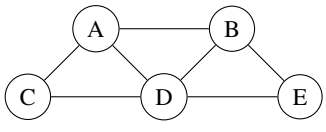
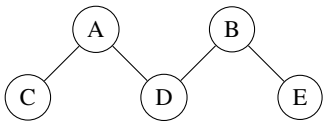
Chapter 20
Minimum Spanning Trees

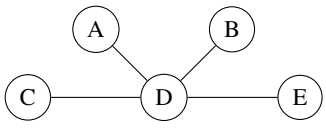
20.1 Introduction to Minimum Spanning Trees

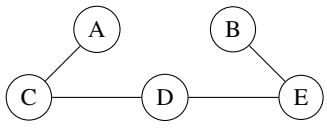
A **spanning tree** of a graph $G = (V, E)$ is a subset of edges E' that connects all vertices V in G with no cycles. This also implies that a spanning tree of G connects all vertices in G using the *fewest number of edges*. For example, consider the following graph:



The following are all valid spanning trees of the above graph. Notice that a graph can have more than one valid spanning tree, as long as all its vertices are connected with no cycles (as shown below).



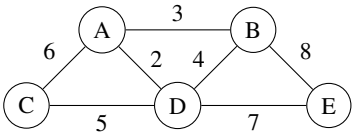




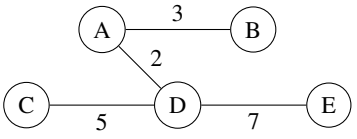
Because spanning trees are connected and acyclic, any spanning tree of a graph with $|V|$ vertices must have $|V| - 1$ edges. If there are fewer than $|V| - 1$ edges, the graph would not be fully connected. If there are more than $|V| - 1$ edges, the graph would have a cycle, since every additional edge beyond $|V| - 1$ would allow a vertex to be visited along more than one unique path. Try it yourself: add an additional fifth edge to any of the three spanning trees above — no matter where you add a connection, you will always end up creating a cycle!

In this chapter, we will focus on a special type of spanning tree, known as a **minimum spanning tree (MST)**. Given an edge-weighted, undirected graph $G = (V, E)$, the minimum spanning tree of G is the spanning tree of G with *the lowest total edge weight*. Since MSTs are a special category of spanning trees, the MST of any graph will always be connected and acyclic.

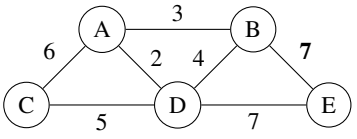
As an example, consider the following weighted, undirected graph:

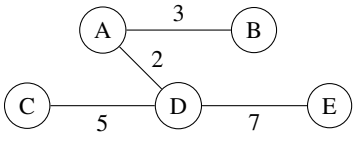


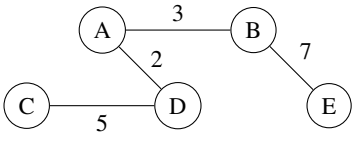
A spanning tree that minimizes total edge weight is shown below, with a total weight of $2 + 3 + 5 + 7 = 17$. Since there is no way to connect all the vertices in the graph with a total edge cost less than 17, this is a valid MST of the previous graph.



Note that it is possible for a graph to have multiple valid MSTs if more than one unique spanning tree minimizes total edge weight. For instance, if we adjusted the weight of edge $\overline{BE}$ to 7, there would be two valid MSTs, which are shown below. This is because there are two different edges we can include to connect vertex E to our MST, both of which share the same edge weight.







Example 20.1 Prove or disprove that a *unique* shortest edge in a graph must be included in its MST.

To show that a unique shortest edge will always be included in a MST, we can use proof by contradiction. Suppose that the unique shortest edge is *not* included in the MST. If this were the case, we would be able to add this unique shortest edge to the MST to create a cycle. Then, if we were to remove some other edge from this cycle, our new spanning tree would have a lower weight than before. This results in a contradiction: because we were able to create a spanning tree with a lower weight, our initial spanning tree without the unique shortest edge must not have been a valid MST!

Minimum spanning trees have significant practical applications in many computer science problems. For example, a MST can be used to efficiently connect different data centers using heavy duty fiber-optic cables (which can be quite expensive). To find out which pairs of data centers you should connect so that all data centers are reachable from each other using the lowest cost possible, you can represent these data centers as a graph and calculate its MST. Spanning trees also show up in the field of networking: Ethernet networking systems, for instance, rely on spanning trees to ensure that data in a network do not get stuck in cycles, which wastes network resources and consumes bandwidth at the expense of other network traffic (known as a *broadcast storm*). Lastly, as you will see in chapter 22, minimum spanning trees can be used to approximate solutions to complex problems such as the *traveling salesperson problem (TSP)*, which, given a graph of cities and distances between pairs of cities, seeks to find the shortest route that visits each city exactly once and starts and ends at the same origin city.

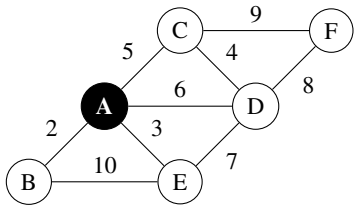
Because of how useful minimum spanning trees are in computer science, it would be remiss of us not to explore MSTs and MST algorithms in this course. In this chapter, we will focus on two important algorithms that can be used to find a minimum spanning tree of a weighted, undirected graph: *Prim's algorithm* and *Kruskal's algorithm*.

20.2 Prim's Algorithm

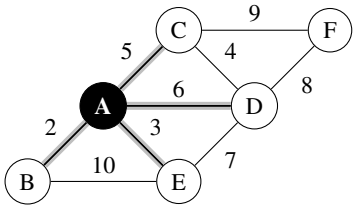
Prim's algorithm is an algorithm that can be used to find a MST of a weighted, connected, and undirected graph. This algorithm works by growing a tree from a single vertex, greedily selecting vertices to add to the tree based on the edge weights of the graph. The steps of Prim's algorithm are as follows:

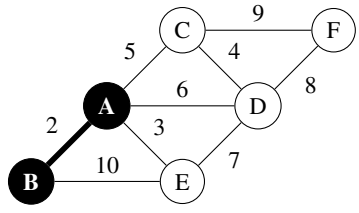
1. A tree is initialized starting from a single vertex that is arbitrarily chosen from the graph. The vertex chosen does not matter, as Prim's will always return a valid MST regardless of which vertex is used as a starting point. However, if there are multiple MSTs that exist within a graph, the choice of starting vertex could affect the MST found by the algorithm.
2. Then, of the edges that connect the tree to vertices not yet in the tree, the algorithm finds the edge with the lowest weight and adds it to the tree. This ends up growing the tree by connecting a new vertex that was previously not in the tree.
3. Step 2 is then repeated until all the vertices are in the tree. After the algorithm completes, the final tree is a valid minimum spanning tree of the entire graph.

