# Module 9 - Graph Algorithms

## Exhaustive Search

When considering the processing of a graph $G = (V, E)$, one problem to be solved is searching the graph for a particular vertex of that graph having certain characteristics. Often such a search amounts to "walking" the graph along various edges and constructing various paths through the graph until the desired vertex is found. Within this context, we will consider various algorithms for performing such a search.

The initial set of graph search algorithms we will consider are referred to as "exhaustive" because, in the limit, they may require searching the set of all possible vertices. When evaluating any search strategy, but especially exhaustive search strategies, we are interested in evaluating a number of different performance criteria. Of particular interest are the following:

1. Completeness: A "complete" search strategy is one that guarantees finding a target vertex or path given such a target vertex or path exists.

2. Time Complexity: As with any other algorithm, "time complexity" refers to a measure of the amount of work required by the execution of an algorithm.

3. Space Complexity: Similar to time complexity, "space complexity" refers to a measure of the amount of memory required by the execution of an algorithm.

4. Optimality: When searching for some vertex or path, a cost measure can be applied to the search process and the cost of the path by which the vertex is found determined. This provides a measure of the cost of that path. An optimal search algorithm finds the path to the element that minimizes cost. Optimality is sometimes referred to as "admissibility."

We now consider two specific exhaustive search methods—breadth-first search and depth-first search.

### Breadth-First Search

When exploring alternative search strategies, we need a way to manage the vertices to be explored in some orderly way. Ultimately, this depends on defining an appropriate data structure for holding candidate vertices. For breadth-first search, we will use a simple queue. In this case, from the "current" vertex in our search, we will place adjacent vertices to be explored on the queue. This means we will be exploring vertices in a first-in first-out order.

Using this approach, we find that our search algorithm is complete. This is because we end up potentially exploring all possible vertices (thus breadth-first search is exhaustive). This leads to the question of optimality. Unfortunately, we will find that the optimality of breadth-first search depends on the nature of the costs applied to the edges of the graph. If all of the edge costs are uniform (i.e., the same), then breadth-first search will find an optimal path to the desired vertex. Unfortunately, non-uniform costs result in our losing any guarantee of finding that optimal path.

**The Algorithm** Let's consider the algorithm itself. To do this, we will need some terminology. For a vertex $u$, let $\pi[u]$ denote the predecessor of vertex $u$ in a traversal of the graph. Let $d[u]$ be the distance (i.e., cost) from the source vertex $s$ (i.e., the vertex from which search begins) to vertex $u$. We will "color-code" the vertices as we search the graph as a way of keeping track of our progress. Let $color[u]$ indicate the state of discovery for vertex $u$ where WHITE indicates the vertex has not been "discovered," GRAY indicates the vertex has been discovered but not fully explored, and BLUE indicates the search algorithm has finished exploring all options from the vertex.

Given the above, the pseudocode for breadth-first search is as follows:

**Algorithm 1** Breadth-First Search

```
BFS(G,s)
   for each vertex u ∈ V[G] do
      color[u] ← WHITE;
      d[u] ← ∞;
      π[u] ← NIL;
   color[s] ← GRAY;
   d[s] ← 0;
   π[s] ← NIL;
   Q ← {s};
   while Q ≠ ∅ do
      u ← head[Q];
      for each v ∈ Adj[u] do
         if color[v] = WHITE then
            color[v] ← GRAY;
            d[v] ← d[u] + 1;
            π[v] ← u;
            ENQUEUE(Q,v);
      DEQUEUE(Q);
      color[u] ← BLUE;
```

To interpret, search begins at a source vertex $s$. The first loop initializes the search process by setting the colors of all of the vertices to WHITE, the distances covered in search to infinity (or some very large value), and the predecessors to NIL. Distances and predecessors will be updated as the search proceeds since they will identify how the search was performed. After initializing, the distance and predecessor for s are set to 0 and NIL respectively. These will be their final values. The queue is then initialized to contain only the source vertex.

