Minimum_Spanning_Tree

June 16, 2022

[1]:

```
```

```
<IPython.core.display.HTML object>
```

[4]:

```
from IPython.display import Image
# Image(filename = "gd5.png", width = "100%")
```

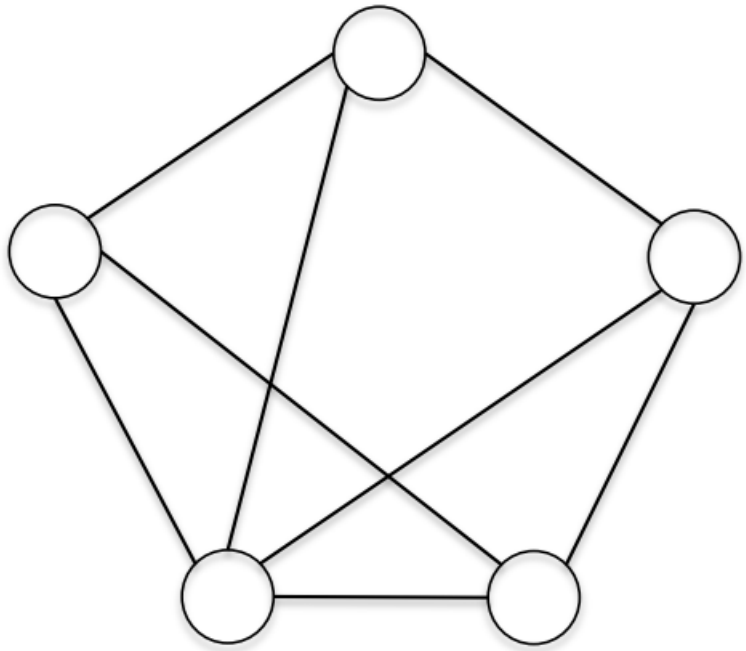
0.1 Minimum Spanning Tree problem

- The minimum spanning tree problem is about connecting a bunch of objects as cheaply as possible.
- The objects and connections could represent something physical, like computer servers and communication links between them.
- Objects and connections between them are most naturally modeled with graphs. A graph G = (V, E) has two ingredients: a set V of vertices and a set E of edges.
- Let us considers only undirected graphs, in which each edge e is an unordered pair {v, w} of vertices (written as e = (v, w) or e = (w, v)), which are called the endpoints of the edge.
- The numbers |V| and |E| of vertices and edges are usually denoted by n and m, respectively.

[6]:

```
Image(filename = "mst1.png", width = "50%")
```

[6]:

