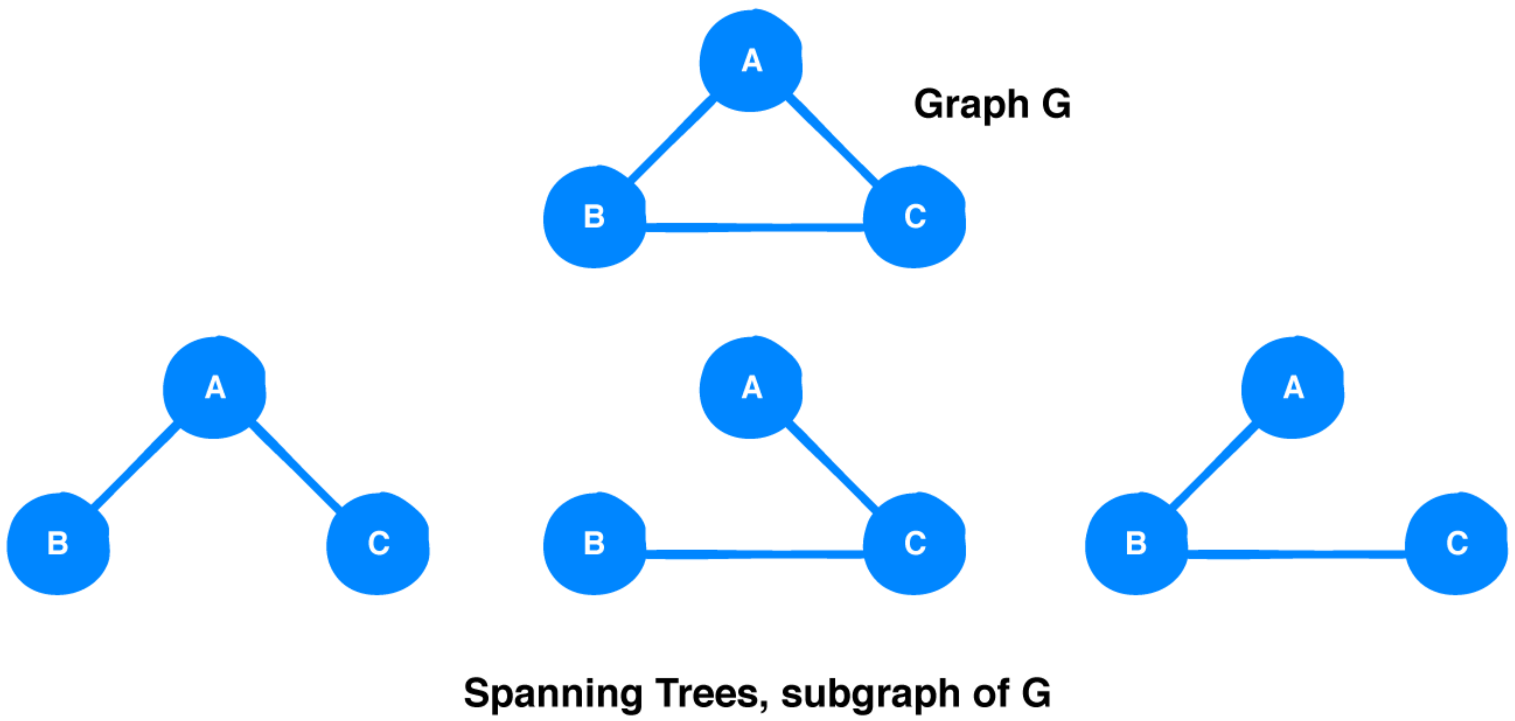
23 Prim's Algorithm Written by Irina Galata

In previous chapters, you've looked at depth-first and breadth-first search algorithms. These algorithms form **spanning trees**.

A **spanning tree** is a subgraph of an undirected graph, containing all of the graph's vertices, connected with the fewest number of edges. A spanning tree cannot contain a cycle and cannot be disconnected.

Here's an example of some spanning trees:

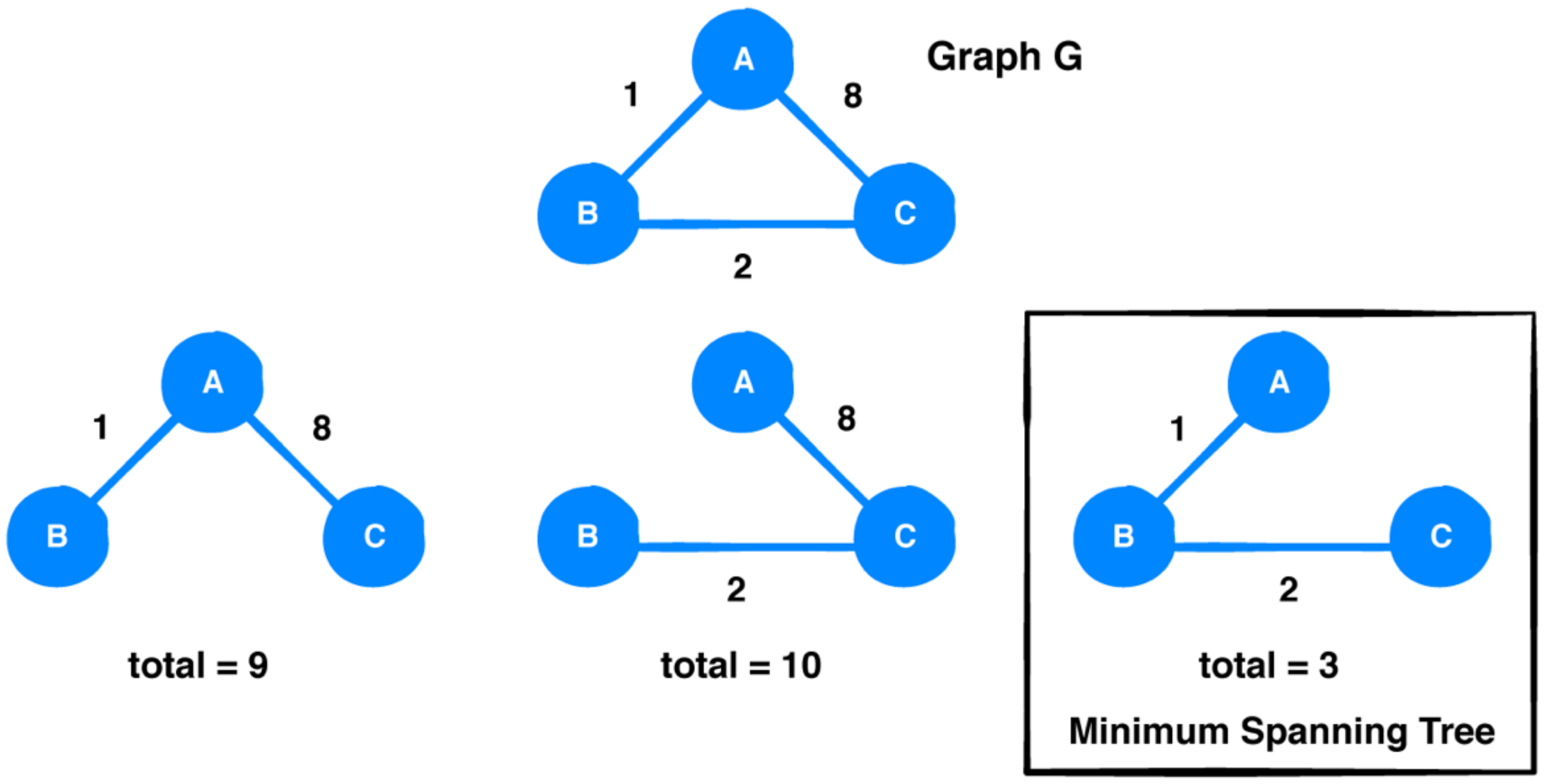


From this undirected graph that forms a triangle, you can generate three different spanning trees in which you require only two edges to connect all vertices.