As an example, consider the graph below. We will run Prim's algorithm on this graph, using vertex A as our starting vertex. To make explaining a bit easier, we will use the term *innie* to describe a vertex that is within the partial MST at any point during the algorithm, and the term *outie* to describe a vertex that has not yet been added to our partial MST. In the figures below, innies are colored black and outies are colored white.

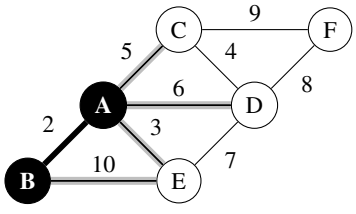


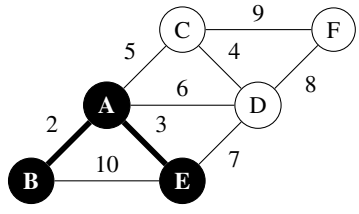
At every step of Prim's algorithm, we find the outie with the *smallest distance to any innie* and add it to the partial MST. Of the edges connected to the partial MST (shaded using a gray border), edge $\overline{AB}$ has the lowest weight, so we will add vertex B to our partial MST using edge $\overline{AB}$. This converts vertex B from an outie to an innie.



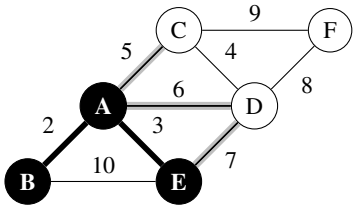


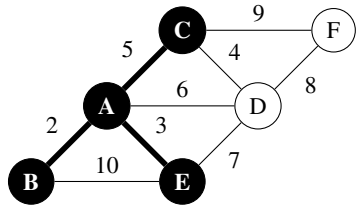
There are still outies remaining, so we continue this procedure. Again, we will look for the outie with the smallest distance to any innie and add it to our partial MST. In this case, this is vertex E, which can be connected using a weight of 3 via edge $\overline{AE}$.



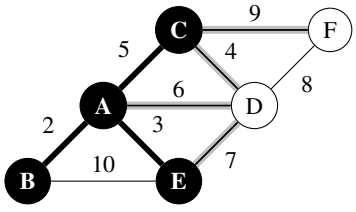


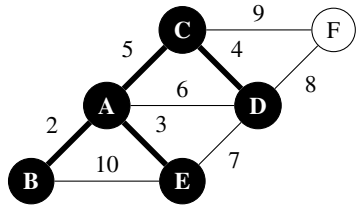
Next, the outie with the smallest distance to any innie is vertex C, which can be connected using a weight of 5 via edge $\overline{AC}$.



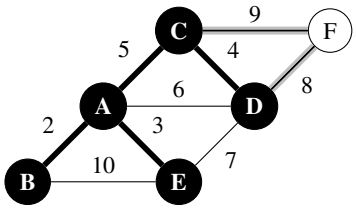


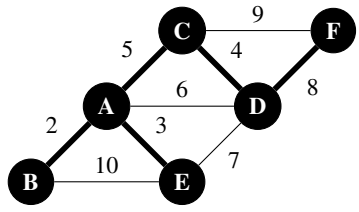
The outie with the smallest distance to any innie is now vertex D, which can be connected using a weight of 4 via edge $\overline{CD}$.



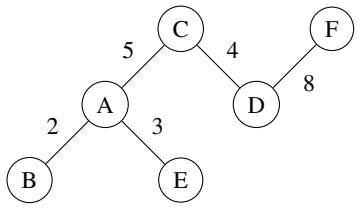


The outie with the smallest distance to any innie is now vertex F, which can be connected using a weight of 8 via edge $\overline{DF}$.





There are no more outies left, so every vertex has been added to our tree. Prim's algorithm is now complete, and this is our MST.



How can we turn this algorithm into code? In terms of data structures, each vertex v will need to keep track of three things, which we will denote as k_v, d_v and p_v:

- k_v: whether vertex v has been added to the MST (i.e., whether it is an innie).
- d_v: the minimum edge weight that connects vertex v to the partial MST ($d_v = \infty$ if there is no edge that directly connects vertex v to the partial MST yet).
- p_v: the vertex that connects vertex v to the MST, also known as the parent of v (initially unknown for all v).

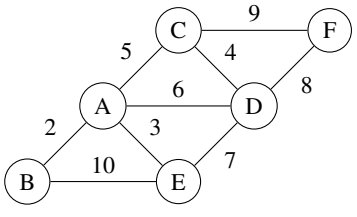
One way to store this information is to use a vector of objects that track these three values for each vertex v.[1] An outline of this is shown in the code below. Here, each `PrimData` object in the `prim_table` vector stores the values of k_v, d_v, and p_v for each vertex v in the graph (i.e., vertex 0's data is stored in `prim_table[0]`, vertex 1's data is stored in `prim_table[1]`, etc.).

```
1   struct PrimData {
2     double d;       // lowest distance to MST
3     int32_t p;      // index of vertex's parent in MST
4     bool k;         // whether or not vertex already in MST
5
6     PrimData()
7       : d{ std::numeric_limits<double>::infinity() }, p{ -1 }, k{ false } {}
8   };
9
10  std::vector<PrimData> prim_table;
```

After this data structure is set up, Prim's algorithm can be implemented using the following procedure:

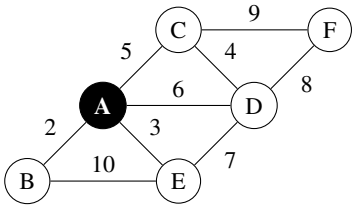
1. Set the d value of the starting vertex to 0.
2. From the set of vertices for which k_v is `false`, select the vertex v that has the smallest value of d_v.
3. Set the value of k_v for this vertex to `true`.
4. For each vertex w adjacent to v for which k_w is `false`, check whether d_w is greater than the edge weight that connects v and w. If it is, set d_w to the weight of the edge that connects v and w, and set p_w to vertex v.
5. Repeat steps 2-4 until k_v is true for every vertex (i.e., every vertex has been added to the tree).