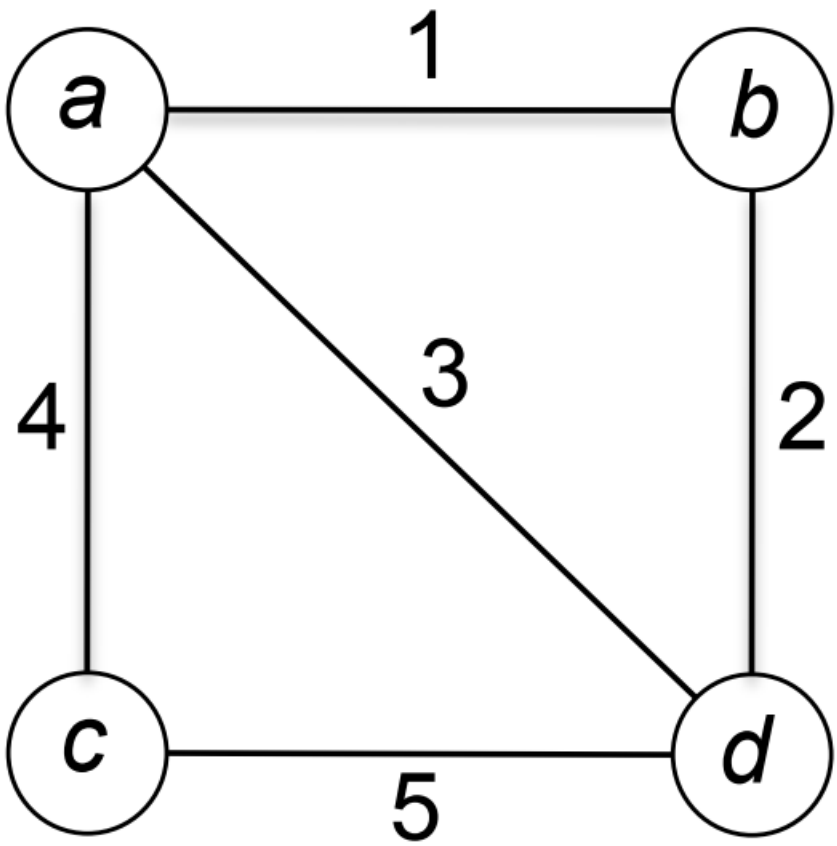


0.2 Spanning Trees

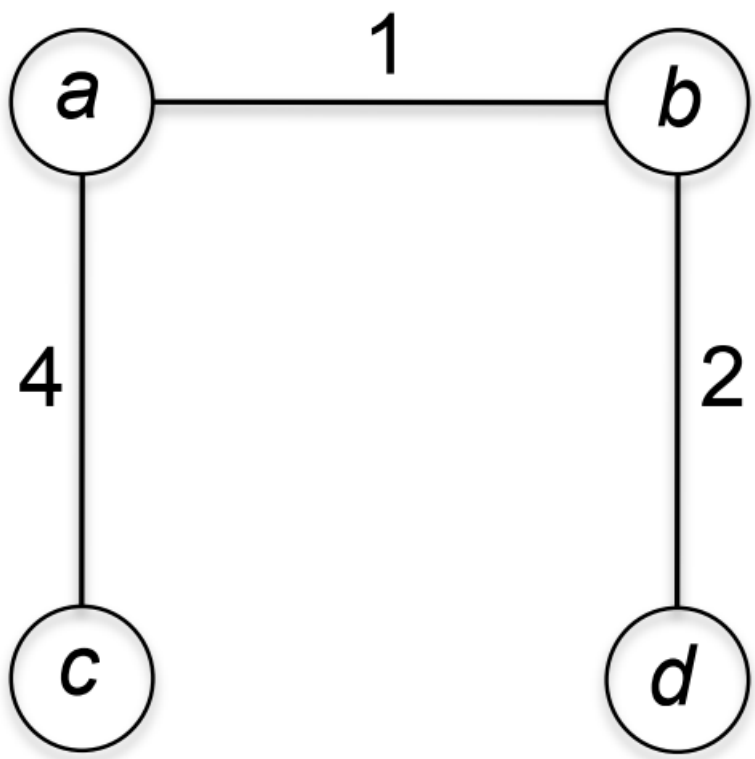
- The input in the minimum spanning tree problem is an undirected graph G = (V, E) in which each edge e has a real-valued cost c_e. (For example, c_e could indicate the cost of connecting two computer servers.)
- The goal is to compute a spanning tree of the graph with the minimum-possible sum of edge costs.
- By a spanning tree of G, we mean a subset $T \subset E$ of edges that satisfies two properties.
  - First, T should not contain a cycle (this is the ``tree'' part).
  - Second, for every pair $v, w \in V$ of vertices, T should include a path between v and w (this is the ``spanning'' part)
- It makes sense only to talk about spanning trees of connected graphs G = (V, E), in which it's possible to travel from any vertex $v \in V$ to any other vertex $w \in V$ using a path of edges in E.
- What is the minimum sum of edge costs of a spanning tree of the following graph? (Each edge is labeled with its cost.)

[8]:
```
Image(filename = "mst2.png", width = "30%")
```

[8]:



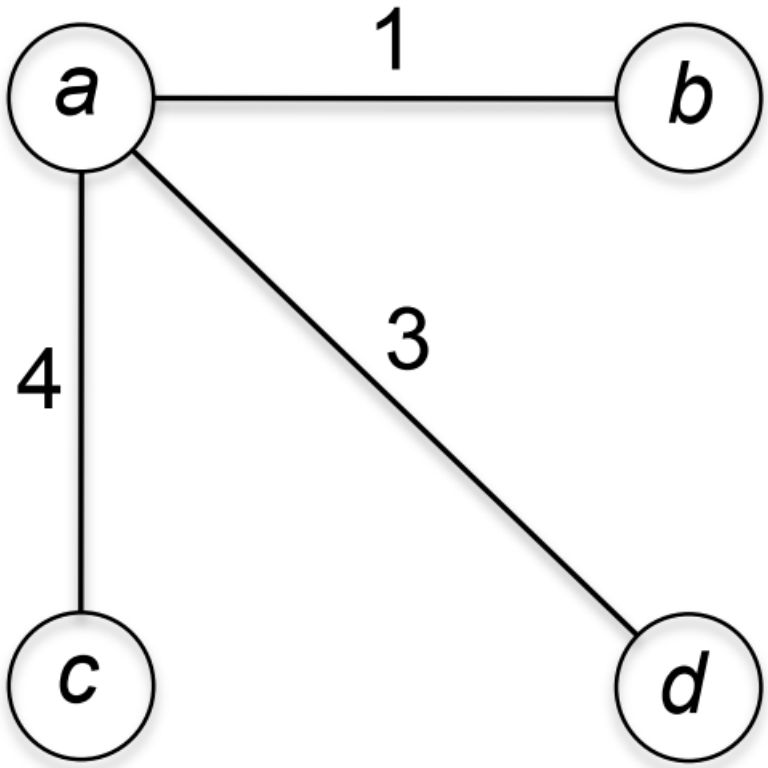
[12]:
```
Image(filename = "mst3.png", width = "30%")
```
[12]:




```
Total Cost = 7
```