In this chapter, you will look at **Prim's algorithm**, a greedy algorithm used to construct a **minimum spanning tree**. A **greedy** algorithm constructs a solution step-by-step and picks the most optimal path at every step.

A minimum spanning tree is a spanning tree with weighted edges in which the total weight of all edges is minimized. For example, you might want to find the cheapest way to lay out a network of water pipes.

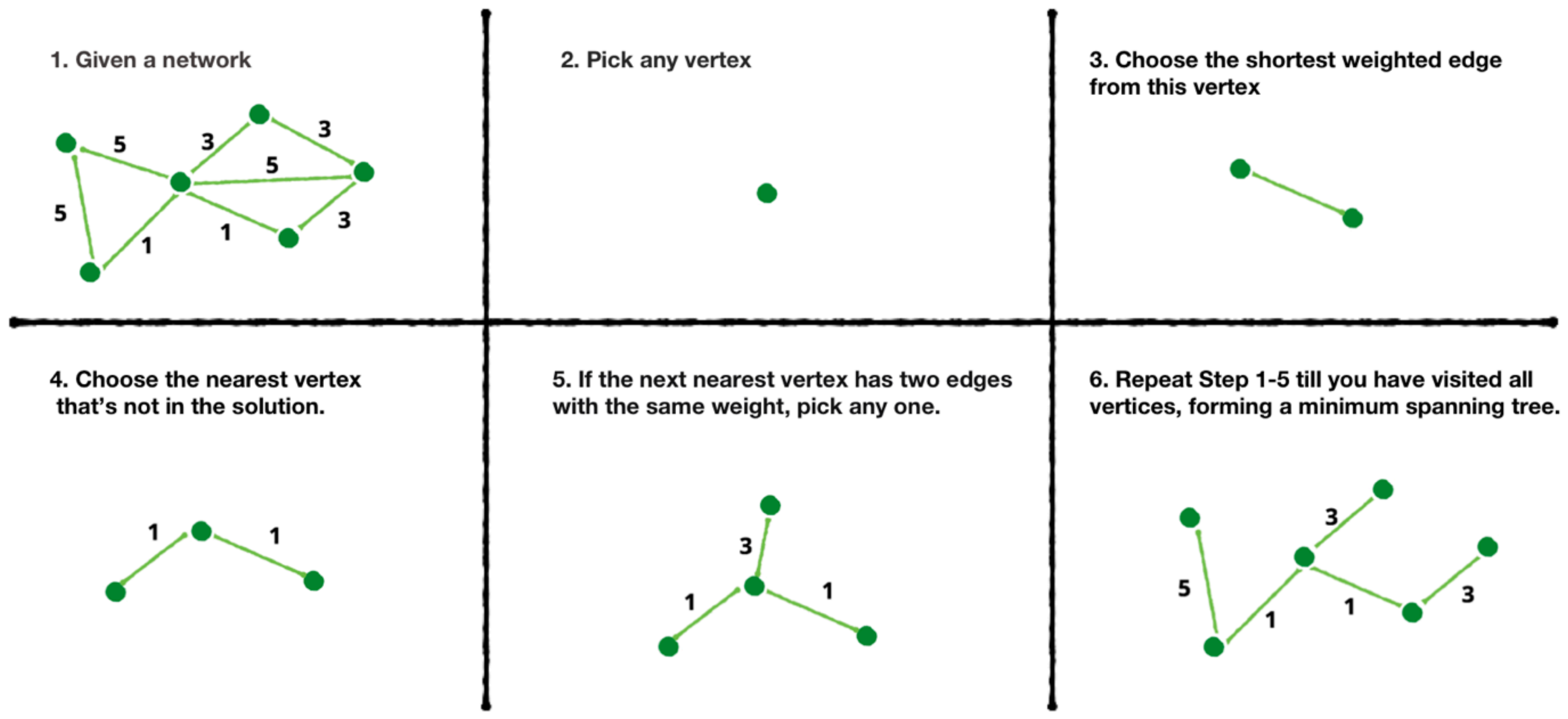
Here's an example of a minimum spanning tree for a weighted undirected graph:



Notice that only the third subgraph forms a minimum spanning tree, since it has the minimum total cost of `3`.

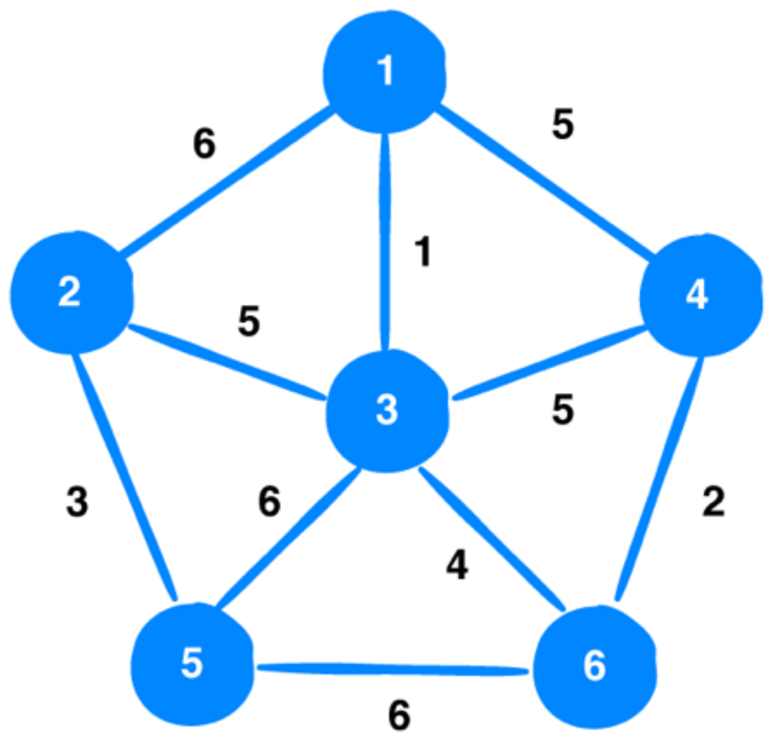
Prim's algorithm creates a minimum spanning tree by choosing edges one at a time. It's greedy because, every time you pick an edge, you pick the smallest weighted edge that connects a pair of vertices.