Let's go through this process using the graph from above. Here, the data in the `prim_table` vector is represented using a table, where each row holds the values of k_v, d_v, and p_v for each vertex v. Since we will select vertex A as our starting vertex, the value of d_A is initially set to 0.



v	k_v	d_v	p_v
A	F	0	-
B	F	∞	-
C	F	∞	-
D	F	∞	-
E	F	∞	-
F	F	∞	-

Now, we will begin our loop. We first look through all the vertices for which k_v is false and find the vertex with the smallest value of d_v; this ends up being vertex A. Thus, we will add vertex A to our tree and set k_A to `true`.

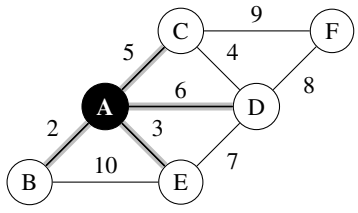


v	k_v	d_v	p_v
A	**T**	0	-
B	F	∞	-
C	F	∞	-
D	F	∞	-
E	F	∞	-
F	F	∞	-

[1] It is preferable to use a single vector of objects than to use three separate vectors for each of k_v, d_v, and p_v. Not only is this cleaner and more maintainable, it is also faster since you do not need to perform multiple vector accesses to retrieve the data of a single vertex.

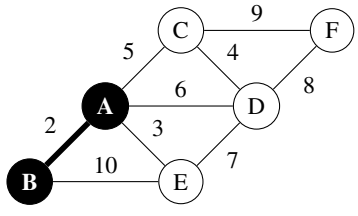
Then, we iterate over the neighbors of vertex A whose values of k are `false` (i.e., neighbors that are currently outies). For each neighbor w, we compare d_w with the weight of the edge connecting w with vertex A. If the edge weight is lower, we set d_w to this value and update p_w to vertex A. In this graph:

- $\overline{AB}$ (2) < d_B (∞), so set d_B to 2 and p_B to A.
- $\overline{AC}$ (5) < d_C (∞), so set d_C to 5 and p_C to A.
- $\overline{AD}$ (6) < d_D (∞), so set d_D to 6 and p_D to A.
- $\overline{AE}$ (3) < d_E (∞), so set d_E to 3 and p_E to A.



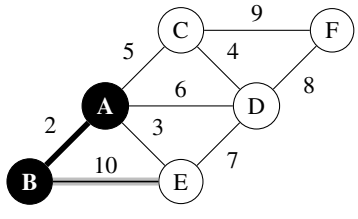
v	k_v	d_v	p_v
A	T	0	-
B	F	**2**	**A**
C	F	**5**	**A**
D	F	**6**	**A**
E	F	**3**	**A**
F	F	∞	-

We will repeat this process until all the vertices are added to our MST. Again, we will select the vertex v with the smallest d_v among the vertices whose k_v is `false`. In this case, vertex B has the smallest value of d_v among the unadded vertices, so we add B to our tree and set k_B to `true`.



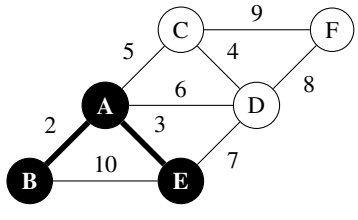
v	k_v	d_v	p_v
A	T	0	-
B	**T**	2	A
C	F	5	A
D	F	6	A
E	F	3	A
F	F	∞	-

We then iterate over the neighbors of vertex B whose values of k are `false` and update their distance d in the table if the edge connecting them to vertex B has a lower weight. The only neighbor of B whose k is `false` is vertex E, but the weight of edge $\overline{BE}$ is worse than the current distance of $d_E = 3$ via edge $\overline{AE}$. Thus, nothing is updated at this step.



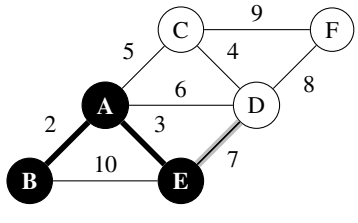
v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	F	5	A
D	F	6	A
E	F	3	A
F	F	∞	-

The next vertex we add to the tree is vertex E, whose d value of 3 is smallest among the vertices whose k values are `false`.



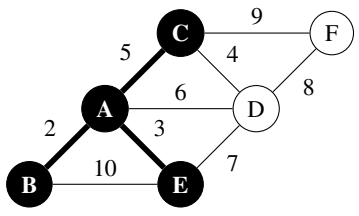
v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	F	5	A
D	F	6	A
E	**T**	3	A
F	F	∞	-

We then iterate over the neighbors of vertex E whose values of k are `false` and update their distance d in the table if the edge connecting them to vertex E has a lower weight. The only neighbor of E whose k is `false` is vertex D, but the weight of edge $\overline{ED}$ is worse than the current distance of $d_D = 6$ via edge $\overline{AD}$. Thus, nothing is updated at this step.



v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	F	5	A
D	F	6	A
E	T	3	A
F	F	∞	-

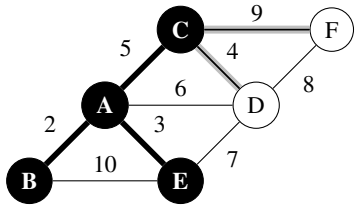
The next vertex we add to the tree is vertex C, whose d value of 5 is smallest among the vertices whose k values are `false`.



v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	**T**	5	A
D	F	6	A
E	T	3	A
F	F	∞	-

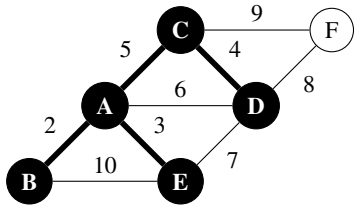
We then iterate over the neighbors of vertex C whose values of k are `false` and update their distance d in the table if the edge connecting them to vertex C has a lower weight. In this graph:

- $\overline{CD}$ (4) < d_D (6), so set d_D to 4 and p_D to C.
- $\overline{CF}$ (9) < d_F (∞), so set d_F to 9 and p_C to C.



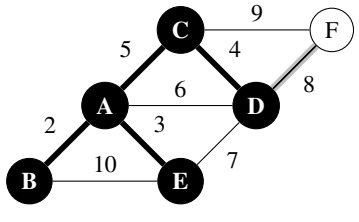
v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	T	5	A
D	F	**4**	**C**
E	T	3	A
F	F	**9**	**C**

The next vertex we add is vertex D, whose d value of 4 is the smallest among the vertices whose k values are `false`.