[14]:
```
Image(filename = "mst4.png", width = "30%")
```

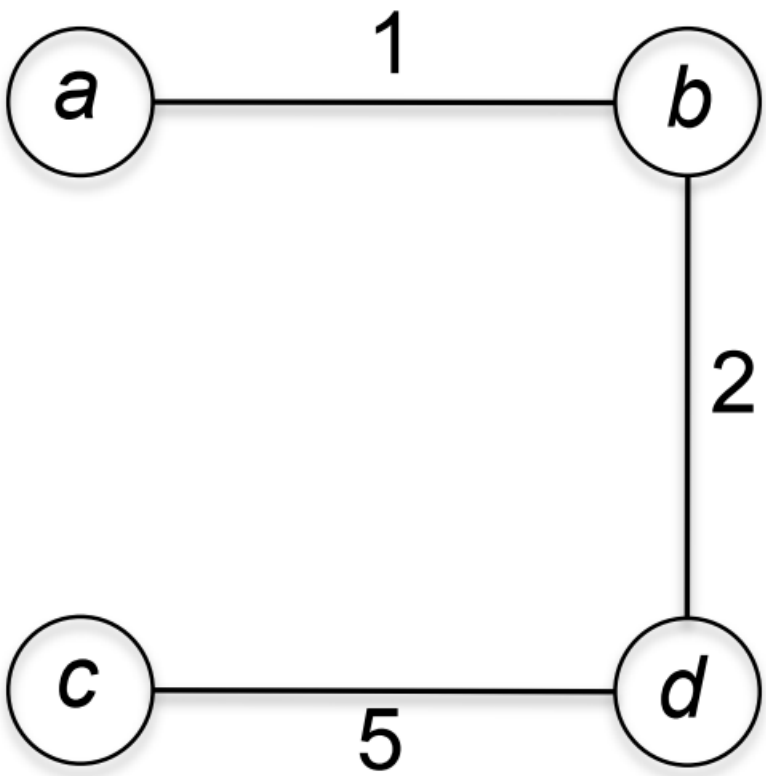
[14]:




```
Total Cost = 8
```

[13]:
```
Image(filename = "mst5.png", width = "30%")
```

[13]:



Total Cost = 8

- It's easy enough to compute the minimum spanning tree of a four-vertex graph like the one above; what about in general?
- Cayley's formula is a famous result from combinatorics stating that the n-vertex complete graph (in which all the n_{c_2} possible edges are present) has exactly n^{n-2} different spanning trees.
- This is bigger than the estimated number of atoms in the known universe when $n \geq 50$.

[7]:
```
Image(filename = "mst7.png", width = "100%")
```

[7]:

Problem: Minimum Spanning Tree (MST)

Input: A connected undirected graph $G = (V, E)$ and a real-valued cost c_e for each edge $e \in E$.

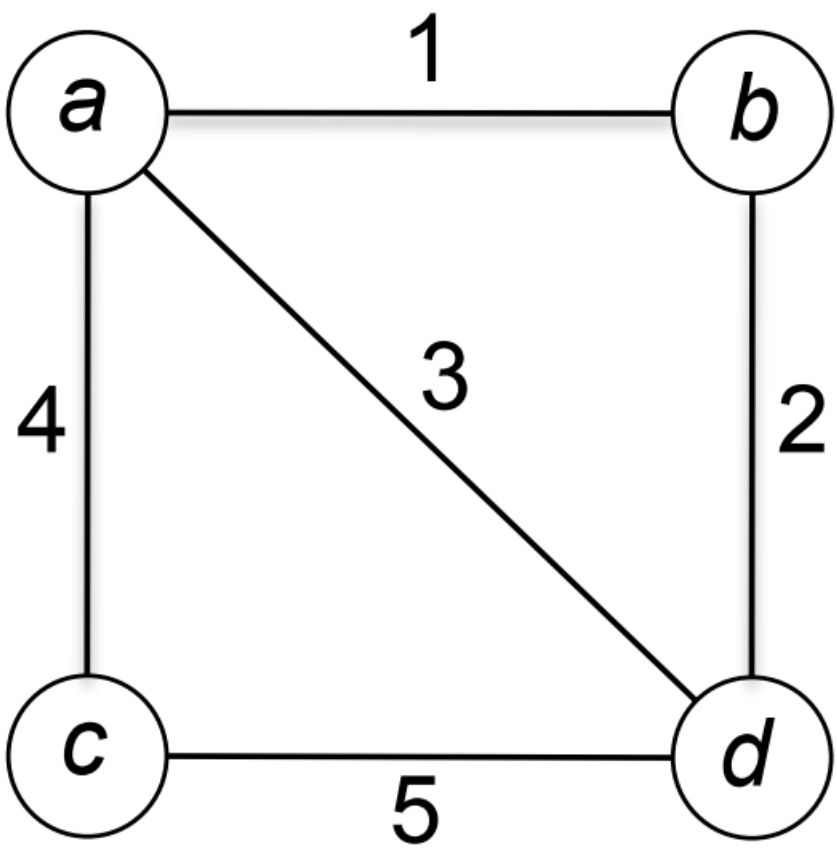
Output: A spanning tree $T \subseteq E$ of G with the minimum-possible sum $\sum_{e \in T} c_e$ of edge costs.[6]

