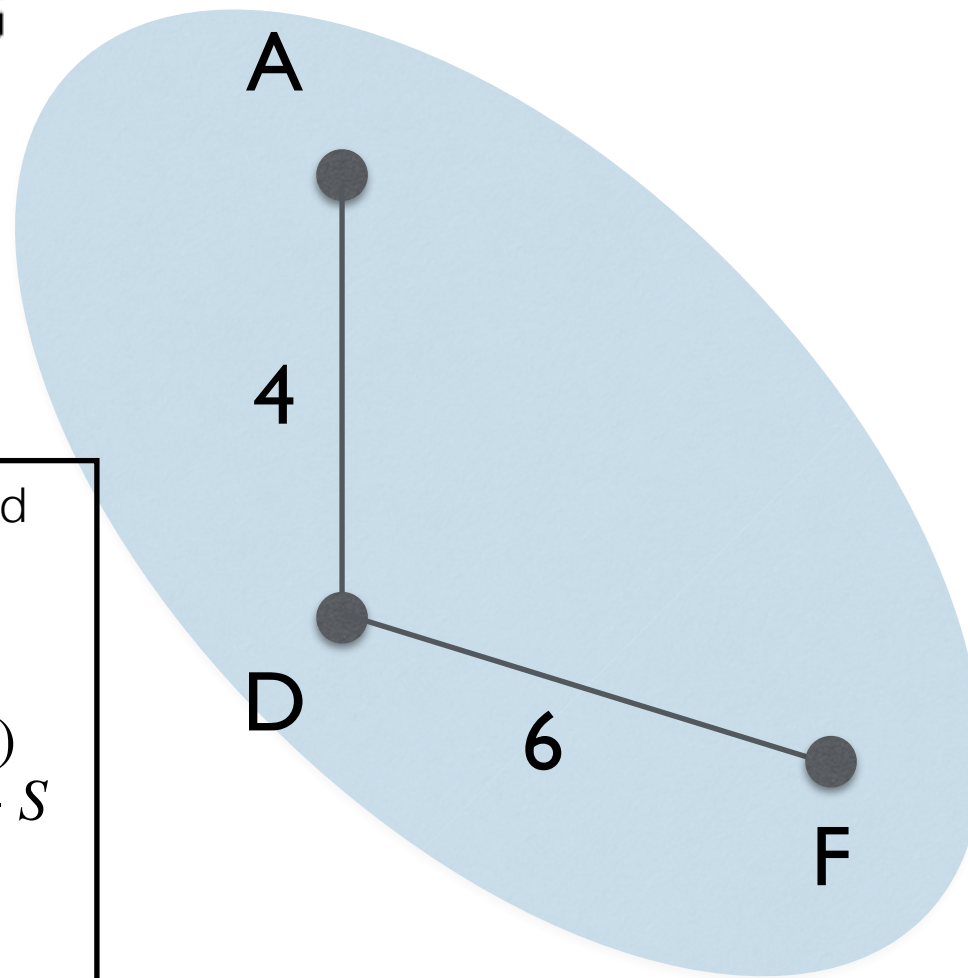


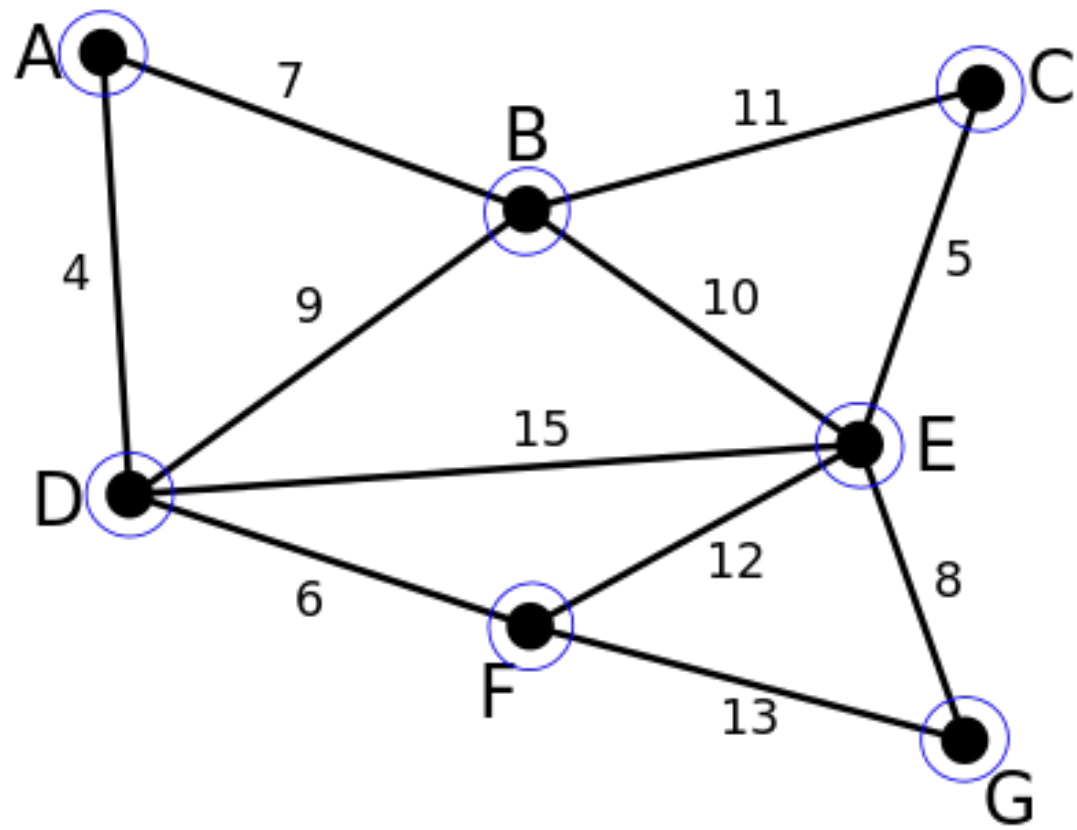
- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$