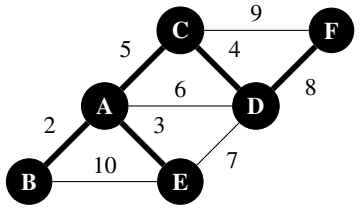
v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	T	5	A
D	**T**	4	C
E	T	3	A
F	F	9	C

We then iterate over the neighbors of vertex D whose values of k are `false` and update their distance d in the table if the edge connecting them to vertex D has a lower weight. The only neighbor of D whose k is `false` is vertex F. The weight of edge $\overline{DF}$, or 8, is better than the current weight of $d_F = 9$, so we update d_F to 8 and set p_F to D.



v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	T	5	A
D	T	4	C
E	T	3	A
F	F	**8**	**D**

The next vertex we add is vertex F, whose d value of 8 is the smallest among the vertices whose k values are `false`.



v	k_v	d_v	p_v
A	T	0	-
B	T	2	A
C	T	5	A
D	T	4	C
E	T	3	A
F	**T**	8	D

All of the vertices have been added, so Prim's algorithm completes, and we have our final MST (comprised of the bolded edges above). The total weight of the MST is the sum of all the values in the d_v column (in this case, $2 + 5 + 4 + 3 + 8 = 22$). To determine which edges exist in the MST, simply look at the parents of each of the vertices in the table. Here, the parent of B is A, which means that the edge $\overline{AB}$ in the MST (the same applies for the remaining entries in the table).

The time complexity of Prim's algorithm depends on the data structure that is used to represent the graph, as well as the process by which the outie with the smallest d_v is found. For instance, consider an implementation of Prim's algorithm that conducts a *linear search* of all the vertices to find the outie with the smallest d_v. The pseudocode for this implementation is shown below:

```
1  Algorithm PrimLinearSearch():
2    initialize Prim table (which stores k, d, and p for each vertex v)
3    set d of starting vertex to 0
4    for each vertex v in graph:
5      find vertex v_min with smallest d value among vertices whose k is false
6      set k of vertex v_min to true
7      for each neighbor n of vertex v_min in graph:
8        if k of vertex n is false and edge weight between n and v_min < d of n:
9          set d of vertex n to weight of edge between n and v_min
10         set p of vertex n to index of v_min
```

Since a linear search is completed to find `v_min`, or the outie with the smallest d_v, the time complexity of completing line 5 of the pseudocode above is $\Theta(|V|)$, where $|V|$ is the number of vertices in the graph. The time complexity of performing the loop on line 7 is also linear on the number of vertices in the graph. Since all of this is nested in a `for` loop on line 4 that runs $|V|$ times, the overall time complexity of Prim's algorithm using linear search is worst-case $\Theta(|V|^2)$.

Another common implementation of Prim's algorithm uses a *min-heap* to store the vertices of the graph, where the priority of a vertex v is determined by d_v, or the smallest edge weight that connects vertex v to the partial MST. This removes the need to conduct a linear search to find the vertex with the smallest d_v, as such a vertex can be obtained by popping values off the top of the heap. Every time a vertex v is popped out of the heap and added to the partial MST, we go through the outies that are directly connected to v — if an outie w has a value of d_w that is greater than the edge weight connecting it to v, the value of d_w is updated to this edge weight, which increases the priority of w in the min-heap. The code for this implementation is shown below:

```
1   struct PrimData {
2     double d;
3     int32_t p;
4     bool k;
5     PrimData()
6       : d{ std::numeric_limits<double>::infinity() }, p{ -1 }, k{ false } {}
7   };
8
9   using AdjList = std::vector<std::vector<std::pair<int32_t, int32_t>>>;
10  using DistPair = std::pair<int32_t, int32_t>;  // <d_v, v>
11
12  std::vector<PrimData> prim_heap(const AdjList& adj_list, int32_t start) {
13    std::vector<PrimData> prim_table(adj_list.size());
14    std::priority_queue<DistPair, std::vector<DistPair>, std::greater<DistPair>> pq;
15    prim_table[start].d = 0;
16    pq.emplace(prim_table[start].d, start);
17    while (!pq.empty()) {
18      int32_t parent = pq.top().second;
19      pq.pop();
20      prim_table[parent].k = true;
21      for (auto& neighbor_dist_pair : adj_list[parent]) {
22        auto [neighbor, dist] = neighbor_dist_pair;
23        if (!prim_table[neighbor].k && dist < prim_table[neighbor].d) {
24          prim_table[neighbor].d = dist;
25          prim_table[neighbor].p = parent;
26          pq.emplace(dist, neighbor);
27        } // if
28      } // for neighbor_dist_pair
29    } // while
30    return prim_table;
31  } // prim_heap()
```

In this code:

- Line 9 provides an alias for the adjacency list object, which is represented using a 2-D vector of pairs, where each pair stores a connection with its weight.
- Line 10 provides an alias for the object we will store in the priority queue, which is a pair that stores d_v with each vertex v.
- Lines 13-14 initialize the Prim table and priority queue, and lines 15-16 set the d value of the starting vertex to 0 and pushes it into the priority queue.
- Lines 17-29 perform the following loop, as long as the priority queue is not empty:
  - We take out the vertex at the top of the queue and set its k to `true` (lines 18-20). We use `.second` on line 18 because each vertex is stored as part of a pair, and `.first` stores its value of d (used to determine its priority).
  - We iterate over all the edges of this vertex in its adjacency list (line 21). Note: since the adjacency list stores *pairs* of vertices and weights, a structured binding was used to retrieve both components on one line (see section 11.13.1 for how they work). It's just a another way of writing:

    ```
    1   auto neighbor = neighbor_dist_pair.first;
    2   auto dist = neighbor_dist_pair.second;
    ```

– As long as a neighbor of the parent vertex has not been added to the partial MST (i.e., k is `false`), check and update its d value if necessary. If an update is made, the priority of the vertex should be updated in the priority queue as well. Since there is no way to directly update the priority of an element in a `std::priority_queue<>`, the above implementation pushes the vertex back into the priority queue so that its priority uses the updated d value (duplicates in the priority queue are okay, as any duplicates of lower priority are ignored if its vertex has already been seen; i.e., the first check on line 23). However, there exist heap implementations, such as the *Fibonacci heap*, that support direct priority modification in amortized constant time (see the remark below on what this specific type of heap, but you will not be required to know about it for the class).