0.3 Prim's algorithm

It is named after Robert C. Prim, who discovered the algorithm in 1957.

[11]:
```
Image(filename = "mst2.png", width = "50%")
```

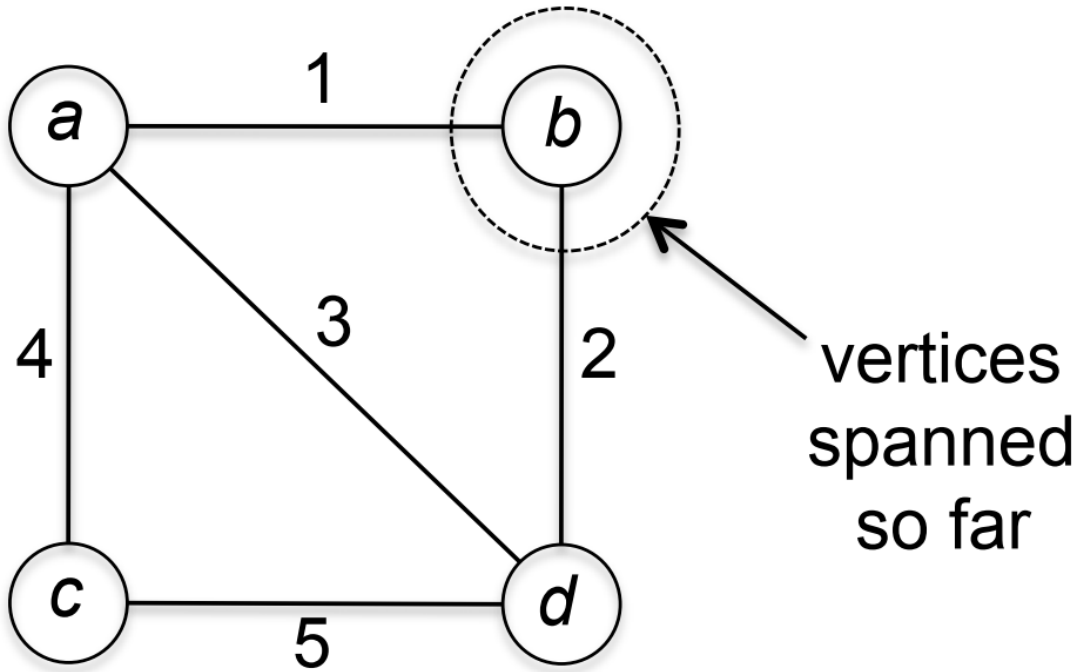
[11]:



Prim's algorithm begins by choosing an arbitrary vertex---let's say vertex b in our example. (In the end, it won't matter which one we pick.)

[21]:
```
Image(filename = "mst8.png", width = "50%")
```

[21]:



The plan is to construct a tree one edge at a time, starting from b and growing like a mold until the tree spans the entire vertex set.

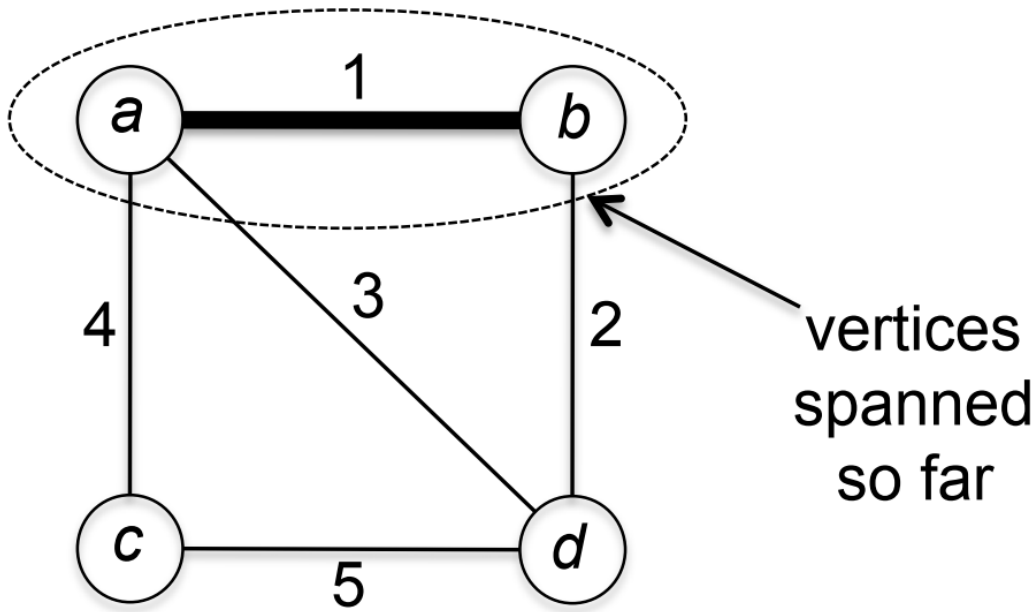
In each iteration, we'll greedily add the cheapest edge that extends the reach of the tree-so-far.