There are six steps to finding a minimum spanning tree with Prim's algorithm:

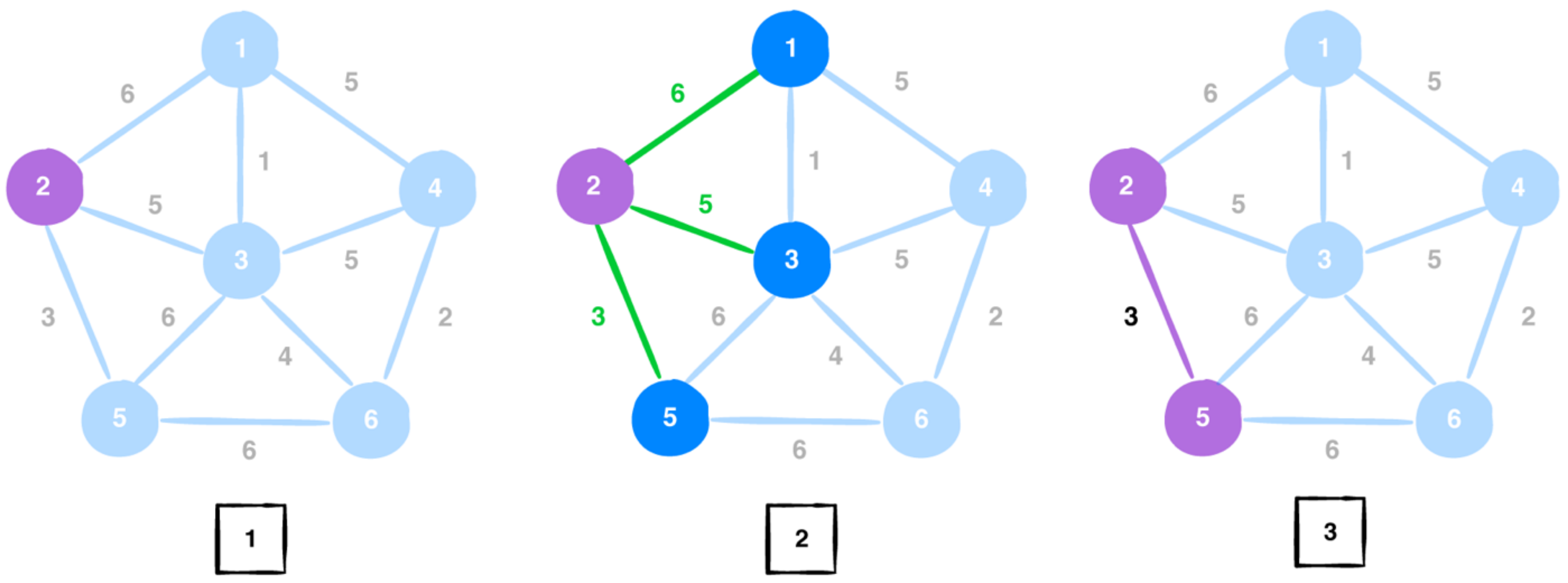

1. Given a network
5
3
3
5
5
1
1
3
2. Pick any vertex
3. Choose the shortest weighted edge from this vertex
4. Choose the nearest vertex that's not in the solution.
1
1
5. If the next nearest vertex has two edges with the same weight, pick any one.
3
1
1
6. Repeat Step 1-5 till you have visited all vertices, forming a minimum spanning tree.
3
5
1
1
3

Example

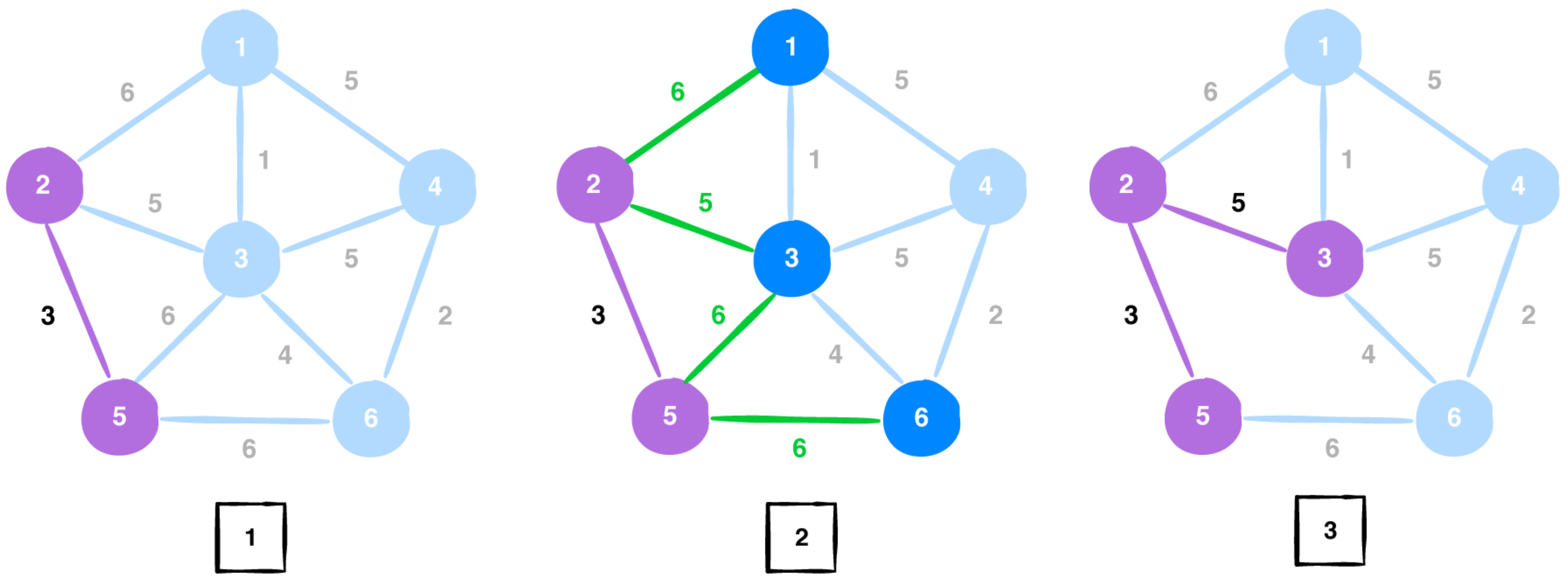
Imagine the graph below represents a network of airports. The vertices are the airports, and the edges between them represent the cost of fuel to fly an airplane from one airport to the next.



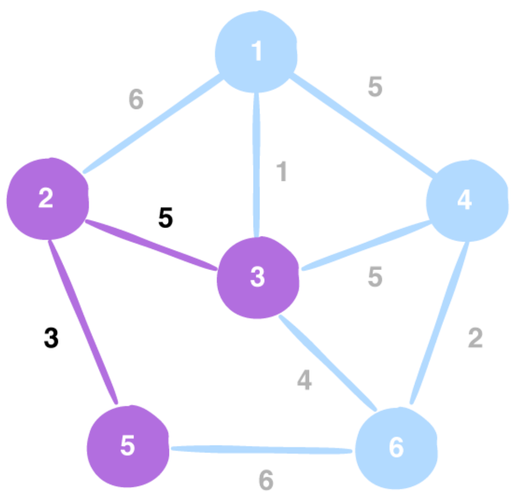
Let's start working through the example:



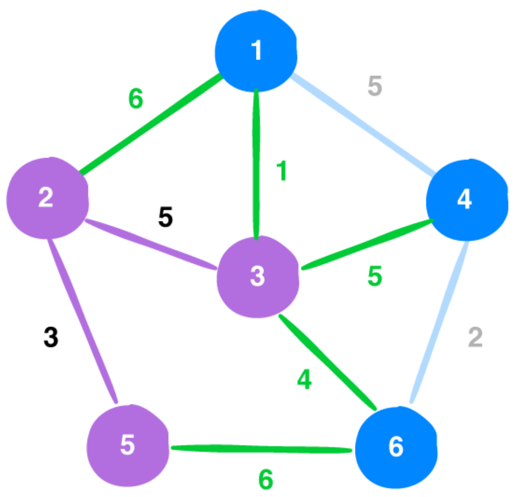
1. Choose any vertex in the graph. Let's assume you chose **vertex 2**.
2. This vertex has edges with weights **[6, 5, 3]**. A greedy algorithm chooses the smallest-weighted edge.
3. Choose the edge that has a weight of **3** and is connected to **vertex 5**.