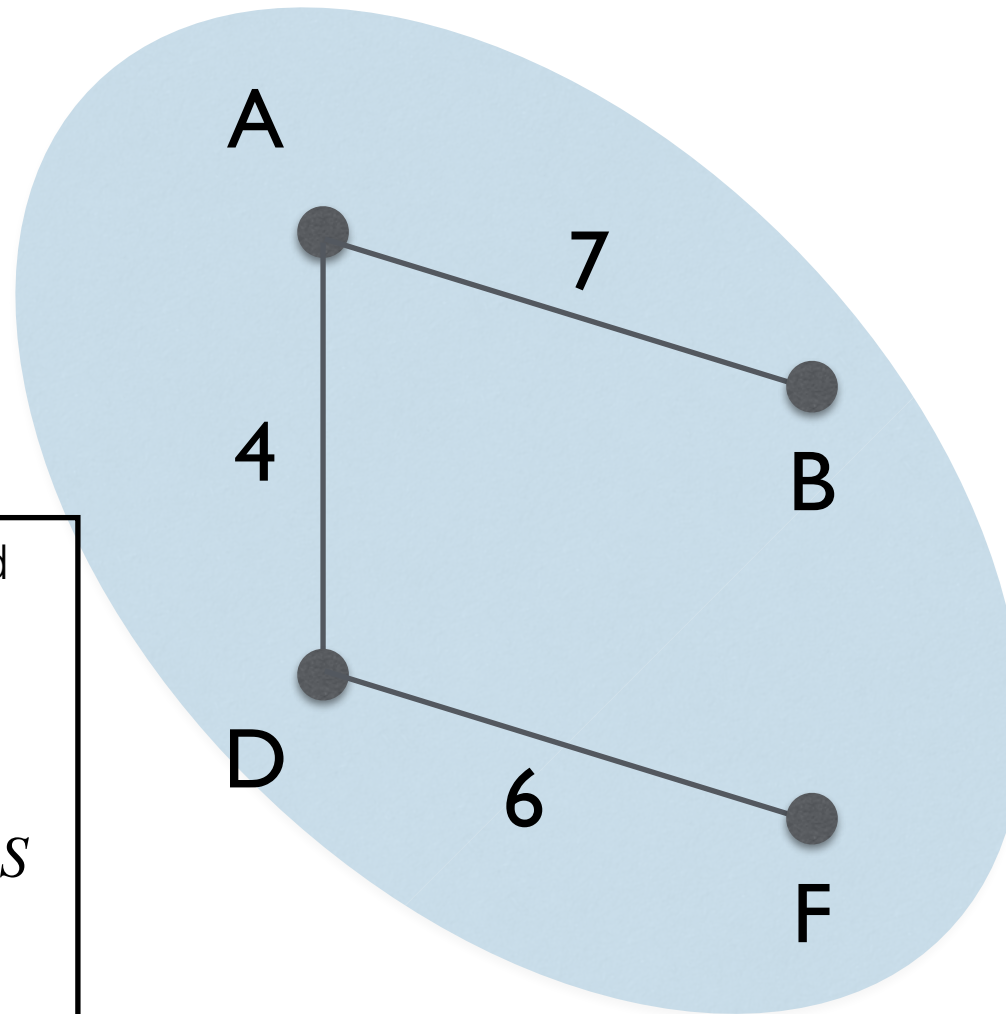


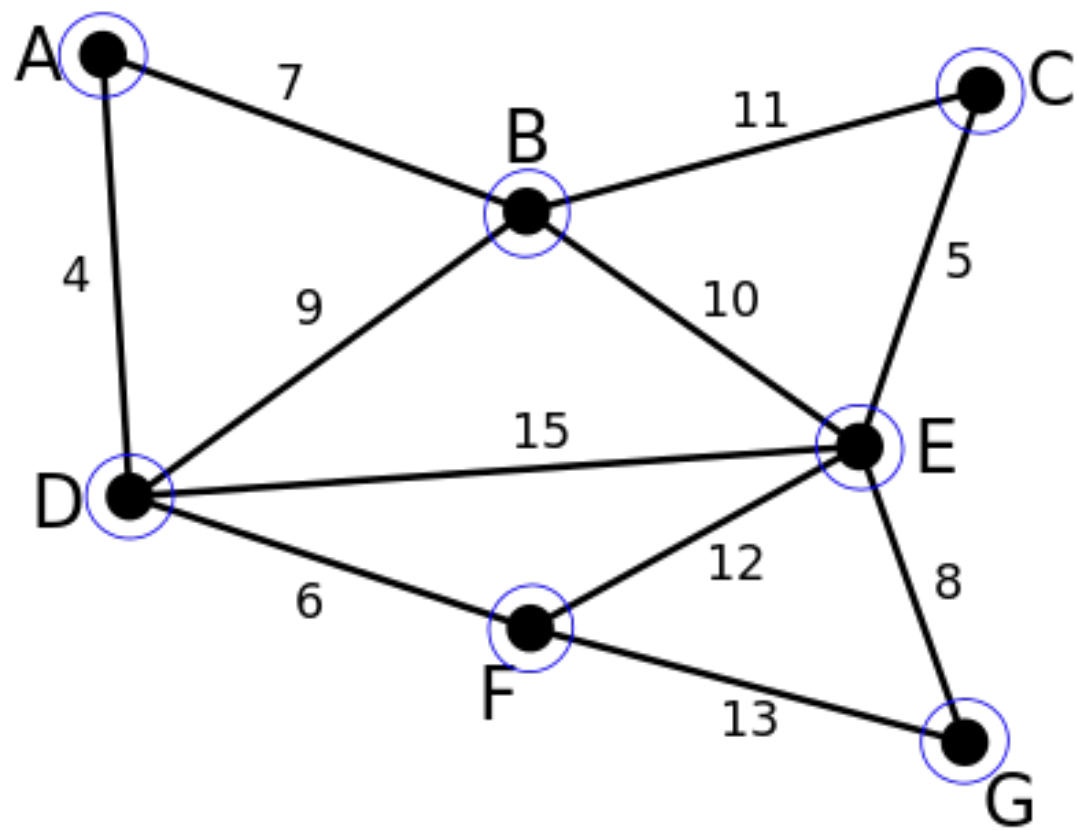
- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$