The algorithm's initial (empty) tree spans only the starting vertex b. There are two options for expanding its reach: the edge (a, b) and the edge (b, d).

The former is cheaper, so the algorithm chooses it. The tree-so-far spans the vertices a and b.

[20]:
```
Image(filename = "mst9.png", width = "50%")
```

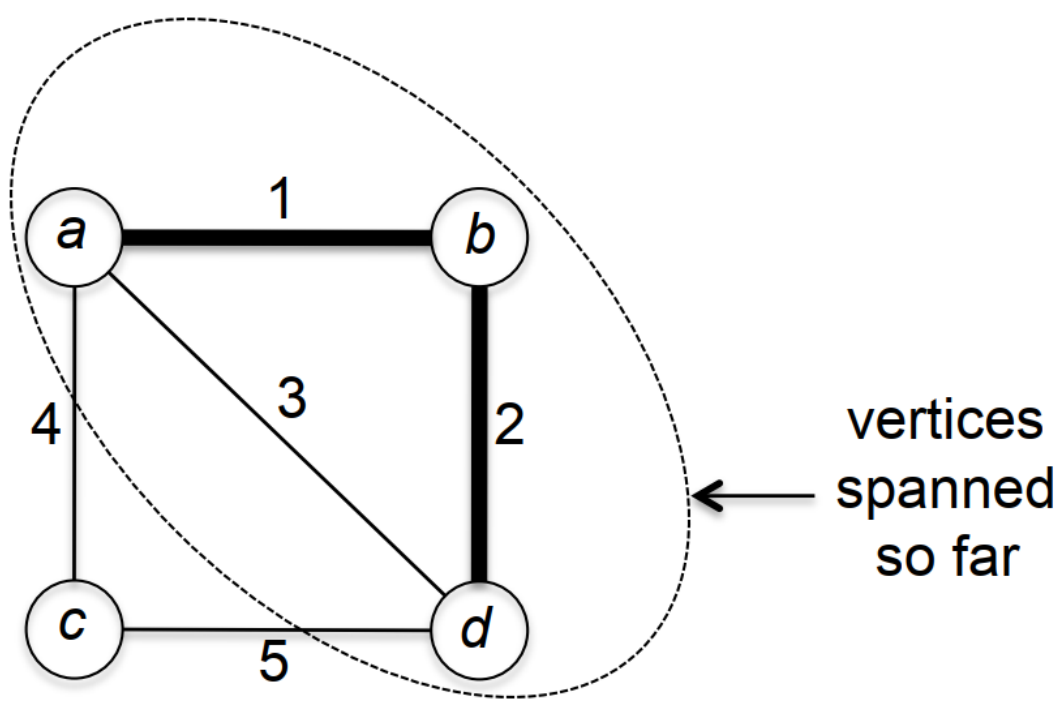
[20]:



In the second iteration, three edges would expand the tree's reach: (a, c), (a, d), and (b, d). The cheapest of these is (b, d). After its addition, the tree-so-far spans a, b, and d.

[19]:
```
Image(filename = "mst10.png", width = "50%")
```

[19]:

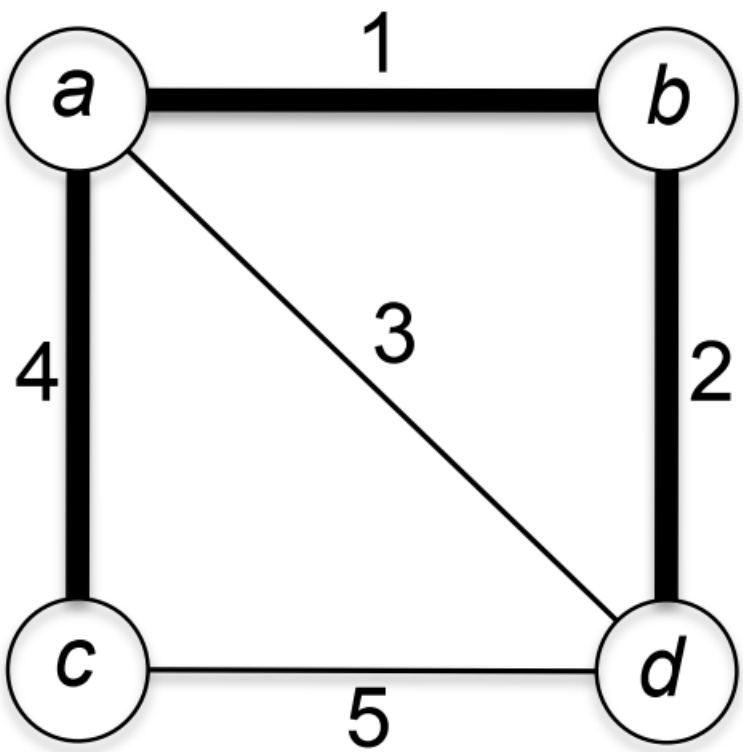


Both endpoints of the edge (a, d) have been sucked into the set of vertices spanned so far; adding this edge in the future would create a cycle, so the algorithm does not consider it further.

In the final iteration, there are two options for expanding the tree's reach to c, the edges (a, c) and (c, d): Prim's algorithm chooses the cheaper edge (a, c), resulting in the same minimum spanning tree identified earlier.

[17]:
```
Image(filename = "mst11.png", width = "30%")
```

[17]:



[18]:
```
Image(filename = "mst12.png", width = "100%")
```

[18]:

Prim

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

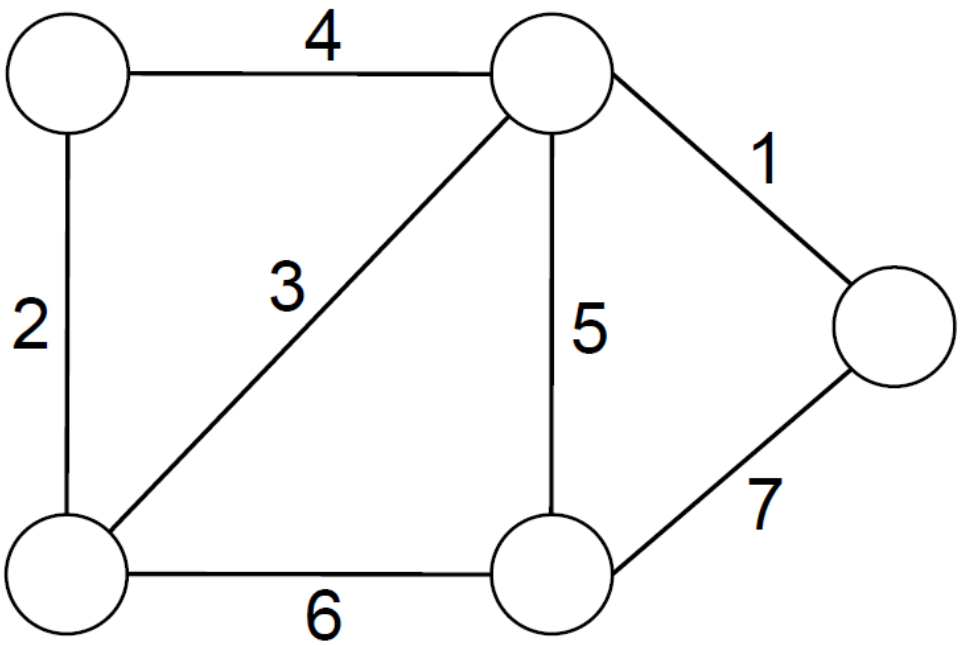
```
// Initialization
X := {s}    // s is an arbitrarily chosen vertex
T := ∅      // invariant:  the edges in T span X
// Main loop
while there is an edge (v, w) with v ∈ X, w ∉ X do
    (v*, w*) := a minimum-cost such edge
    add vertex w* to X
    add edge (v*, w*) to T
return T
```

After n - 1 iterations, the algorithm runs out of new edges. So the number of iterations is O(n) and each ieration takes O(m) time, the overall running time is O(mn).