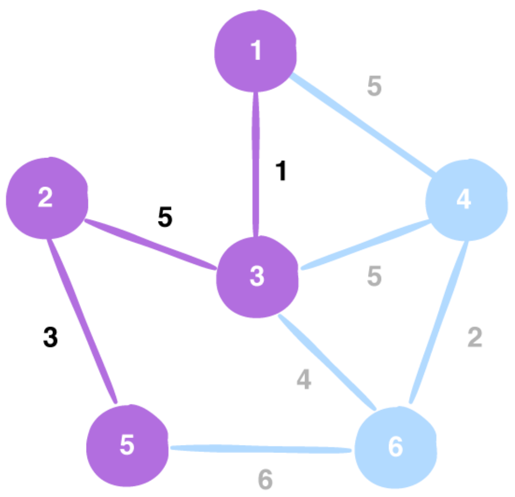
1. The explored vertices are **{2, 5}**.
2. Choose the next shortest edge from the explored vertices. The edges are **[6, 5, 6, 6]**. You choose the edge with weight **5**, which is connected to **vertex 3**.
3. Notice that the edge between **vertex 5** and **vertex 3** can be removed since both are already part of the spanning tree.



1

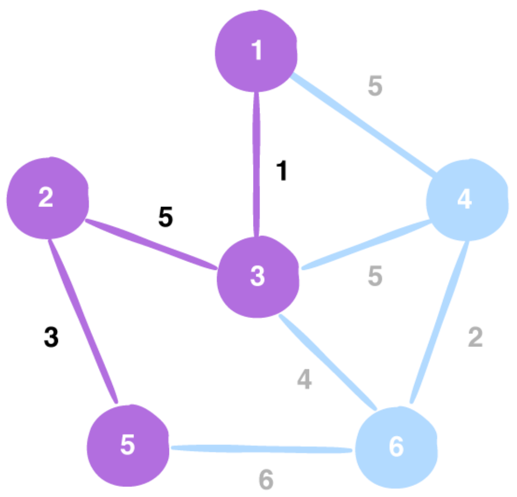


2

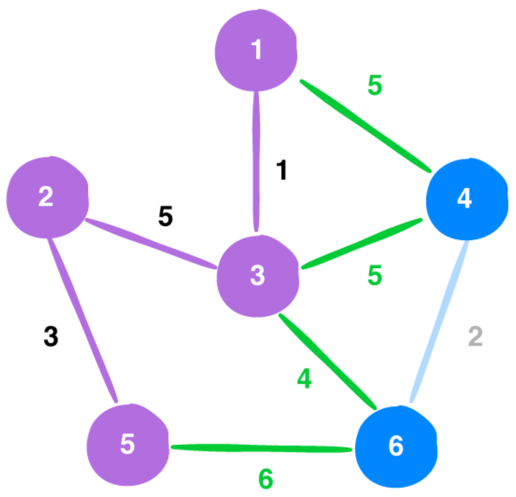


3

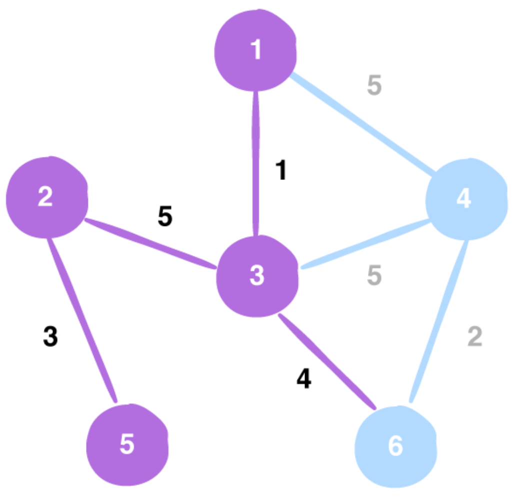
1. The explored vertices are **{2, 3, 5}**.
2. The next potential edges are **[6, 1, 5, 4, 6]**. You choose the edge with weight **1**, which is connected to **vertex 1**.
3. The edge between **vertex 2** and **vertex 1** can be removed.