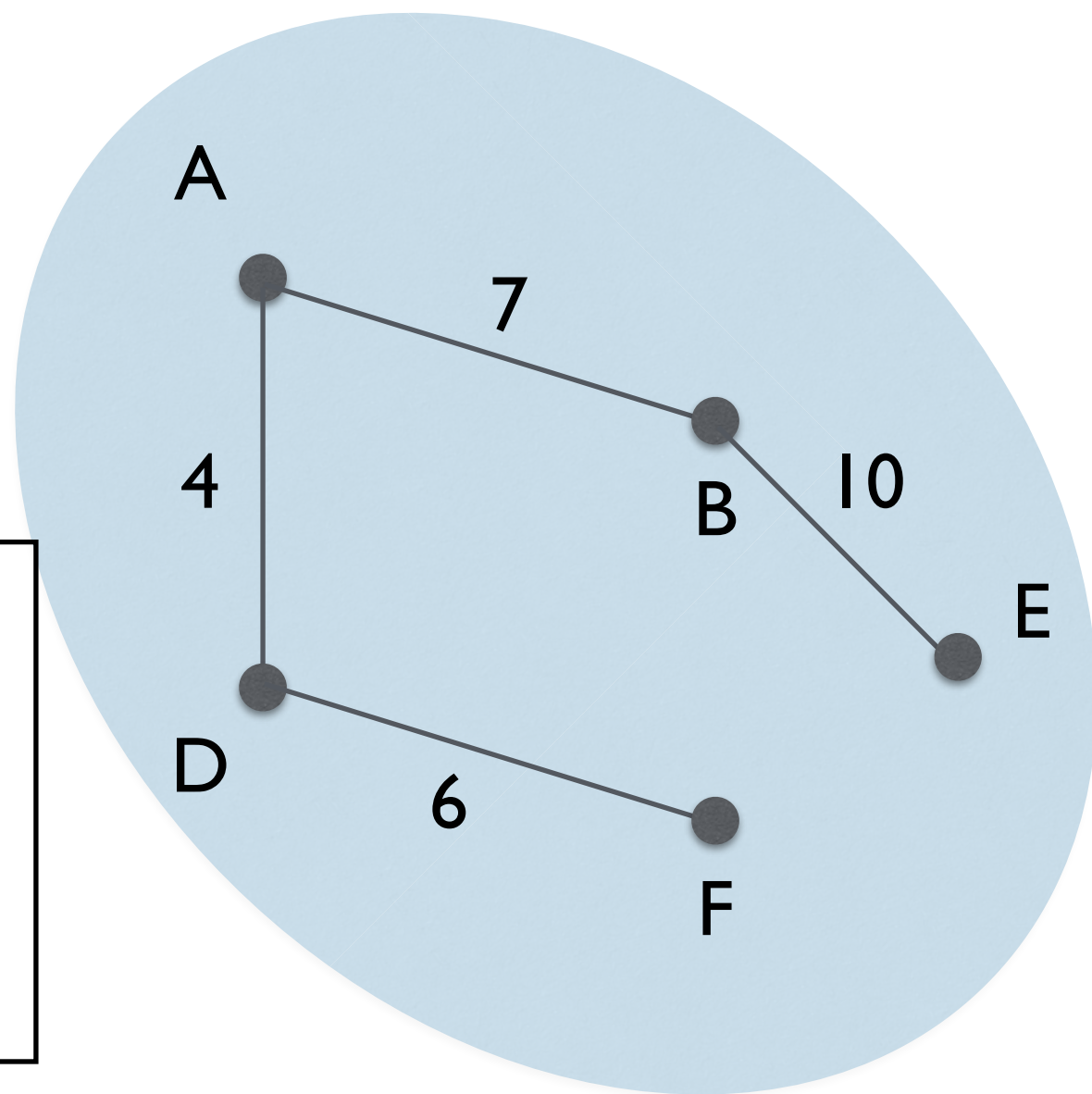


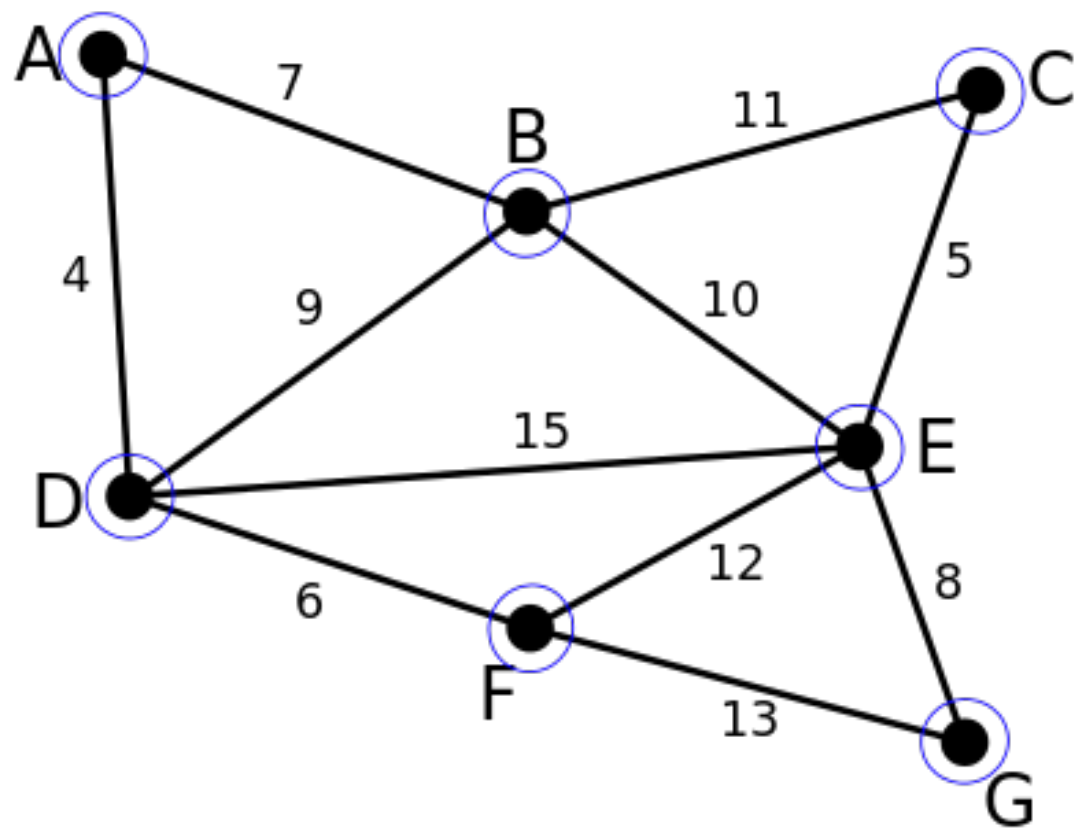
- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$