0.4 Kruskal's Algorithm

[5]:
```
Image(filename = "mst13.png", width = "50%")
```

[5]:

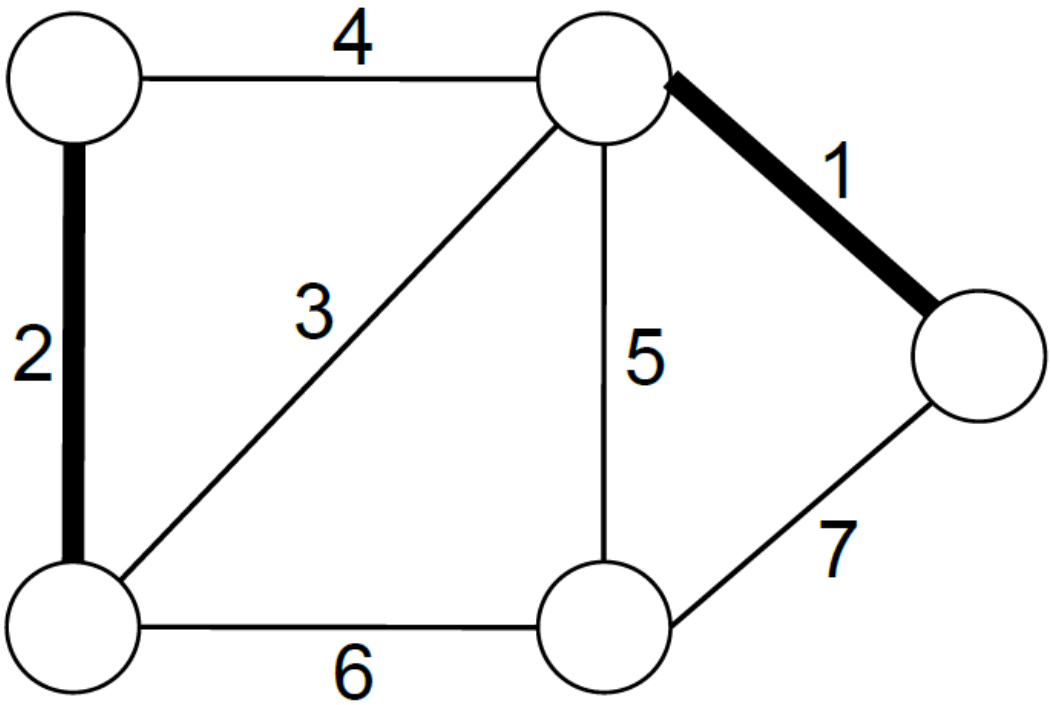


Kruskal's algorithm starts with an empty edge set T and, in its first iteration, greedily considers the cheapest edge (the edge of cost 1) and adds it to T.

The second iteration follows suit with the next-cheapest edge (the edge of cost 2). At this point, the solution-so-far T looks like:

[6]:
```
Image(filename = "mst14.png", width = "50%")
```

[6]:

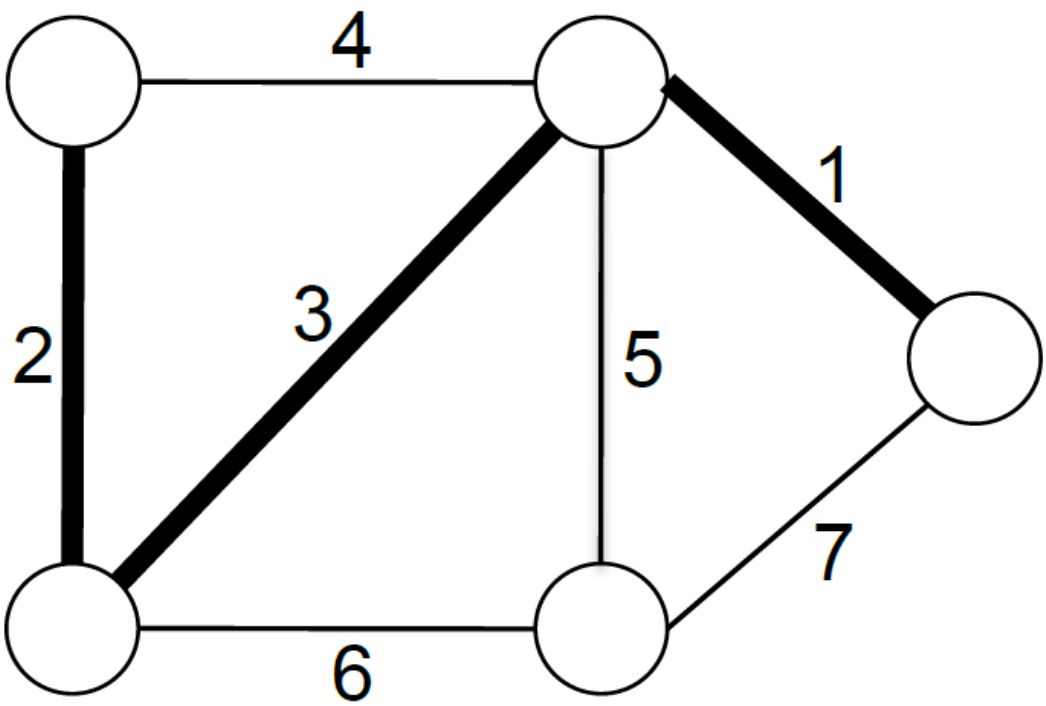


The two edges chosen so far are disjoint, so the algorithm is effectively growing two trees in parallel.