1

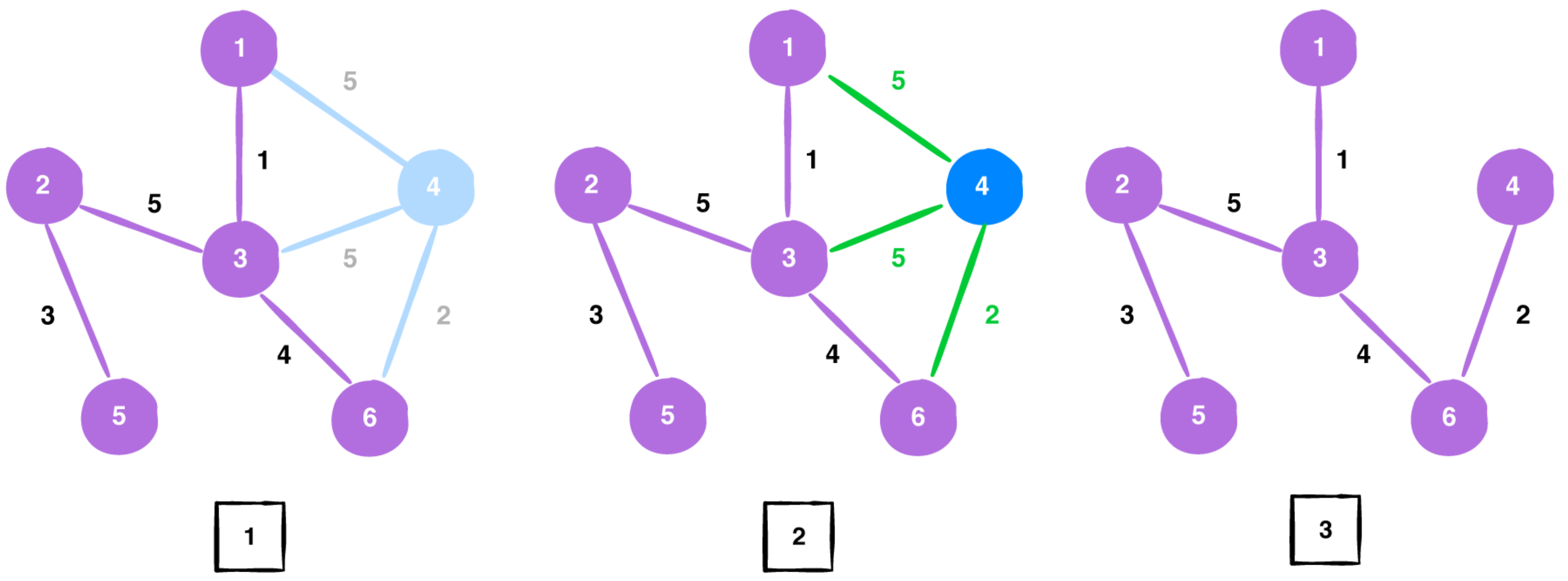


2



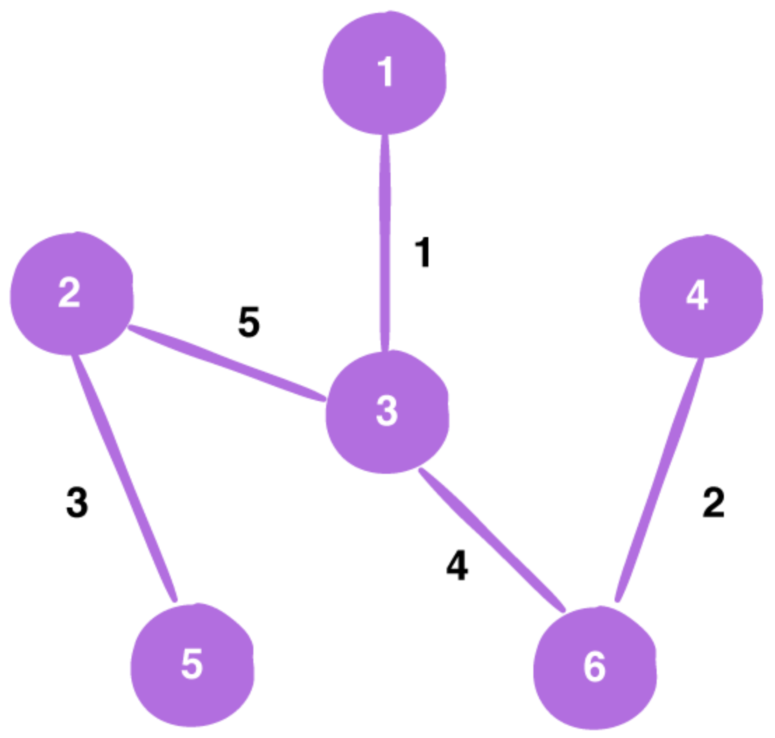
3

1. The explored vertices are **{2, 3, 5, 1}**.
2. Choose the next shortest edge from the explored vertices. The edges are **[5, 5, 4, 6]**. You choose the edge with weight **4**, which is connected to **vertex 6**.
3. The edge between **vertex 5** and **vertex 6** can be removed.



1. The explored vertices are **{2, 5, 3, 1, 6}**.
2. Choose the next shortest edge from the explored vertices. The edges are **[5, 5, 2]**. You choose the edge with weight **2**, which is connected to **vertex 4**.
3. The edges **[5, 5]** connected to **vertex 4** from **vertex 1** and **vertex 3** can be removed.

> **Note**: If all edges have the same weight, you can pick any one of them.



This is the minimum spanning tree from our example produced from Prim's algorithm.

Next, let's see how to build this in code.

Implementation

Open up the starter project for this chapter. This project comes with an adjacency list graph and a priority queue, which you will use to implement Prim's algorithm.

The priority queue is used to store the edges of the explored vertices. It's a min-priority queue so that every time you dequeue an edge, it gives you the edge with the smallest weight.

Start by defining an `object` called `Prim`. Create it in the **Prim.kt** file:

```
object Prim
```

Helper methods

Before building the algorithm, you'll create some helper methods to keep you organized and consolidate duplicate code.

Copying a graph

To create a minimum spanning tree, you must include all vertices from the original graph. Open up **AdjacencyList.kt** and add the following to class `AdjacencyList`:

```
val vertices: Set<Vertex<T>>
  get() = adjacencies.keys
```

This property returns the set of vertices that your `AdjacencyList` is currently storing.

Next, add the following method:

```
fun copyVertices(graph: AdjacencyList<T>) {
  graph.vertices.forEach {
    adjacencies[it] = arrayListOf()
  }
}
```

This copies all of a graph's vertices into a new graph.

Finding edges

Besides copying the graph's vertices, you also need to find and store the edges of every vertex you explore. Open up **Prim.kt** and add the following to `Prim`:

```
private fun <T: Any> addAvailableEdges(
    vertex: Vertex<T>,
    graph: Graph<T>,
    visited: Set<Vertex<T>>,
    priorityQueue: AbstractPriorityQueue<Edge<T>>
) {
  graph.edges(vertex).forEach { edge -> // 1
    if (edge.destination !in visited) { // 2
      priorityQueue.enqueue(edge) // 3
    }
  }
}
```

This method takes in four parameters:

1. The current `vertex`.
2. The `graph`, wherein the current `vertex` is stored.
3. The vertices that have already been visited.
4. The priority queue to add all potential edges.

Within the function, you do the following:

1. Look at every edge adjacent to the current `vertex`.
2. Check to see if the `destination` vertex has already been visited.
3. If it has not been visited, you add the edge to the priority queue.

Now that we have established our helper methods, let's implement Prim's algorithm.

Producing a minimum spanning tree

Add the following method to `Prim`:

```
fun <T: Any> produceMinimumSpanningTree(
    graph: AdjacencyList<T>
): Pair<Double, AdjacencyList<T>> { // 1
  var cost = 0.0 // 2
  val mst = AdjacencyList<T>() // 3
  val visited = mutableSetOf<Vertex<T>>() // 4
  val comparator = Comparator<Edge<T>> { first, second -> // 5
    val firstWeight = first.weight ?: 0.0
    val secondWeight = second.weight ?: 0.0
    (secondWeight - firstWeight).roundToInt()
  }
  val priorityQueue = ComparatorPriorityQueueImpl(comparator) // 6

  // to be continued
}
```

Here's what you have so far:

1. `produceMinimumSpanningTree` takes an undirected graph and returns a minimum spanning tree and its cost.
2. `cost` keeps track of the total weight of the edges in the minimum spanning tree.
3. This is a graph that will become your minimum spanning tree.
4. `visited` stores all vertices that have already been visited.
5. You create the `Comparator<Edge<T>>` to use for the priority queue.
6. This is a min-priority queue to store edges.

Next, continue implementing `produceMinimumSpanningTree` with the following:

```
mst.copyVertices(graph) // 1

val start = graph.vertices.firstOrNull() ?: return Pair(cost, mst) // 2

visited.add(start) // 3
addAvailableEdges(start, graph, visited, priorityQueue) // 4
```

This code initiates the algorithm:

1. Copy all the vertices from the original graph to the minimum spanning tree.
2. Get the starting vertex from the graph.
3. Mark the starting vertex as visited.
4. Add all potential edges from the `start` vertex into the priority queue.

Finally, complete `produceMinimumSpanningTree` with:

```
while (true) {
  val smallestEdge = priorityQueue.dequeue() ?: break // 1
  val vertex = smallestEdge.destination // 2
  if (visited.contains(vertex)) continue // 3

  visited.add(vertex) // 4
  cost += smallestEdge.weight ?: 0.0 // 5

  mst.add(EdgeType.UNDIRECTED, smallestEdge.source,  smallestEdge.destinati

  addAvailableEdges(vertex, graph, visited, priorityQueue) // 7
}

return Pair(cost, mst) // 8
```

Going over the code:

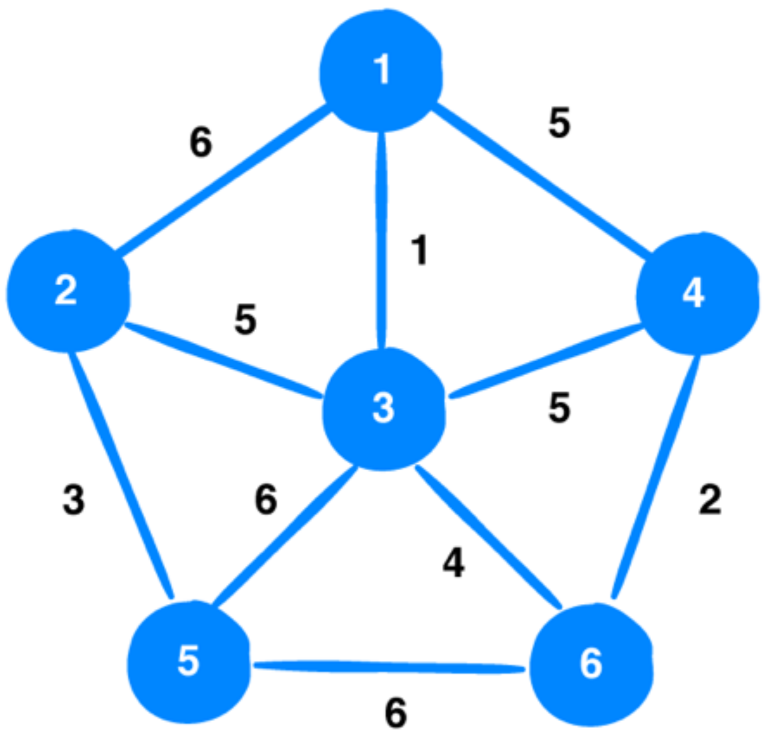
1. Continue Prim's algorithm until the queue of edges is empty.
2. Get the `destination` vertex.
3. If this vertex has been visited, restart the loop and get the next

smallest edge.