In performing our search, we will dequeue a vertex from $Q$, identify the vertices adjacent to that vertex, and if their colors are WHITE, enqueue them only $Q$. This is what happens in the second while loop. When an adjacent vertex is added to $Q$, its color is set to GRAY, its predecessor is set to point to the current vertex, and its distance is set to be one more than the distance to its predecessor vertex $v$. After adding the adjacent vertices to $Q$, the current vertex's color is set to BLUE.

This algorithm, as presented, does no more than walk the graph until all vertices are traversed (thus emphasizing the exhaustive nature of the algorithm). If we are searching for a vertex with certain properties, then another termination criterion can be incorporated into the algorithm. Since the set of predecessor pointers are constructed as search occurs, once the desired vertex is found, the path to that vertex can be reconstructed from these pointers.

**Analysis** Now we analyze the complexity of this algorithm. Clearly, the algorithm's performance is bound by the size of the graph since we end up exploring all of the vertices of the graph. But notice that, after initialization, no vertex is ever enqueued or dequeued more than once because of the color coding of the vertices. Recall that each enqueue and dequeue operation requires $O(1)$ time; therefore, the entire queuing effort is $O(|V|)$. We also see that at most $|E|$ edges are traversed thus requiring $O(|E|)$ time. Thus the total time complexity for breadth-first search is $O(|V| + |E|)$.

### Depth-First Search

Depth-first search is similar to breadth-first search in that it too explores all possible vertices of a graph (in the limit). The primary difference (in fact, it can be shown that it is the only significant difference) is the data structure that is used. Rather than a queue, depth-first search places vertices to be explored into a stack. When the graph is finite (which will be true for all graphs explored in this class), then depth-first

search is also complete. Unfortunately, even with uniform cost edges, depth first search is not guaranteed to find an optimal path.

**The Algorithm** To prepare for the algorithm, we define some terms. First, let $d[u]$ be the "time" (or distance from $s$) that vertex $u$ is first discovered. Then let $f[u]$ be the time the search process finishes with vertex $u$. We will use other definitions consistent with breadth-first search (namely predecessor and color). The main driver for depth-first search is as follows:

**Algorithm 2** Depth-First Search

```
DFS(G)
   for each vertex u ∈ V[G] do
      color[u] ← WHITE;
      π[u] ← NIL;
   time ← 0;
   for each vertex u ∈ V[G] do
      if color[u] = WHITE then
         DFSVisit(u);
```

As with breadth-first search, the first loop initializes the algorithm by setting all of the colors of the vertices to WHITE and all of the predecessor pointers to NIL. The "time" is then set to 0. Then, assuming some ordering of the vertices, each WHITE vertex is explored using DFSVisit. This is significant because it permits multiple entry points into the graph. What is also significant is that depth-first search, as defined here, operates on a directed graph where breadth first search assumed an undirected graph.

The bulk of the search occurs in DFSVisit, and the pseudocode for this function is as follows:

**Algorithm 3** DFS Visit

```
DFSVisit(u)
   color[u] ← GRAY;
   d[u] ← time ← time + 1;
   for each v ∈ Adj[u] do
      if color[v] = WHITE then
         π[v] ← u;
         DFSVisit(v);
   color[u] ← BLUE;
   f[u] ← time ← time + 1;
```

For a specific vertex $u$, we change the color to GRAY, set its discovery time to the current time plus one, and then proceed to consider each of the vertices adjacent to $u$. For any of the adjacent vertices currently colored WHITE, those vertices have their predecessors set to $u$, and they are then explored through the recursive call to DFSVisit. Notice that the recursive nature of the algorithm is what implements the stack. Since the subtree rooted at $u$ is completely scanned, $u$'s color is set to BLUE, its finish time set to the current search time plus one, and control is returned to the previous level in the recursion.