What is the time complexity of the heap implementation of Prim's algorithm? If a binary heap is used as the underlying structure of the priority queue (which it is for a `std::priority_queue<>`), then the cost of finding the next vertex to add to the MST is $\Theta(\log(|V|))$ (from retrieving popping the vertex at the top of the heap). This is done a total of $|V|$ times, for a total complexity of $\Theta(|V|\log(|V|))$. However, this is not all: we also need to iterate over the neighbors of each vertex (line 21) and update their distance values if necessary. The cost of updating the priority of a vertex is $\Theta(\log(|V|))$, since we need to push the updated vertex into the priority queue (line 26). This work is done at most $\Theta(|E|)$ times, since it is executed within a loop that iterates over all edges in the graph. Thus, the overall cost of updating the distance of vertices in the algorithm is worst-case $\Theta(|E|\log(|V|))$. Putting both steps together, the worst-case time complexity of the heap implementation of Prim's algorithm is $\Theta((|V|+|E|)\log(|V|))$, or simply $\Theta(|E|\log(|V|))$. Note that we can make this simplification because the number of edges $|E|$ will always be larger than $|V|-1$ in a connected graph — thus, the contribution of $|E|$ toward the time complexity will be *at least as dominant* as the contribution of $|V|$.

Remark: The time complexity of Prim's algorithm actually depends on the type of heap that serves as the underlying structure of the priority queue. If you use a binary heap as the implementation (which the STL does), the worst-case time complexity of Prim's algorithm is $\Theta(|V|\log(|V|)+|E|\log(|V|)) = \Theta(|E|\log(|V|))$. However, there exists another type of heap known as a *Fibonacci heap*, which allows you to decrease the priority of an element in amortized $\Theta(1)$ time. This drops the worst-case time complexity of Prim's algorithm from $\Theta(|V|\log(|V|)+|E|\log(|V|))$ to $\Theta(|V|\log(|V|)+|E|)$, since the cost of updating a vertex's priority no longer takes $\Theta(\log(|V|))$ time.

Is the heap implementation of Prim's algorithm always more efficient than the linear search implementation? A complexity of $\Theta(|E|\log(|V|))$ might appear faster than the $\Theta(|V|^2)$ time required in a linear search, but this actually depends on the density of the underlying graph. If the heap implemention is used on a sparse graph, $|E| = O(|V|)$, so the worst-case time complexity becomes $\Theta(|E|\log(|V|)) = \Theta(|V|\log(|V|))$, which is indeed better than $\Theta(|V|^2)$. However, if the heap implementation were used on a dense graph, $|E| = \Theta(|V|^2)$, so the worst-case time complexity becomes $\Theta(|E|\log(|V|)) = \Theta(|V|^2\log(|V|))$, which is *worse* than $\Theta(|V|^2)$. Thus, if you want to run Prim's algorithm on a graph, check to see if it is dense or sparse first. If the graph is dense, the linear search implementation of Prim's algorithm is preferred; if the graph is sparse, the heap implementation of Prim's algorithm is preferred.

Summary of Prim's Algorithm Time Complexities

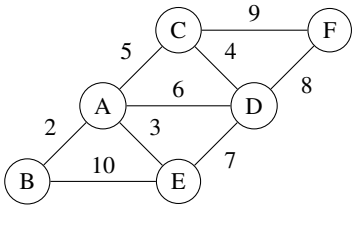
Implementation Method	Time Complexity
Adjacency matrix, linear search	$\Theta(\|V\|^2)$
Adjacency list, binary heap	$\Theta(\|E\|\log(\|V\|))$
Adjacency list, Fibonacci heap	$\Theta(\|E\|+\|V\|\log(\|V\|))$

20.3 Kruskal's Algorithm

Kruskal's algorithm is another algorithm that can be used to find the MST of a weighted, connected, and undirected graph. This algorithm greedily selects edges one at a time and adds it to a growing subgraph, which produces a "forest" of smaller, disjoint trees that are eventually merged into a single minimum spanning tree. Kruskal's algorithm can be summarized using the following two steps:

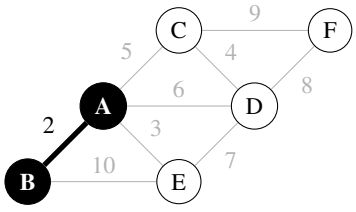
1. Sort the edges of the tree in order of increasing edge weight.
2. Iterate over all the edges in sorted order, and add each edge to the partial MST only if it does *not* produce a cycle. If the inclusion of an edge creates a cycle, discard that edge. This step is repeated until $|V|-1$ edges are added to the MST.

Because Kruskal's algorithm grows a partial MST by greedily adding edges in order of increasing edge weight, the initial edges of the partial MST may be disjoint. However, all of these disjoint components will eventually be linked together once Kruskal's algorithm runs to completion. As an example, consider the same graph that we used to demonstrate Prim's algorithm. When running Kruskal's algorithm, we first sort the edges of the graph in order of increasing edge weight:



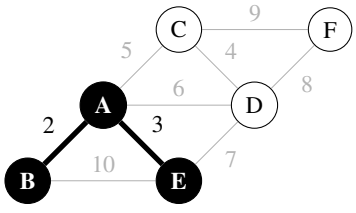
$\overline{AB}$ - 2
$\overline{AE}$ - 3
$\overline{CD}$ - 4
$\overline{AC}$ - 5
$\overline{AD}$ - 6
$\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

Then, we go through each of these edges and add them to the MST, as long as the edge does not result in a cycle. First, we will add edge $\overline{AB}$ to our MST, since it has the lowest weight.



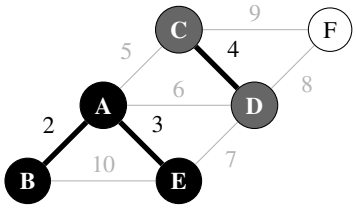
➔ $\overline{AB}$ - 2
$\overline{AE}$ - 3
$\overline{CD}$ - 4
$\overline{AC}$ - 5
$\overline{AD}$ - 6
$\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

The next edge we consider is edge $\overline{AE}$. The addition of $\overline{AE}$ does not produce a cycle, so we add it to the MST.



