Min span tree & shortest single path

Spanning Tree

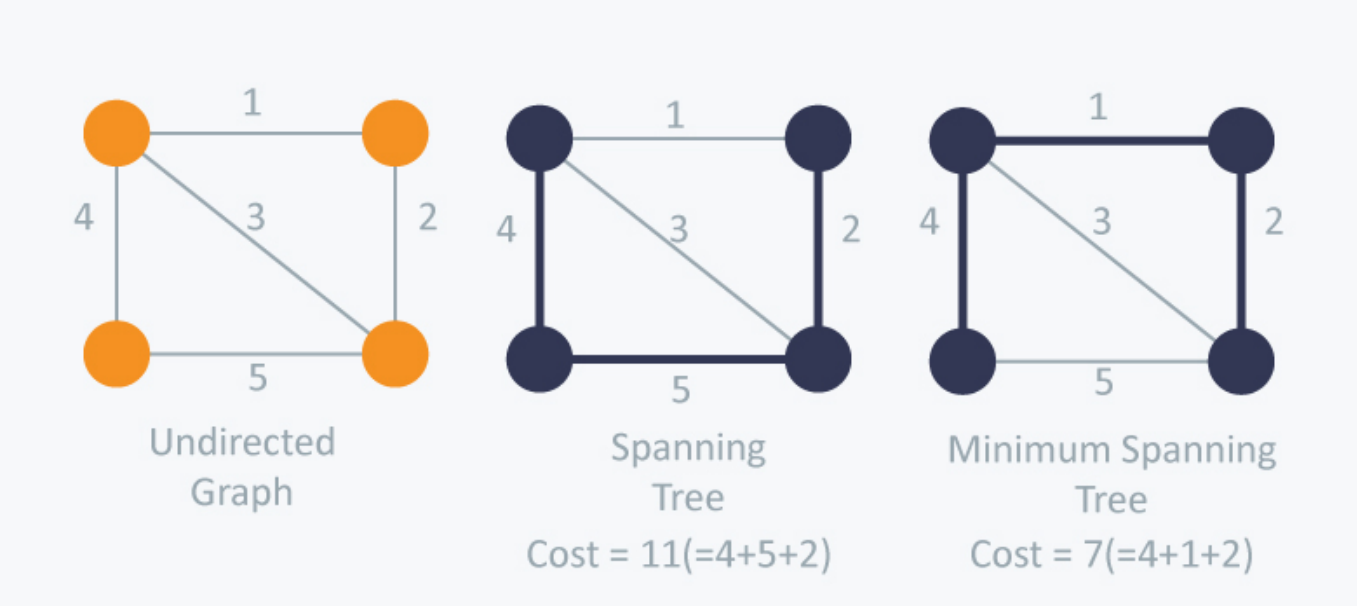
Given an undirected and connected graph $G = (V, E)$. A **spanning tree** of graph G is a tree that spans G.

- includes every vertex of G
- is a subgraph of G.
  - Meaning Every edge in the tree belongs to G

Minimum Spanning Tree

The cost of a spanning tree is the sum of the weight all edges. The minimum spanning tree is the tree where the cost of the tree is the lowest between all spanning trees.

This isn't only one tree, depending on the graph there can be many minimum spanning trees with equal sum.