4. Mark the `destination` vertex as visited.
5. Add the edge's `weight` to the total `cost`.
6. Add the smallest edge into the minimum spanning tree you are constructing.
7. Add the available edges from the current vertex.
8. Once the `priorityQueue` is empty, return the minimum cost, and minimum spanning tree.

Testing your code

Navigate to the `main()` function, and you'll see the graph on the next page has been already constructed using an adjacency list.

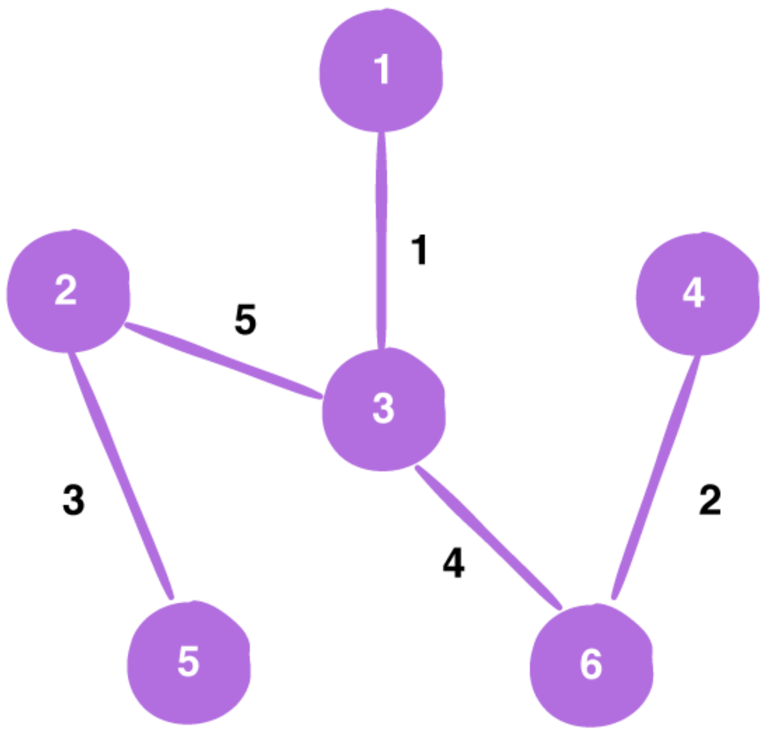


It's time to see Prim's algorithm in action. Add the following code:

```
val (cost, mst) = Prim.produceMinimumSpanningTree(graph)
println("cost: $cost")
println("mst:")
println(mst)
```

This constructs a graph from the example section. You'll see the following output:

```
cost: 15.0
mst:
3 ---> [ 1, 6, 2 ]
4 ---> [ 6 ]
1 ---> [ 3 ]
5 ---> [ 2 ]
2 ---> [ 3, 5 ]
6 ---> [ 3, 4 ]
```



Performance

In the algorithm above, you maintain three data structures:

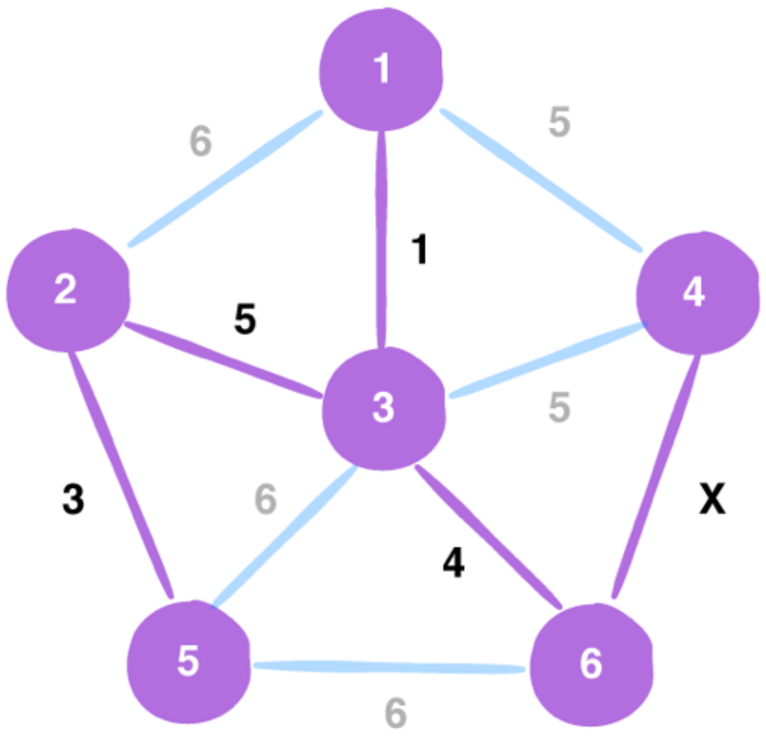
1. An adjacency list graph to build a minimum spanning tree. Adding vertices and edges to an adjacency list is $O(1)$.
2. A `Set` to store all vertices you have visited. Adding a vertex to the set and checking if the set contains a vertex also have a time complexity of $O(1)$.
3. A min-priority queue to store edges as you explore more vertices. The priority queue is built on top of a heap and insertion takes $O(\log E)$.

The worst-case time complexity of Prim's algorithm is O(E log E). This is because, each time you dequeue the smallest edge from the priority queue, you have to traverse all the edges of the `destination` vertex (O(E)) and insert the edge into the priority queue (O(logE)).

Challenges

Challenge 1: Discover the edge weight

Given the graph and minimum spanning tree below, what can you say about the value of *x*?