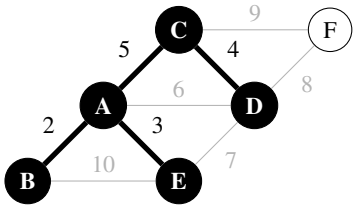
$\overline{AB}$ - 2
➔ $\overline{AE}$ - 3
$\overline{CD}$ - 4
$\overline{AC}$ - 5
$\overline{AD}$ - 6
$\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

The next edge we consider is edge $\overline{CD}$. The addition of $\overline{CD}$ does not produce a cycle, so we add it to the MST.



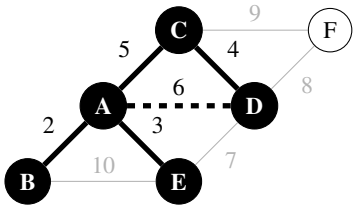
$\overline{AB}$ - 2
$\overline{AE}$ - 3
➔ $\overline{CD}$ - 4
$\overline{AC}$ - 5
$\overline{AD}$ - 6
$\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

The next edge we consider is edge $\overline{AC}$. The addition of $\overline{AC}$ does not produce a cycle, so we add it to the MST.



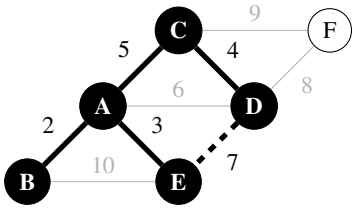
$\overline{AB}$ - 2
$\overline{AE}$ - 3
$\overline{CD}$ - 4
➔ $\overline{AC}$ - 5
$\overline{AD}$ - 6
$\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

The next edge we consider is edge $\overline{AD}$. The addition of edge $\overline{AD}$ would produce a cycle $A \to C \to D \to A$, so this edge is discarded and *not* added to the MST.



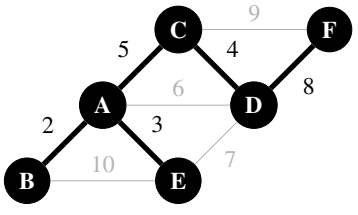
$\overline{AB}$ - 2
$\overline{AE}$ - 3
$\overline{CD}$ - 4
$\overline{AC}$ - 5
➔ $\overline{AD}$ - 6
$\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

The next edge we consider is edge $\overline{DE}$. The addition of edge $\overline{DE}$ would produce a cycle $A \to C \to D \to E \to A$, so this edge is also discarded and *not* added to the MST.



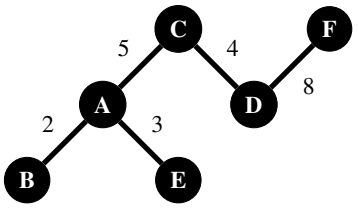
$\overline{AB}$ - 2
$\overline{AE}$ - 3
$\overline{CD}$ - 4
$\overline{AC}$ - 5
$\overline{AD}$ - 6
➔ $\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

The next edge we consider is edge $\overline{DF}$. The addition of $\overline{DF}$ does not produce a cycle, so we add it to the MST.



$\overline{AB}$ - 2
$\overline{AE}$ - 3
$\overline{CD}$ - 4
$\overline{AC}$ - 5
$\overline{AD}$ - 6
$\overline{DE}$ - 7
➔ $\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

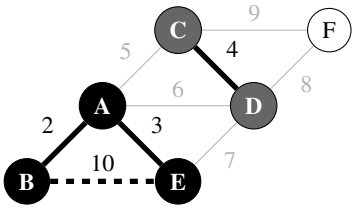
A total of $|V| - 1$ edges have been added to the MST, so Kruskal's algorithm is complete. Any additional edge that remains would produce a cycle, so edges $\overline{CF}$ and $\overline{BE}$ are both discarded. This gives us our completed MST.



$\overline{AB}$ - 2
$\overline{AE}$ - 3
$\overline{CD}$ - 4
$\overline{AC}$ - 5
$\overline{AD}$ - 6
$\overline{DE}$ - 7
$\overline{DF}$ - 8
$\overline{CF}$ - 9
$\overline{BE}$ - 10

To determine if adding an edge would create a cycle, Kruskal's algorithm relies on the union-find data structure. This is because an edge is only safe to add if it connects two vertices belonging to *different* disjoint sets. If two vertices are already part of the same disjoint set, there must exist a path from one to the other, and any new edge between the two vertices would create a cycle.

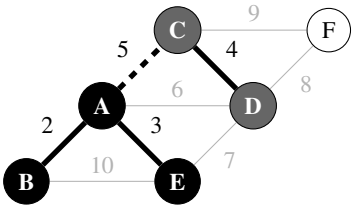
B and ***E*** part of same disjoint set, so $\overline{BE}$ creates a cycle



Element	A	B	C	D	E	F
Representative	A	A	C	C	A	F

```
find_set(B) == find_set(E)
```

A and ***C*** part of different disjoint sets, so $\overline{AC}$ safe to add



Element	A	B	C	D	E	F
Representative	A	A	C	C	A	F

```
find_set(A) != find_set(C)
```

An implementation of Kruskal's algorithm is shown in the code below (the function itself begins on line 19). This implementation returns a vector containing all the edges in the discovered MST.

```
1  struct EdgeData {
2    double weight;
3    std::pair<int32_t, int32_t> endpoints;
4    bool operator<(const EdgeData& rhs) {
5      return weight < rhs.weight;
6    }
7  };
8
9  class UnionFind {
10   // ... see chapter 13 for full implementation
11 public:
12   UnionFind(int32_t size);
13   int32_t find_set(int32_t x);
14   void union_set(int32_t x, int32_t y);
15 };
16
17 using AdjList = std::vector<std::vector<std::pair<int32_t, double>>>;
18
19 std::vector<EdgeData> kruskal(const AdjList& adj_list) {
20   std::vector<EdgeData> edge_list;
21   for (int32_t v = 0; v < adj_list.size(); ++v) {
22     for (auto& neighbor_dist_pair : adj_list[v]) {
23       auto [neighbor, dist] = neighbor_dist_pair;
24       if (v < neighbor) {
25         edge_list.push_back({dist, {v, neighbor}});
26       } // if
27     } // for neighbor_dist_pair
28   } // for v
29   std::sort(edge_list.begin(), edge_list.end());
30   UnionFind uf(adj_list.size());
31   std::vector<EdgeData> in_mst;
32   for (auto& edge : edge_list) {
33     double dist = edge.weight;
34     auto [v1, v2] = edge.endpoints;
35     int rep_v1 = uf.find_set(v1);
36     int rep_v2 = uf.find_set(v2);
37     if (rep_v1 != rep_v2) {
38       in_mst.push_back(edge);
39       if (in_mst.size() == adj_list.size() - 1) {
40         break;
41       } // if
42       uf.union_set(v1, v2);
43     } // if
44   } // for edge
45   return in_mst;
46 } //kruskal()
```

In this code:

- Lines 21-28 use the adjacency list to create a vector of edges that can be sorted based on edge weight. The check on line 24 is needed because the graph is undirected, and we don't want to add the same edge twice.
- The edges are then sorted by increasing edge weight on line 29.
- Line 30 initializes the union-find container, and line 31 initializes the result vector.
- Lines 32-44 use the union-find container to determine whether an edge should be added to the MST or not. An edge is only added if it connects two vertices in different disjoint sets (i.e., their ultimate representatives are unequal), and the algorithm exits once $|V|-1$ edges are added to the MST.