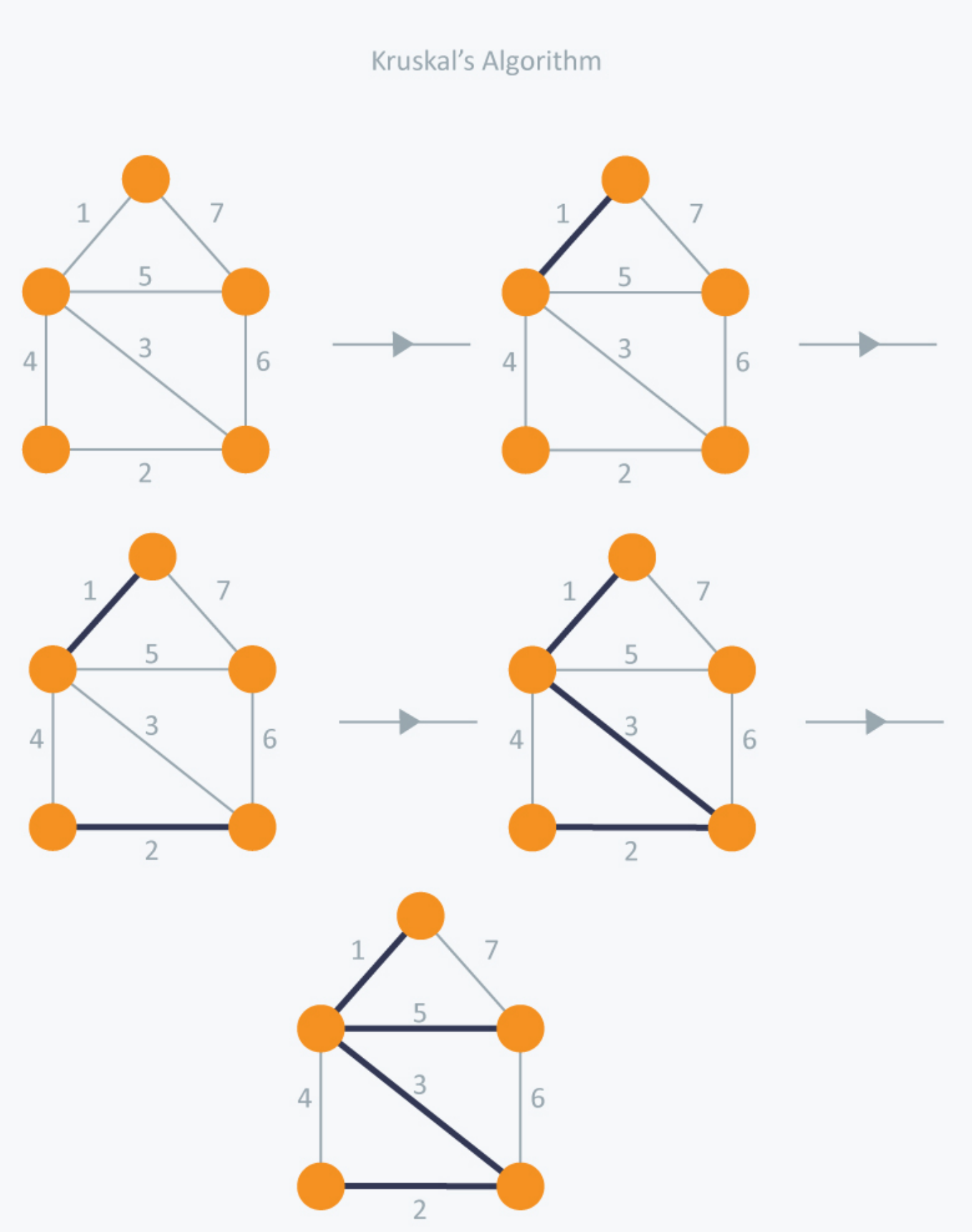


Kruskal’s Algorithm

Kruskal's is a greedy algorithm that builds a MST by adding edges one by one into a growing span tree. Each iteration it tries to find the edge with the least weight.

One issue we run into during Kruskal's is seeing if 2 vertexes are connected or not. DFS would take too much time.
The best solution is **Disjoint sets** whose intersetion is an empty set which will mean they have no element in common.
Another name for this is a **union-find** data structure.