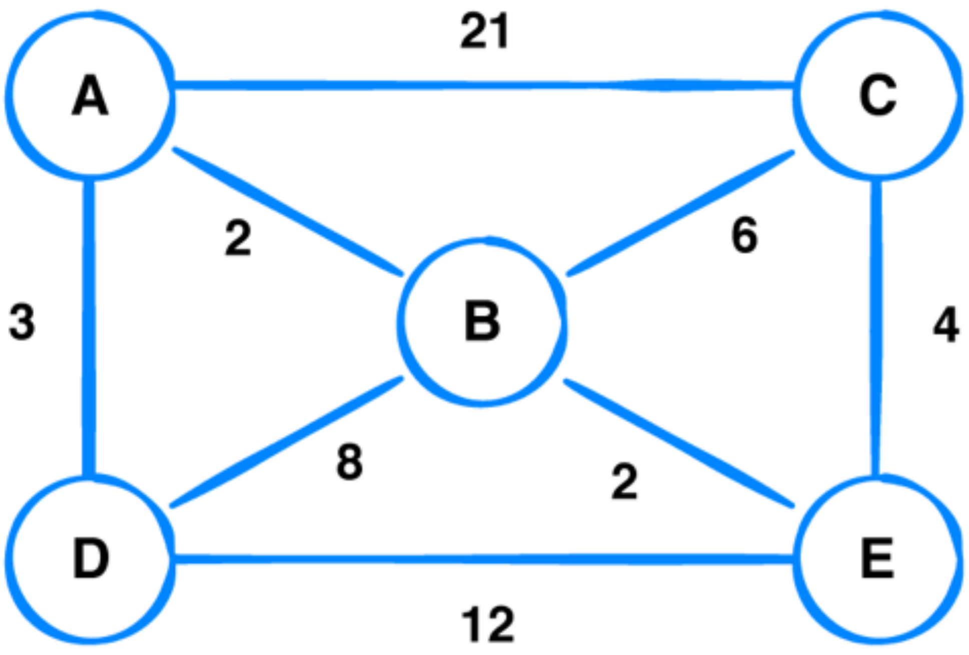


Solution 1

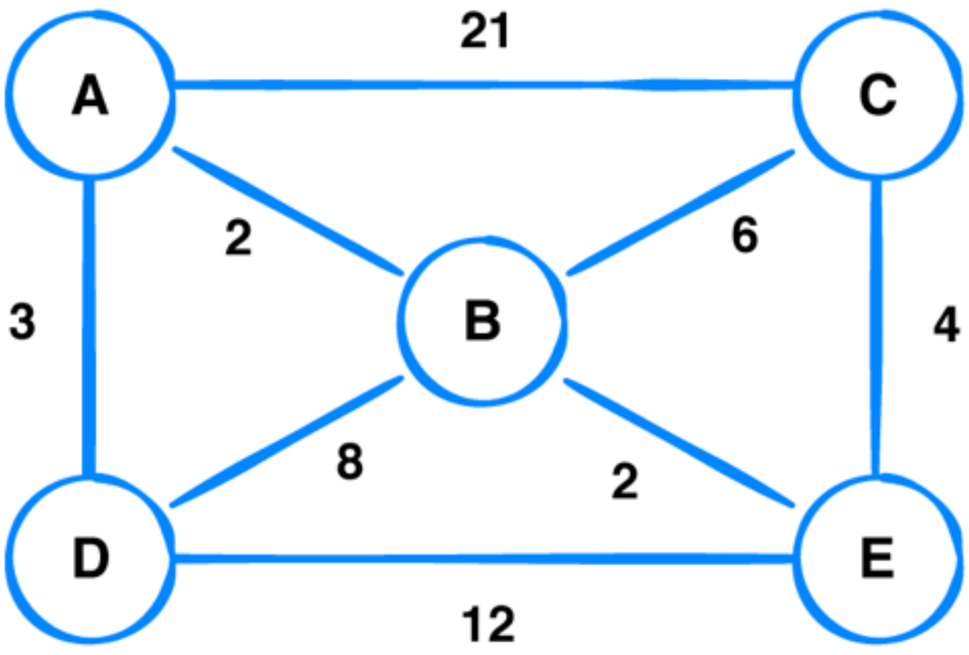
The value of x is no more than **5**.

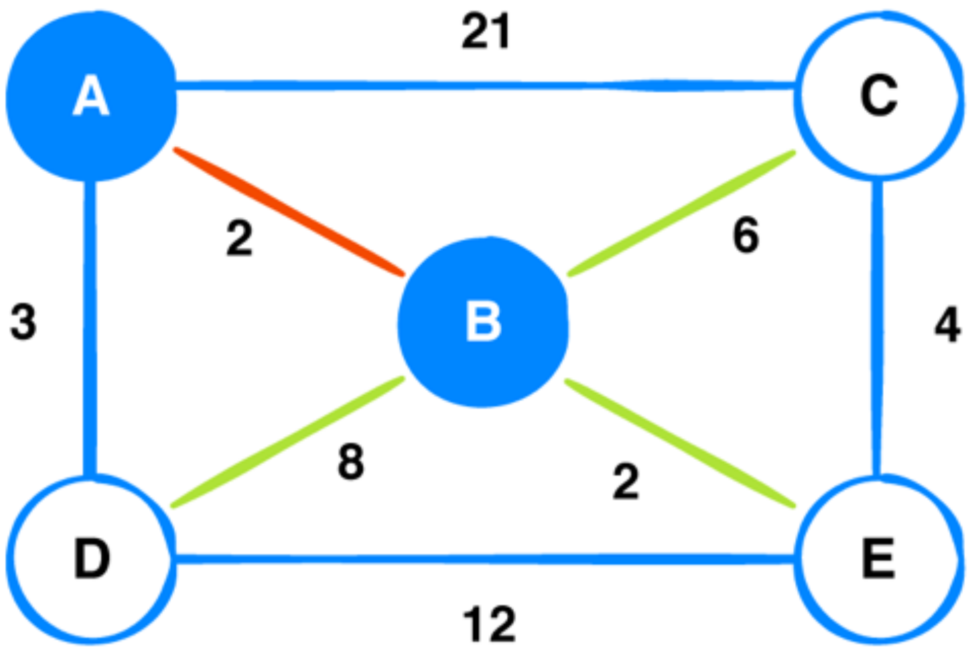
Challenge 2: One step at the time

Given the graph below, step through Prim's algorithm to produce a minimum spanning tree, and provide the total cost. Start at vertex **B**. If two edges share the same weight, prioritize them alphabetically.