As we consider the execution of depth-first search, we will apply a classification scheme to the edges as they are explored. Search effectively builds a "tree" structure since re-visiting a previously visited vertex signifies the end of a search path. That path is then terminated before the cycle to the previously visited vertex is made. Given this view of search, we say an edge is a tree edge in the depth-first forest (it is more than a simple tree because of searching a directed graph ... one tree for every starting point in the search). Edge $(u, v)$ is a tree edge if v was discovered when exploring the edge $(u, v)$ in the graph.

Similar to breadth-first search, this algorithm, as presented, does no more than walk the graph until all vertices are traversed (thus emphasizing the exhaustive nature of the algorithm). We can apply similar

termination criteria if we are searching for a vertex with certain properties and return the search path using the set of predecessor pointers are constructed as search occurs.

When expanding the edges of the graph, we define the following:

**Definition:** A back edge is an edge in the depth-first search that connects vertex $u$ to some vertex $v$ that is an ancestor of $u$ in the depth-first tree.

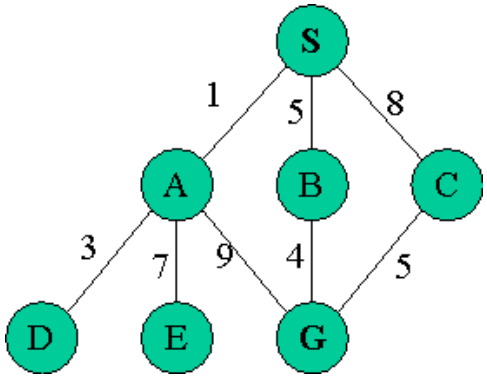
**Definition:** A forward edge is an edge in the depth-first search that connects vertex $u$ to some vertex $v$ that is an existing descendant of $u$ in the depth-first tree.

**Definition:** A cross edge is an edge in the depth-first search that connects vertex $u$ to some vertex $v$ that exists in some other depth-first tree in the forest.

**Analysis** Now we analyze the performance of depth-first search. The initialization step requires $O|V|)$ time as in breadth-first search. DFSVisit occurs exactly once for every vertex in the graph because of its dependence on color. Thus there are $O(|V|)$ calls to DFSVisit. Within DFSVisit, each edge is considered only once, thus requiring $O(|E|)$ time. Thus, DFSVisit requires time $O(|V| + |E|)$.

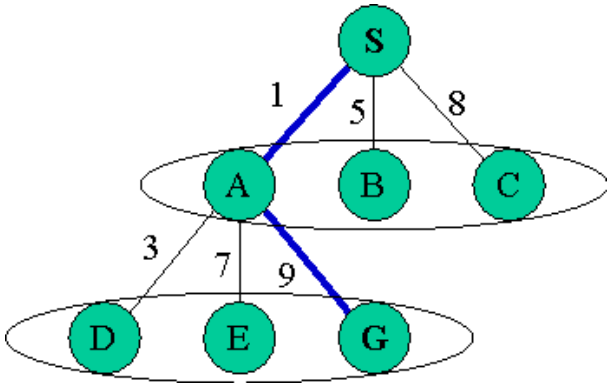
### Optimal Graph Search

As suggested above, suppose we wish to find a path from the source vertex s to a particular vertex, identified by some characteristic or set of properties? In addition, suppose we wish to find the path that minimizes the cost along the edges? For example, suppose we are presented with the following graph:



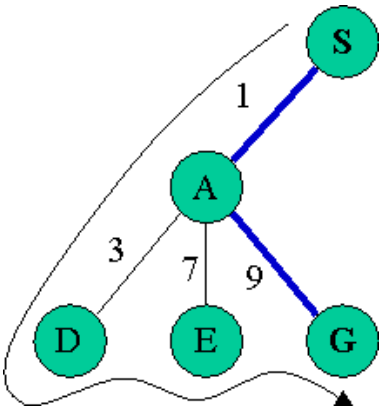

In this example, we are searching from the vertex designated by "S" and want to find the optimal path to the vertex designated by "G." Simple inspection reveals that the path S–B–G yields the optimal cost path with a total cost of $5 + 4 = 9$.