The next iteration considers the edge with cost 3. Its inclusion does not create a cycle and also happens to fuse the two trees-so-far into one:

[7]:
```
Image(filename = "mst15.png", width = "50%")
```

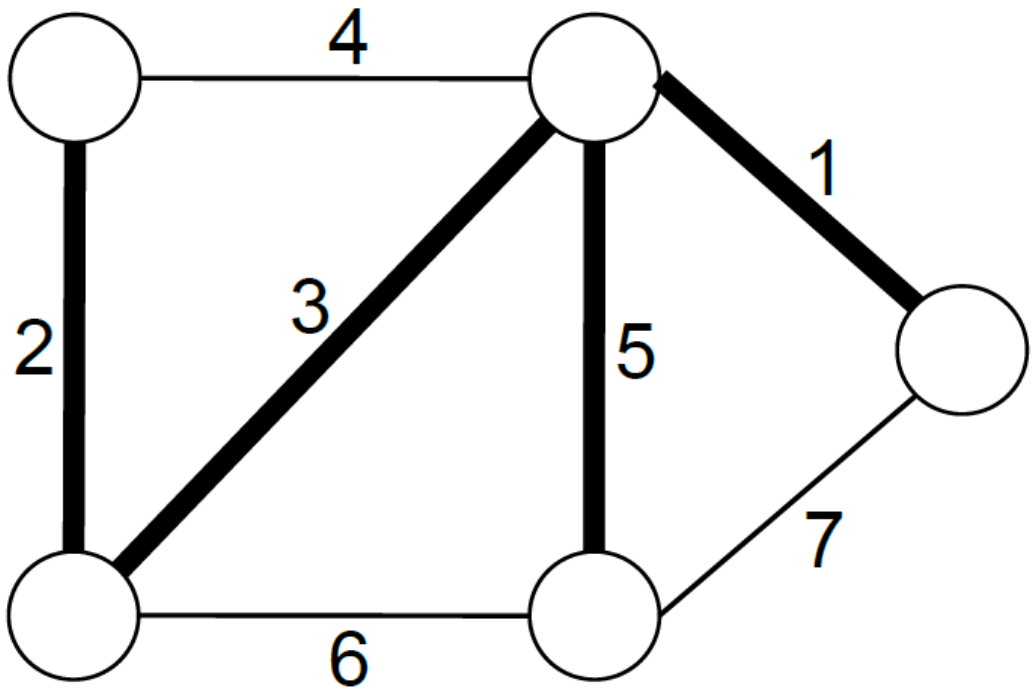
[7]:



The algorithm next considers the edge of cost 4. Adding this edge to T would create a cycle (with the edges of cost 2 and 3), so the algorithm is forced to skip it. The next-best option is the edge of cost 5; its inclusion does not create a cycle and, in fact, results in a spanning tree:

[8]:
```
Image(filename = "mst16.png", width = "50%")
```

[8]:



The algorithm skips the edge of cost 6 (which would create a triangle with the edges of cost 3 and 5) as well as the final edge, of cost 7 (which would create a triangle with the edges of cost 1 and 5).

The final output above is the minimum spanning tree of the graph.

[10]:
```
Image(filename = "mst17.png", width = "100%")
```

[10]:

```
Kruskal

Input: connected undirected graph G = (V, E) in
  adjacency-list representation and a cost c_e for each
  edge e ∈ E.
Output: the edges of a minimum spanning tree of G.

---

// Preprocessing
T := ∅
sort edges of E by cost   // e.g., using MergeSort[26]
// Main loop
for each e ∈ E, in nondecreasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

In the preprocessing step, the algorithm sorts the edge array of the input graph, which has m entries. With a good sorting algorithm (like MergeSort), this step contributes O(mlog n) work to the overall running time.

The number of edges of an n-vertex connected graph with no parallel edges is at least n - 1 (achieved by a tree) and at most n_{c_2}= n(n-1)/2 (achieved by a complete graph). Thus log m lies between log(n - 1) and 2 log n for every connected graph with no parallel edges, which justifies using log m and log n interchangeably inside a big-O expression.

The main loop has m iterations. Each iteration is responsible for checking whether the edge e = (v,w) under examination can be added to the solution-so-far T without creating a cycle. This condition can be checked in linear time using any reasonable graph search algorithm, like breadth- or depth-first search starting from v in O(n) time. The per-iteration running time is therefore O(n), for an overall running time of main loop is O(mn).

Thus the total running time is O(mn).