What is the time complexity of Kruskal's algorithm? If you analyze the steps of the algorithm, you will discover that the cost of sorting the edges is the bottleneck of the entire algorithm. Sorting takes $\Theta(|E|\log(|E|))$ time, while all of the other steps have a lower time complexity: the cost of looping over $|E|$ edges takes $\Theta(|E|)$ time, and the process of checking for a cycle can be treated as a constant time operation, assuming that the union-find data structure is efficiently implemented using path compression and either union-by-rank or union-by-size.[2] Thus, the sorting step is the dominant term in the complexity class, and the overall time complexity of Kruskal's algorithm is $\Theta(|E|\log(|E|))$.

Remark: You will often see the time complexity of Kruskal's algorithm expressed as $\Theta(|E|\log(|V|))$ instead of $\Theta(|E|\log(|E|))$. This doesn't mean that our previous analysis was incorrect! In fact, $\Theta(|E|\log(|E|))$ and $\Theta(|E|\log(|V|))$ are part of the same complexity class, since $|V|-1 \leq |E| < |V|^2$, which implies that $\log(|V|) \leq \log(|E|) < \log(|V|^2) = 2\log(|V|) = \Theta(\log(|V|))$. When working with graph complexities, it may make sense to include both $|E|$ and $|V|$, and denoting the complexity of Kruskal's algorithm as $\Theta(|E|\log(|V|))$ makes it easier to compare it with other algorithms, such as Prim's algorithm. That being said, the two complexities are interchangeable, and both can be used to describe the runtime of Kruskal's algorithm.

[2] More accurately, a single cycle check takes $\Theta(\alpha(|V|))$ time, where α represents the inverse Ackermann function. However, this function grows so slowly that its cost can essentially be treated as a constant (see chapter 13 for more details).

Example 20.2 Given the following adjacency matrix, what is the total weight of its MST?

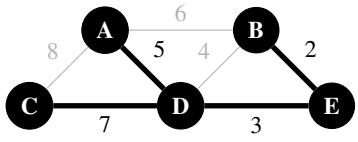
	A	B	C	D	E
A	-	6	8	5	-
B	6	-	-	4	2
C	8	-	-	7	-
D	5	4	7	-	3
E	-	2	-	3	-

With an adjacency matrix like this, you may be tempted to draw everything out. However, there is no need to do so! All of the information you need to calculate the weight of the MST is provided in the table — you just need to sort the edges in order of increasing weight, and then greedily select edges that do not produce a cycle. The edges in sorted order are:

$$\overline{BE} = 2,\ \overline{DE} = 3,\ \overline{BD} = 4,\ \overline{AD} = 5,\ \overline{AB} = 6,\ \overline{CD} = 7,\ \overline{BC} = 8$$

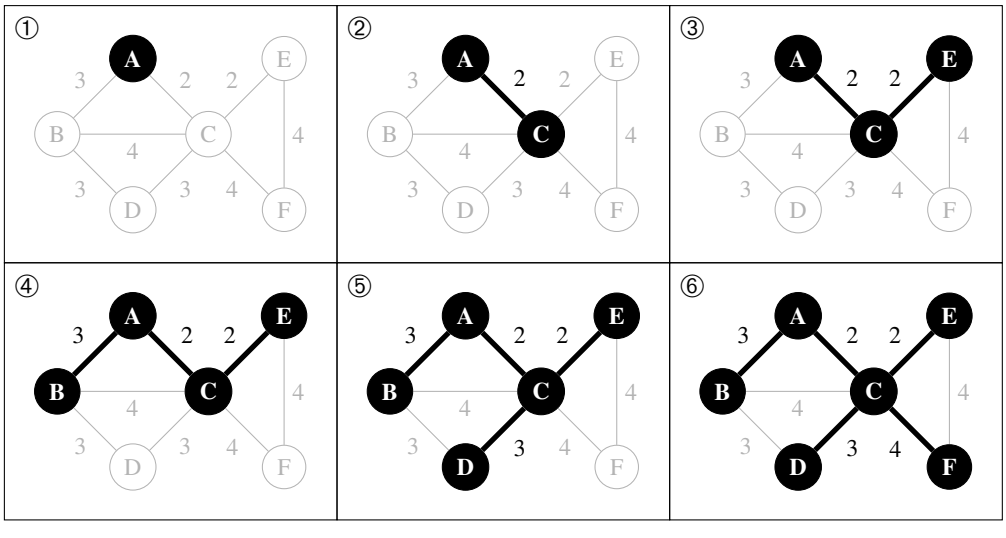
First, we add edges $\overline{BE}$ and $\overline{DE}$ to the MST, since they have the smallest weights and do not produce a cycle. Edge $\overline{BD}$ is omitted because vertices B and D are already part of the same disjoint set, which means the addition of $\overline{BD}$ would produce a cycle. Edge $\overline{AD}$ is added to the MST next, which adds A to the same disjoint set as vertices B, D, and E. Then, the next edge, edge $\overline{AB}$, is omitted. Lastly, edge $\overline{CD}$ is added to connect vertex C to the rest of the vertices in the MST. A total of $|V| - 1$ edges have now been added to the MST, so any remaining edges must not be included. The total weight of the MST is therefore the combined weights of edges $\overline{BE}$, $\overline{DE}$, $\overline{AD}$, and $\overline{CD}$, or $2+3+5+7 = 17$.