Suppose we apply breadth-first search to this graph. If we do that, we obtaining a graph with the following edges explored:



In this figure, we point out that breadth-first search explores the graph in levels, and the expansion of a vertex's adjacent vertices occur in left-to-right fashion. This means that goal vertex "G" will be found along the path through "A" before either of the two remaining paths because of the blind way the edges are ordered. Consequently, path S–A–G is found with a cost of 10. This is a suboptimal path.

What about depth-first search? Will performance be better if we use this alternative algorithm? Since we explained above that at least breadth-first search is optimal with uniform cost but depth-first search is not even optimal then, it seems unlikely depth-first search will improve our situation. In fact, the following is the resulting graph identifying the explored edges:




As we see, for this graph, depth-first search returns exactly the same suboptimal path as breadth-first search. What is interesting is that fewer vertices needed to be explored to get this path. Note, however, that such a result is not guaranteed; however, depth-first search is often found empirically to find a path more quickly than breadth-first search unless the paths explored in the tree prior to reaching the goal go significantly deeper than the position of the goal vertex.

We will revisit this problem of finding an optimal path in a graph later in this discussion.

## Connected Components

In this section, we will accomplish two things. First, we will present an interesting algorithm to solve a specific graph algorithm—the problem of finding connected components in a directed graph. Then we will provide a detailed proof of correctness for this algorithm. Up until now, our focus has been on analyzing the computational complexity of algorithms. By presenting the correctness proof, we focus on the second theoretical skill to be developed in algorithm design, that of proving that the designed algorithm performs that task specified.

### The Algorithm

Recall that we defined several different aspects of connectivity in a graph. Specifically, we defined the following:

**Definition:** An undirected graph is connected if for every pair of vertices $(v_i, v_j)$, a path exists between $v_i$ and $v_j$.

**Definition:** A directed graph is strongly connected if for every pair of vertices $(v_i, v_j)$, a directed path exists between $v_i$ and $v_j$.

**Definition:** A directed graph is said to be weakly connected if for every pair of vertices $(v_i, v_j)$, a corresponding undirected path (i.e., a path assuming the directed edges are actually undirected) exists between $v_i$ and $v_j$.

**Definition:** A connected component is a maximal subgraph of a graph $G$ for which one of the above types of connectivity applies (i.e., a subgraph for which adding any other portion of the graph $G$ will violate the connectedness property).

Because strongly directed components are the most constrained (and therefore the most difficult to identify within a graph) we will focus on identifying them within a graph. To do this, we need to concept of a graph transpose.

**Definition:** Let $G = (V, E)$ be a directed graph. Then the transpose of $G$, denoted $G^T = (V, E^T)$, is a graph such that $E^T = \{(v, u)|(u, v) \in E\}$.

Thus $G^T$ is simply $G$ with the directions on all of $G$'s edges reversed. What we will see is that $G$ and $G^T$ have exactly the same set of connected components, and we use that property now to define the algorithm for finding these connected components.

**Algorithm 4** Strongly Connected Components

```
StronglyConnectedComponents(G)
   DFS(G);   // Compute finish times for all u
   compute G^T;
   DFS(G^T);  // Process vertices in order of decreasing f[u]
   output vertices of each tree as a separate connected component;
```

Key to finding connected components in this algorithm is the first step—finding the finish times for all of the vertices from the initial depth-first search of the graph. These finish times are then used to order the vertices for a second depth-first search, but this time using the graph transpose. Depth-first search produces a set of depth-first trees (as we saw earlier). What is interesting is that each depth-first tree found in the graph transpose corresponds to a strongly connected component of the original graph $G$, the proof of which can be found in the textbook.

## Minimum Spanning Trees

The next topic to consider has application in networking and vehicle routing—finding the minimum spanning tree of a graph.