Time complexity = $O(ElogV)$



Psuedocode:

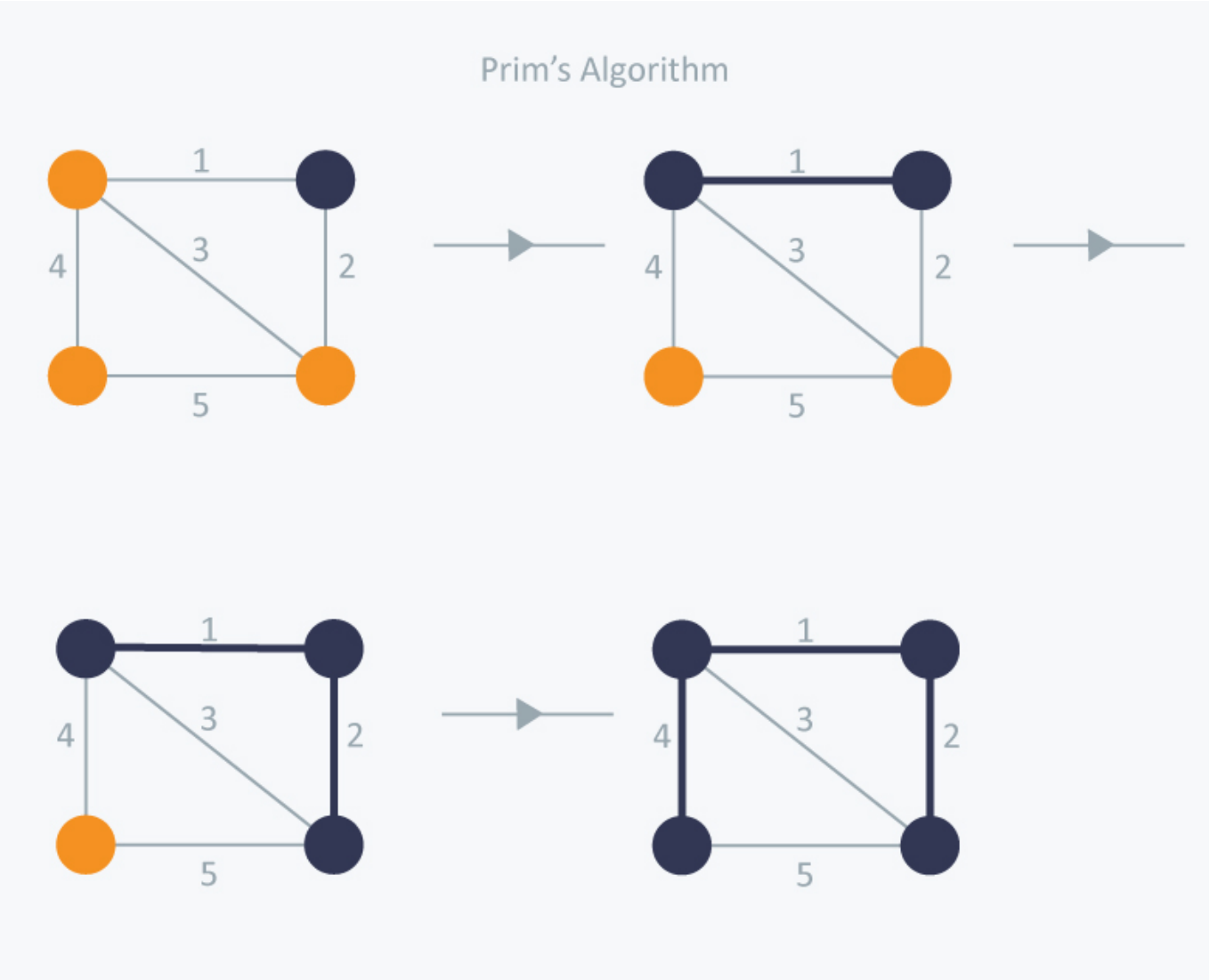
```
def kruskal(V, E):
        sort edges E by increasing weight
        F = ∅
        for each vertex v ∈ V:
                makeSet(v)
        for each edge (u, v) ∈ E:
                if findSet(u) ≠ findSet(v):
                        union(u, v)
                        add uv to F

        F is the MST
```

- `makeSet(v)` : makes a set out of a node
- `findSet(v)` : finds the set that the vertex belongs to
- `union(u, v)` : unions the set of u and v

Prim's Algorithm

Prim's algorithm is a greedy algorithm similar to Kruskal, the difference is instead of adding vertex based on edge size we have a starting algorithm and start there. Then adds the non-connected least edge.