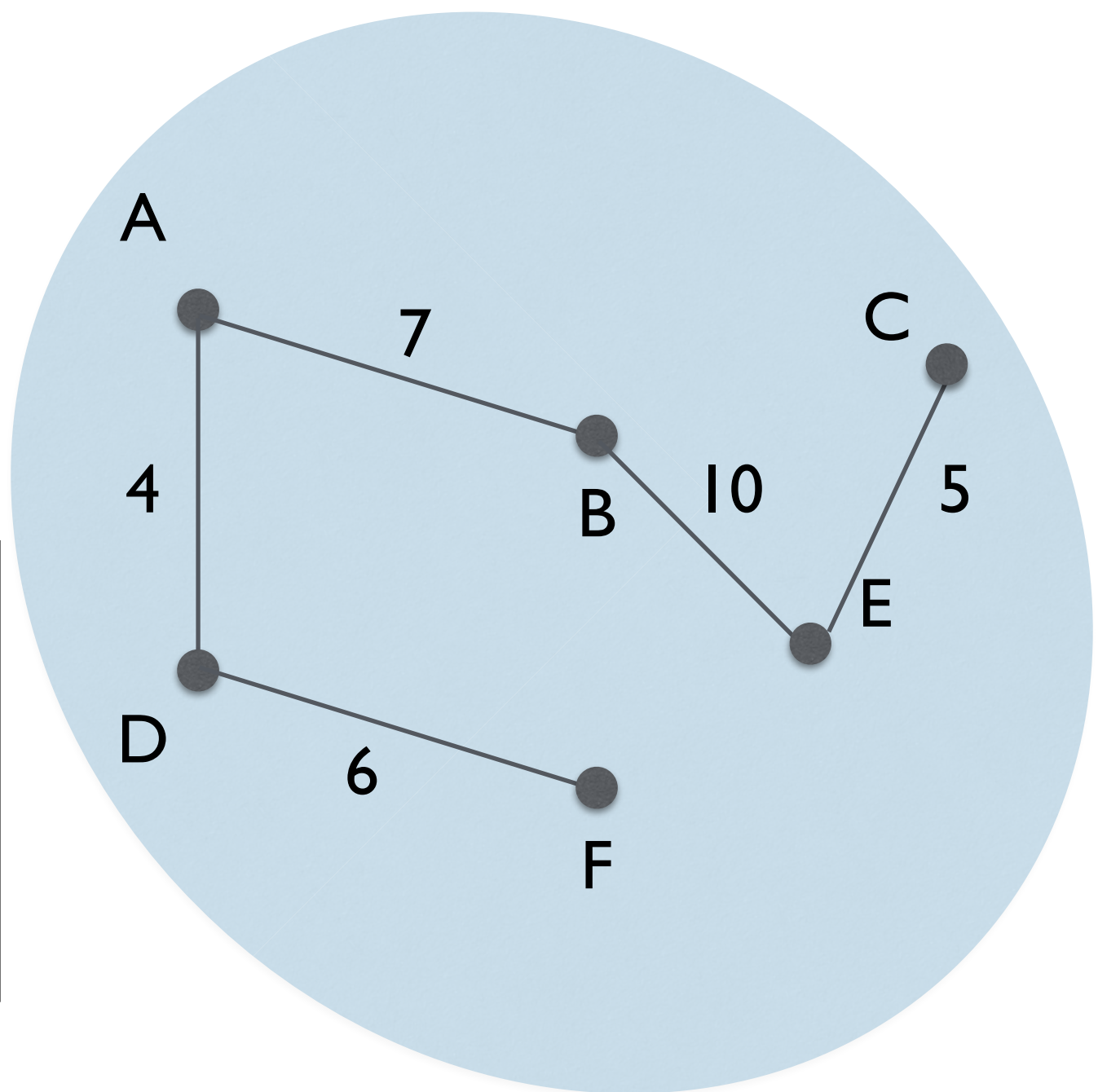


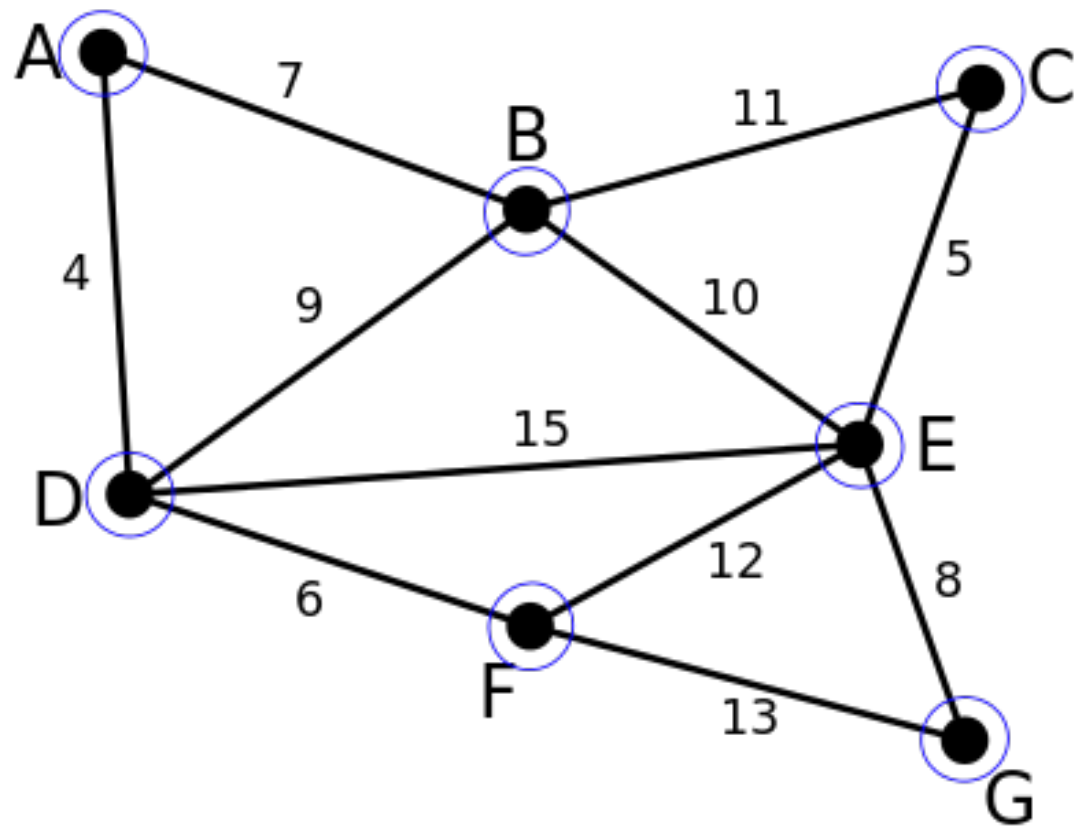
- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