**Definition:** Given a graph $G$, a spanning tree of $G$ is a subgraph of $G$ such that the subgraph is a tree, and all connected components from $G$ are still connected in the subgraph.

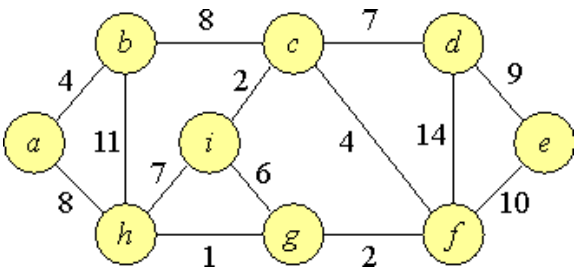
Now let's assume we associate a weight $w(u, v)$ on each edge $(u, v)$ in the graph $G$. Let $w(T) = \sum_{(u,v)\in T} w(u, v)$ be the weight of spanning tree $T$.

**Definition:** Given a graph $G$, a minimum spanning tree of $G$ is the spanning tree $T$ that minimizes $w(T)$.

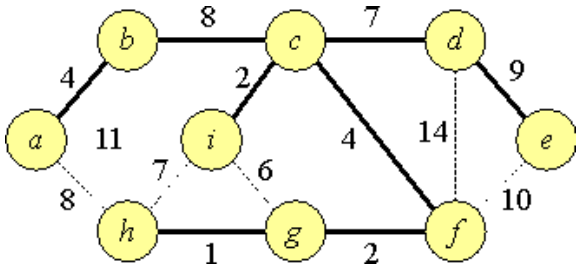
Thus there are two properties for a minimum spanning tree:

1. $T$is a tree.
2. $w(T)$is minimum.

For example, consider the following graph:



Our task will be to find a spanning tree of a graph such as this that minimizes the cost $w(T)$ given the set of weights on the edges of that tree. For this graph, the following is one example of a minimum spanning tree. As we will see a particular graph may have more than one minimum spanning tree.




In this figure, the heavy black lines indicate the edges that are part of the spanning tree, and the dashed lines are edges in the original graph that have been removed to form the spanning tree.

### Growing a Minimum Spanning Tree

The general approach we will take to find a spanning tree will be to "traverse" the graph and decide along the way whether to include an edge in the spanning tree or not. As we prepare to discuss specific algorithms to do this, we need some more definitions.

**Definition:** A safe edge in a graph is an edge that can be added to a minimum spanning tree at some time t without violating the minimum spanning tree property.

In this definition, we focus on the two properties given above. Note, however, that connectivity is a property as well. Unfortunately, as we are growing the tree, we cannot maintain the connectivity property since, by definition, a partially complete spanning tree will not be connected. Therefore, we reserve the connectivity property to the end of the algorithm. Here is pseudocode for a generic approach to growing a spanning tree:

**Algorithm 5** Generic MST

```
GenericMST(G, w)
   A ← ∅;
   while A does not form a spanning tree do
      find edge (u, v) that is safe for A;
      A ← A ∪ {(u, v)};
   return A;
```

We will present two different "greedy" algorithms for finding minimum spanning trees based on the above. As shown, we will incrementally find an edge according whatever greedy measure we are using and then add it to the spanning tree as long as it is safe (i.e., it does not create a cycle in the graph).

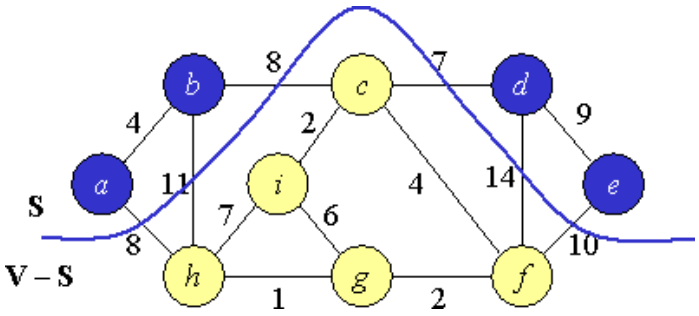
**Definition:** Given an undirected graph $G$, a cut of $G$ is a partition of the vertices of $G$.