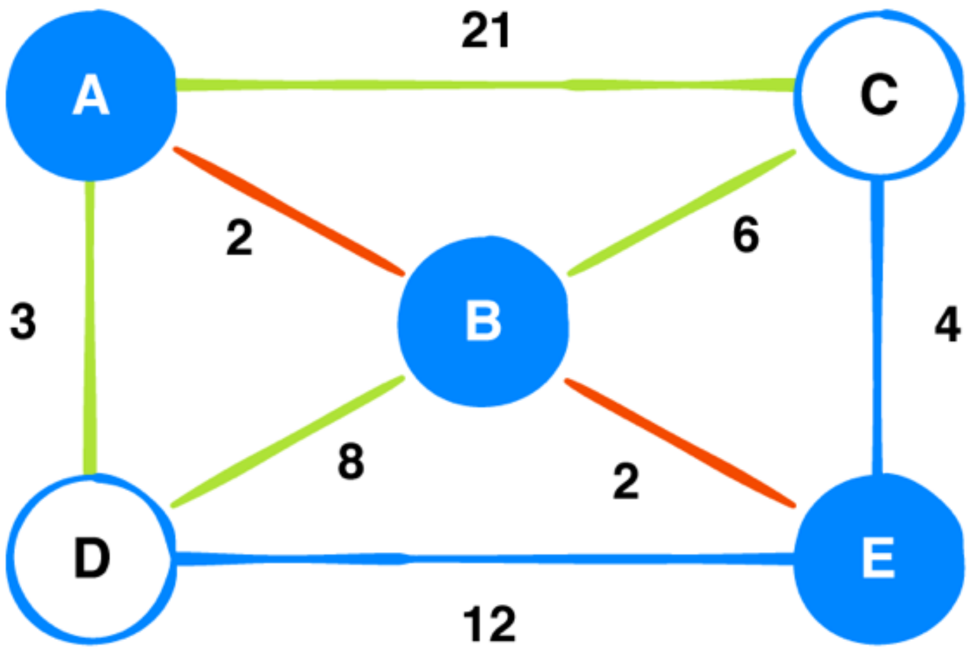


Solution 2

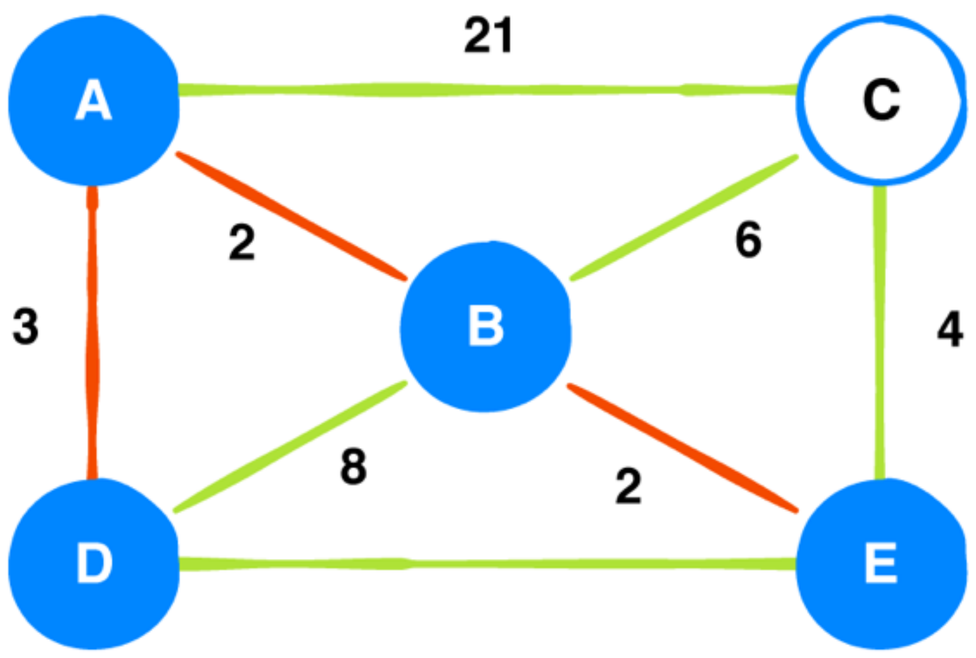




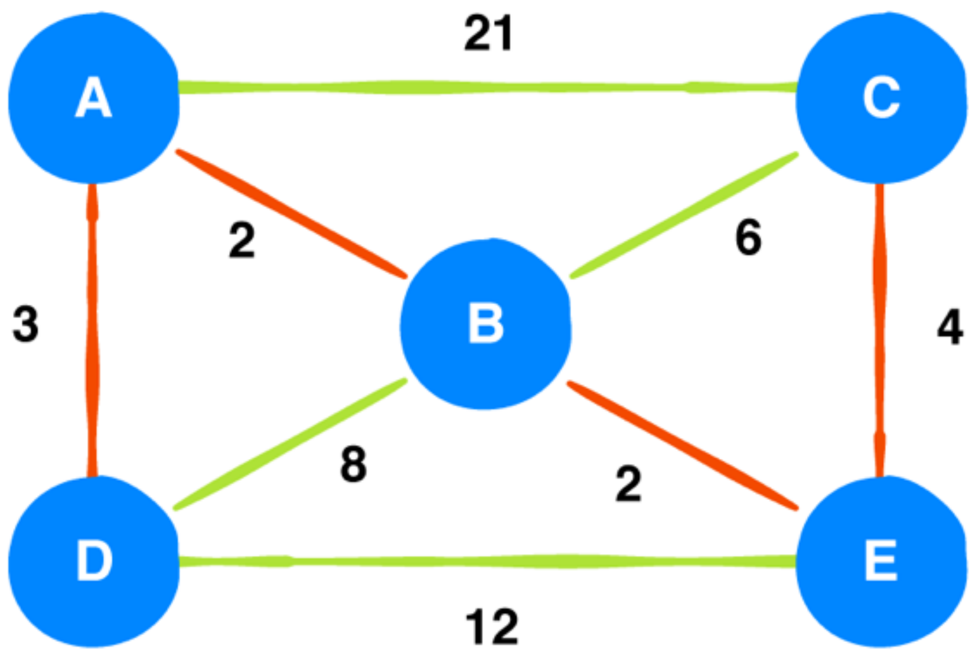
```
Edges [A:2, D:8, C:6, E:2]
Edges part of MST: [A:2]
Explored [A, B]
```



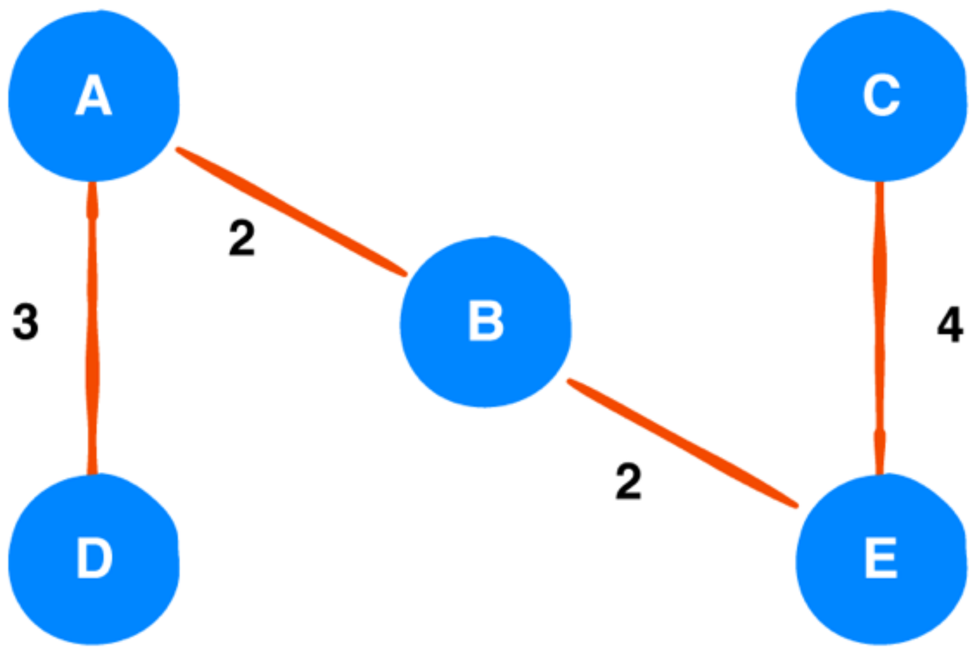
```
Edges [D:8, C:6, E:2, D:3, C:21]
Edges part of MST: [A:2, E:2]
Explored [A, B, E]
```



```
Edges [D:8, C:6, D:3, C:21, D:12, C:4]
Edges part of MST: [A:2, E:2, D:3]
Explored [A, B, E, D]
```



```
Edges [C:6, C:21, C:4]
Edges part of MST: [A:2, E:2, D:3, C:4]
Explored [A, B, E, D, C]
```



```
Edges [A:2, E:2, D:3, C:4]
Explored [A, B, E, D, C]
Total Cost: 11
```

Key points

- You can leverage three different data structures: Priority queue, set, and adjacency lists to construct Prim’s algorithm.
- Prim’s algorithm is a greedy algorithm that constructs a **minimum spanning tree**.
- A spanning tree is a subgraph of an undirected graph that contains all the vertices with the fewest number of edges.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.