$





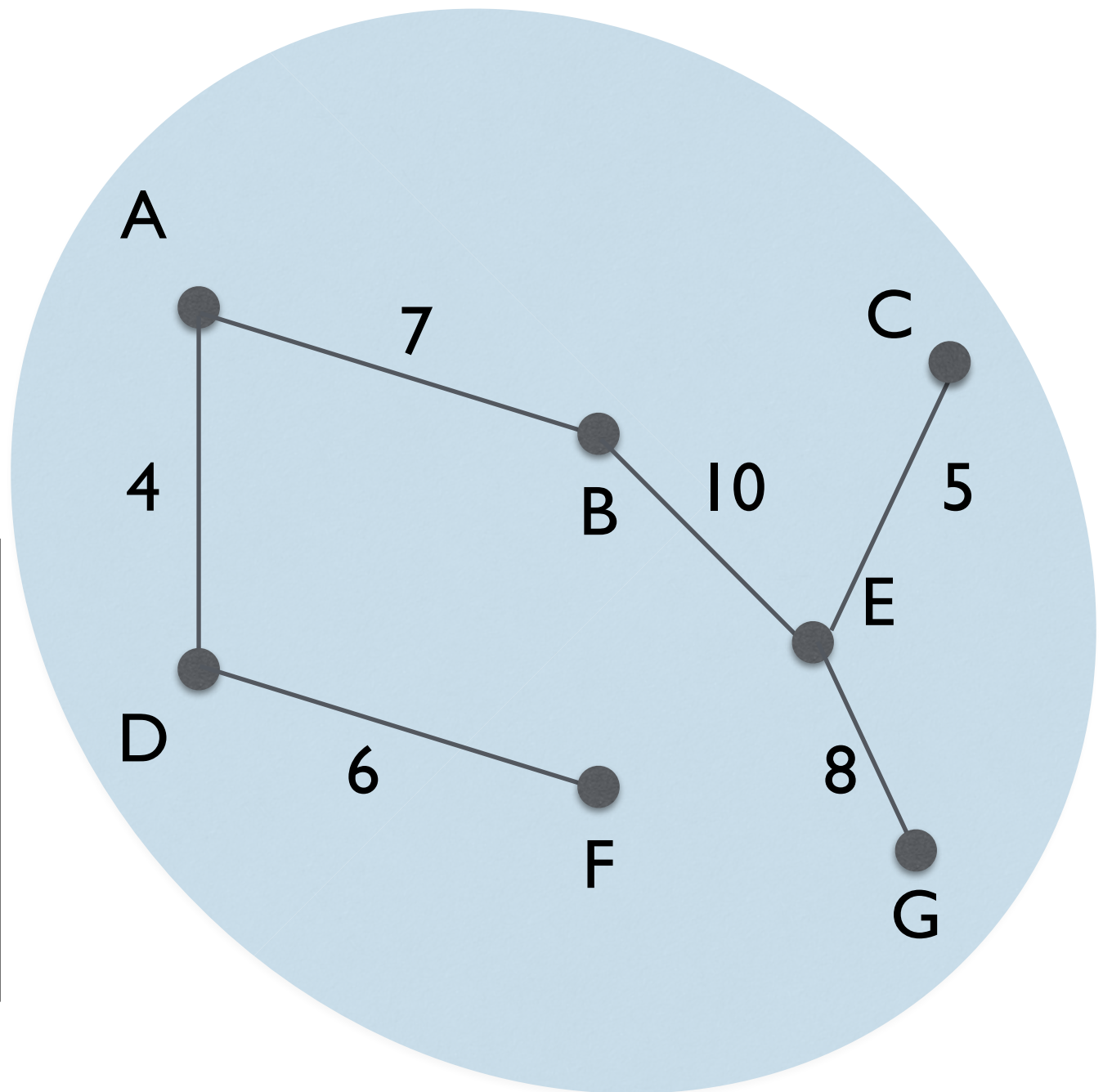
- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$





Total weight: 40

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$



Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
 - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
 - $T \leftarrow T \cup \{e\}$
 - $S \leftarrow S \cup \{v\}$
- Correctness:
 - Why is T a MST?