**Definition:** Given a cut $(S, V - S)$, an edge crosses the cut if one endpoint of the edge is in $S$ and the other endpoint is in $V - S$.

**Definition:** A cut $(S, V - S)$ respects a set of edges $A$ if no edge in $A$ crosses the cut.

**Definition:** An edge $(u, v)$ is a light edge crossing a cut if its weight is a minimum of any edge crossing the cut.

To illustrate the above concepts, consider our sample graph with the following cut:




In this figure, the partitioning of the vertices is shown by color. The partition $S$ consists of the blue vertices, $a$, $b$, $d$, and $e$. The partition $V - S$ contains the remaining yellow vertices, $c$, $f$, $g$, $h$, and $i$. The edges $(a, h)$, $(b, h)$, $(b, c)$, $(c, d)$, $(d, f)$, and $(e, f)$ cross the cut. To illustrate the concept of a light edge, we need to consider situations under which edges will be added to the minimum spanning tree. To do that, we need to determine conditions under which edges are safe.

**Theorem:** Let $G$ be a connected, undirected graph with real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree of $G$. Let $(S, V - S)$ be any cut that respects $A$. Let (u, v) be a light edge crossing $(S, V - S)$. Then $(u, v)$ is safe for $A$.

**Proof:** Let $T$ be a minimum spanning tree for $G$ that includes $A$. Assume $T$ does not include light edge $(u, v)$, since if it did, we would be done. Now construct a new minimum spanning tree $T'$ that includes $A \cup \{(u, v)\}$ as follows. First, suppose we add $(u, v)$ to $T$. This will create a cycle since $T$ is already a spanning tree. Thus, there must exist some other edge $(x, y) \in T$ that connects $u$ and $v$ in $T$ via a path $p$ that crosses the cut. This edge cannot be in $A$ because we said the cut respects $A$. Since $(x, y)$ is on a unique path $p$ from $u$ to $v$ in $T$, removing $(x, y)$ will break $T$ into two components. Then, adding $(u, v)$ back into $T$ will reconnect the components and create a new spanning tree $T'$. Since $(u, v)$ was assumed to be a light edge, $w(u, v) \leq w(x, y)$. Therefore, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$. But recall that $T$ was assumed to be a *minimum* spanning tree, meaning $w(T) \leq w(T')$. The only way for this to be possible is if $w(T) = w(T')$, indicating must also be a minimum spanning tree. Since $A \subseteq T$ and $(x, y) \notin A$, we must also have that $A \subseteq T'$. Therefore, $A \cup \{(u, v)\} \subseteq T'$. Finally, since $T'$ is also a minimum spanning tree, this means $(u, v)$ must be safe for $A$. QED

To strengthen this result as we prepare to define our algorithms, we have the following corollary.

**Corollary:** Let $G$ be a connected, undirected graph with real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree of $G$. Let $C$ be a connected component (tree) in the forest of $G_A = (V, A)$. If $(u, v)$ is a light edge connecting $C$ to some other component in $G$, then $(u, v)$ is safe for $A$.

**Proof:** The cut $(C, V - C)$ respects $A$; therefore, $(u, v)$ is a light edge for this cut. QED

### Kruskal's Algorithm

The first algorithm we will examine, Kruskal's algorithm, grows a minimum spanning tree by implementing the UNION-FIND algorithm we discussed last week. The pseudocode for this algorithm is as follows:

**Algorithm 6** Kruskal's Algorithm

```
MSTKruskal(G, w)
    A ← ∅;
    for each v ∈ V[G] do
        MakeSet(v);
    sort edges of E by nondecreasing weight w;
    for each (u, v) ∈ E do in sorted order
        if FindSet(u) ≠ FindSet(v) then
            A ← A ∪ {(u, v)};
            Union(u, v);
    return A;
```