The tree, when drawn out, looks like this:

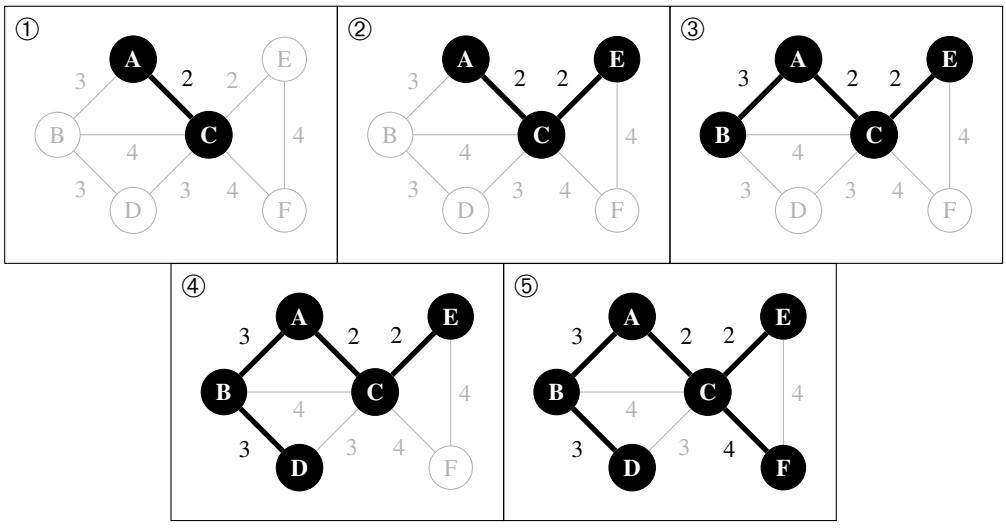


20.4 Comparing Prim's and Kruskal's Algorithms

In this chapter, we covered two algorithms that can be used to identify a minimum spanning tree for a weighted, connected, and undirected graph. *Prim's algorithm* grows a minimum spanning tree from an arbitrarily selected vertex by iteratively adding the outie that is closest to the current tree. The worst-case time complexity of Prim's algorithm is $\Theta(|V|^2)$ if a linear search is used to search for the closest vertex, and $\Theta(|E|\log(|V|))$ if a binary heap is used instead.



On the other hand, *Kruskal's algorithm* grows a tree by traversing the edges in order of increasing weight and iteratively adding the next cheapest edge, provided that the edge does not create a cycle. The time complexity of Kruskal's algorithm is $\Theta(|E|\log(|V|))$, as long as the sorting algorithm and union-find container are implemented efficiently.



Notice that the two MSTs generated by Prim's and Kruskal's algorithm are *different* for the graph above – this is entirely possible if a graph has multiple valid MSTs. It is also possible for an algorithm to return different MSTs for the same graph depending on how it is implemented. For instance, if you start Prim's algorithm on a different vertex, or if you modify the order in which Kruskal's algorithm handles duplicate weights, you could end up with an entirely new MST.

When should you use Prim's algorithm, and when should you use Kruskal's algorithm? If the underlying graph is dense, then the linear search implementation of Prim's algorithm is the best choice. This is because the number of edges $|E|$ in a dense graph is $\Theta(|V|^2)$, which balloons the worst-case time complexity of Kruskal's algorithm (and the heap implementation of Prim's algorithm) to $\Theta(|E|\log(|V|)) = \Theta(|V|^2\log(|V|))$. This is worse off than the $\Theta(|V|^2)$ time complexity of linear search Prim's.

However, if the underlying graph is sparse, the number of edges $|E|$ is $O(|V|)$. Therefore, the worst-case time complexity of Kruskal's algorithm (and the heap implementation of Prim's algorithm) becomes $\Theta(|V|\log(|V|))$, which is asymptotically faster than the $\Theta(|V|^2)$ time complexity of linear search Prim's. Thus, Kruskal's algorithm and the heap implementation of Prim's algorithm work best on sparse graphs. A comparison between the two algorithms is shown below:

Prim's Algorithm	Kruskal's Algorithm
• Builds a MST by iteratively adding the outie that is closest to the current tree. • Can be implemented several ways: 1. Adjacency matrix (dense graph) with linear search, worst-case time complexity $\Theta(\|V\|^2)$ 2. Adjacency list (sparse graph) with binary heap, worst-case time complexity $\Theta(\|E\|\log(V))$ 3. Adjacency list (sparse graph) with Fibonacci heap, worst-case time complexity $\Theta(\|E\|+\|V\|\log(V))$ • The linear search implementation of Prim's algorithm is ideal for dense graphs. • The heap implementation of Prim's works better on sparse graphs (among the heap implementations, a Fibonacci heap works better than a binary heap, since it supports priority updates in amortized constant time).	• Builds a MST by greedily adding edges that do not produce a cycle, in order of edge weight. • Implemented using a sorting algorithm and a union-find (disjoint set) container, has worst-case time complexity $\Theta(\|E\|\log(\|E\|)) = \Theta(\|E\|\log(V))$. Relies on an efficient implementation of the sorting algorithm and union-find. • Kruskal's algorithm is best suited for sparse graphs.

Both Prim's and Kruskal's algorithms work even if the underlying graph contains negative edges, as the logic used by these algorithms is not affected by the sign of an edge (a negative edge simply lowers the total cost of the MST but does not change it, as long as the relative ordering of the other edge weights are the same).

However, these algorithms do *not* work on *directed* graphs. Prim's algorithm makes the assumption that every node is reachable from every other node, which is not true for directed graphs. Thus, you could run into a situation where there is no directed edge that connects an innie with an outie, which would cause Prim's algorithm to get stuck and fail even if there are vertices that remain unexplored. Similarly, directed edges could prevent Kruskal's algorithm from properly detecting cycles using union-find, since two vertices may be in the same disjoint set even if there is no way to travel between the two vertices due to edge direction. With directed edges, it may also be needed for cycle-inducing edges to be added to minimize the weight of the solution, which is something that Kruskal's algorithm cannot do.

Lastly, it should also be mentioned that the concept of a "minimum spanning tree" does not make much sense for directed graphs. For directed graphs, a similar concept known as a *minimum spanning arborescence* is used instead. You will not need to know this for this class, but an *arborescence* is a directed graph where, for each vertex u, there is exactly one directed path from u to any other vertex v. An algorithm known as *Edmonds' algorithm* can be used to find the spanning arborescence of a directed graph that minimizes edge weight, with the same time complexity as that of Prim's algorithm.