Time complexity = $O(ElogV)$



Psuedocode:

```
def prim(V, E):
        tree = ø
        visited = { 1 }

        # We connected the least weighted edge and then add it to visited
        while visited != V:
                (u, v) = lowest cost edge that u ∈ U and v ∈ V - U
                tree = tree add {(u, v)}
                visited = visited add {v}
```

Single Source Shortest Path (SSSP)

We want to find the shortest path from a single vertex to all paths. Because all paths are unique we can create a **shortest path tree** (SPT) that contains all of the paths we desire.

Differences:

Minimum Spanning Tree	Shortest Path Tree
undirected	directed
Does not have a root	Source node as root
Can be unique	**Different for every root**

Negative Note:
Keep in mind negative edges. For the majority of SSSP algorithms negative edges can cause issues.

Bellman Ford Algorithm

Bellman Ford is the algorithm referred to as "The Only SSSP Algorithm" and helps us find the shortest path to a source to all other vertexes.

Bellman Ford is similar to Djikstra's algorithm with the only difference being it can handle negative edges.

High level overview:

1. The algorithm first works by overestimating all the paths to all vertices
2. Iteratively it relaxes and replaces the original values with more accurate values
   Reference: Programiz

Each node v in the graph stores two values which describe the distance from the source s to v.

- dist(v): is the length of the shortest path from the source. ∞ if no path exists
- pred(v): is the predecessor of v in the shortest path. NULL if no path exists.
  - Can also be named previous

Time complexity = $O(V * E)$

Psuedocode:

```
def bellmanFord(graph, source):
        distance[] = [infinite] * number of nodes
        previous[] = [NULL] * number of nodes

        distance[source] = 0

        for vertex in graph:
                for edge in graph[vertex]
                        tempDist = distance[edge] + weight(edge)
                        if tempDist < distance[vertex]:
                                distance[vertex] = tempDist
                                previous[vertex] = u
```

```
        for edge in graph:
                if distance[edge] + weight(edge) < distance[v]:
                        # Negative cycle exists

        return distance, previous
```

Dijkstra's Algorithm

The man, the myth, the legend Dijkstra. The algorithm everyone has heard of.

Dijkstra's algorithm is a more efficient version of Bellman Ford's Algorithm with a time complexity of $O(ELogV)$ vs $O(V * E)$ of Bellman Ford.

One limitiation of the algorithm is Dijkstra's cannot handle negative edges while Bellman Ford can.

--

Dijkstra's follows the principle that any subpath $B \rightarrow D$ of the shortest path $A \rightarrow D$ is valid. Meaning the shortest path between A & D is also the shortest path between B & D.

--
High level overview:

1. Go to each node and check paths
   - If a path is shorter we take that one instead
2. We go to the next unvisited nodes and check their lengths
   Reference: Programiz

Psuedocode:

```
def dijkstra(graph, source):
        distance[] = [infinite] * number of nodes
        previous[] = [NULL] * number of nodes
        prioQueue = []

        # Add all nodes not source into priority queue
        if vertex != source:
                prioQueue.add(vertex)

        distance[source] = 0

        while not prioQueue.empty():
                vertex = prioQueue.get()
                for **unvisited** neighbor in graph[vertex]:
                        tempDist = distance[vertex] + weight(vertex, neighbor)
                        if tempDist < distance[vertex]:
                                distance[vertex] = tempDist
```

```
                                previous[vertex] = neighbor

        return distance, previous
```