In this algorithm, the first step is to create one set for every vertex in the graph $G$. Next the edges of the graph are sorted by their weight. Then each edge is considered in nondecreasing order by checking the vertices at the endpoints to see if they are in different sets. If they are in different sets, then the edge is crossing a cut defined by those two sets. Because of the way the sets are maintained, a) we know that edge is a light edge (because of sorting the edges by weight), and b) we know that the edge is a safe edge (because it crosses a cut). After adding the edge to the tree, the two sets defining the cut are merged into a new set.

In analyzing the performance of this algorithm it would seem that the algorithm's complexity depends on the implementation of UNION-FIND. The initialization step can be completed in $O(|V|)$ time, and edge sorting requires $O(|E| \lg |E|)$ time. Recall that, if implemented properly, the operations on the disjoint set forest require $O(|E| \lg^* |V|)$ time. This means the sort dominates the process, so finding a minimum spanning tree with Kruskal's algorithm requires $O(|E| \lg |E|)$ time.

### Prim's Algorithm

Another greedy approach to finding minimum spanning trees, called Prim's algorithm, uses a priority queue rather than disjoint sets to maintain the construction. The pseudocode for Prim's algorithm is as follows:

**Algorithm 7** Prim's Algorithm

```
MSTPrim(G, w, r)
    Q ← V[G];
    for each u ∈ Q do
        key[u] ← ∞;
    key[r] ← 0;
    π[r] ← NIL;
    while Q ≠ ∅ do
        u ← ExtractMin(Q);
        for each v ∈ Adj[u] do
            if v ∈ Q and w(u, v) < key[v] then
                π[v] ← u;
                key[v] ← w(u, v);
```

In this algorithm the first step is to define a priority queue as a heap and store all of the vertices (not the edges) on that queue. A key value is associated with each vertex in the queue, and initially, the key value is set to infinity (or some very large number). In this algorithm, some vertex is selected arbitrarily to act as the root of the tree, and that root vertex is passed into the algorithm. This root vertex receives a key value of zero, and its predecessor is set to NIL. The priority queue is heapified as it is created; however, since all key values are infinite (except the root), the order of the vertices is largely arbitrary.

Following initialization, the first step of the algorithm is to extract the minimum vertex from the queue. This will be the root vertex because of setting its key value to zero. Then the adjacent vertices will be examined, and their key values will be adjusted by comparing the edge weight from the current vertex to the adjacent vertex to the current key value. If the new value is less, then the value is changed to equal the edge weight. When such an update occurs, the predecessor link for the adjacent vertex is updated to point to the current vertex.

The algorithm does not show the "safeness" check; however, one must be careful not to update the predecessor link if it creates an unsafe edge. What is interesting about this algorithm, however, is that safeness is guaranteed because of actually extracting the vertices from the priority queue. Because of the check to see if the adjacent vertex is in the queue, only vertices not yet explored will be updated.

Now we need to analyze the performance of the algorithm. In this case, it would appear the complexity depends on the implementation of the priority queue. Recall that the heap can be constructed in $O(|V|)$ time. The interior of the while loop will be executed $|V|$ times, since each vertex is considered exactly once. Each call of ExtractMin will require $O(\lg |V|)$ time because of re-heapifying after the extraction; therefore, the total complexity for ExtractMin is $O(|V| \lg |V|)$. The inner loop will execute $O(|E|)$ times. Adjusting keys using DecreaseKey (with associated re-heapification) requires $O(\lg |V|)$ time. This means the total complexity of the algorithm is $O(|V| \lg |V| + |E| \lg |V|) = O(|E| \lg |V|)$. It is interesting to consider when Prim's algorithm is better than Kruskal's algorithm, and vice versa. Clearly, the answer to this depends on the relative difference in size of $V$ and $E$.