Chapter 9

Graph Algorithms

The previous chapter introduced graphs and the basic traversal algorithms, depth-first search and breadth-first search. In this chapter, we will look at several useful graph algorithms, starting with two applications of search.

In our analysis of these algorithms, we will use the notation $|A|$ to represent the number of elements of a set A. For an algorithms operating on a graph $G = (V, E)$, we will analyze the run time of the algorithm in terms of $|V|$, the number of vertices, and $|E|$, the number of edges.

9.1 Connected Components

Recall that an undirected graph is connected if for every pair of vertices, there is a path in the graph between those vertices. A connected component of an undirected graph is a maximal connected subgraph of the graph. Every vertex of the graph lies in a connected component that consists of all the vertices that can be reached from that vertex, together with all the edges that join those vertices. If an undirected graph is connected, there is only one connected component.

We can use a traversal algorithm, either depth-first or breadth-first, to find the connected components of an undirected graph. If we do a traversal starting from a vertex v, then we will visit all the vertices that can be reached from v. These are the vertices in the connected component that contains v. If there are other connected components, then there will still be unvisited vertices after the traversal is complete. We can do a traversal starting from one of those vertices to find another connected component. If we continue in this way until all vertices have been visited, then we will have discovered all the connected components.

In the following algorithm, we count the connected components and print out the vertices in each component. We use breadth-first search to do the traversal, but depth-first search would work just as well. The algorithm is given in pseudocode that can be adapted to either an adjacency matrix or edge-list representation of the graph.

```
for (int i = 0; i < |V|; i++)
   visited[i] = false;   // Mark all nodes as unvisited.
```

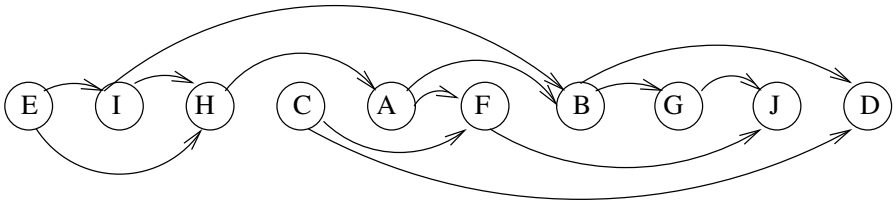
```
int compNum = 0;         // For counting connected components.
for (int v = 0; v < |V|; v++) {

      // If v is not yet visited, it's the start of a newly
      // discovered connected component containing v.

   if ( ! visited[v] ) { // Process the component that contains v.
      compNum++;
      cout << "Component " << compNum << ":  ";
      IntQueue q;   // For implementing a breadth-first traversal.
      q.enqueue(v); // Start the traversal from vertex v.
      visited[v] = true;
      while ( ! q.isEmpty() ) {
         int w = q.dequeue();  // w is a node in this component.
         cout << w << " ";
         for each edge from w to some vertex k {  // ***
            if ( ! visited[k] ) {
                  // We've found another node in this component.
               visited[k] = true;
               q.enque(k);
            }
         }
      }
      cout << endl << endl;
   }
}

if (compNum == 1)
   cout << "The graph is connected.";
else
   cout << "There are " << compNum << " connected components.";
```

The for loop marked *** will be executed exactly once for each vertex in the graph. That is, it is executed $|V|$ times. If an adjacency list representation is used, the for loop will run for w from 0 to $|V|$ each time it is executed, so the run time for the adjacency list representation is $\Theta(|V|^2)$.

For an edge-list representation, the for loop is really a traversal of an edge list, and the body of the for loop is executed twice for each edge in the graph—twice, because an edge in an undirected graph occurs in two edge lists, one for each of its endpoints. So, the body of the for loop is executed $2 * |E|$ times, and the total run time for the algorithm is $\Theta(|V| + |E|)$.

9.2 Topological Sort

In a dag (Directed Acyclic Graph), it is possible to arrange the vertices of the graph in a linear order $v_0, v_1, v_2, \ldots$ such that for any edge $\overrightarrow{v_i v_j}$ in the graph, we will have that $i < j$. Such an ordering is called a **topological sort** of the dag and the vertices are said to be in **topological order**. If we visualize the vertices laid out in a line in topological order, then all the edges of the graph go from left to right:



Note that a topological sort is not unique; that is, a given dag will generally have many different topological orderings. In the graph shown above, for example, the vertices C, A, F could be reordered as A, C, F or A, F, C, and the graph would still be topologically sorted.

Theorem 9.1. *A directed graph can be topologically sorted if and only if it is acyclic.*

Proof. If a graph is not acyclic, then it contains a cycle of vertices, and there is no way to order the vertices in that cycle so that all edges will go from left to right.

On the other hand, if the graph is acyclic, then it must have at least one vertex that has no incoming edges. To see this, let $u_1 u_2 \ldots u_m$ be a path of maximal length in the graph. Since the graph is acyclic, no vertex can occur more than once in any path. I claim that u_1 has no incoming edges. For if $\overrightarrow{u_0 u_1}$ were an edge, then $u_0 u_1 u_2 \ldots u_m$ would be a path in the graph with length greater than the length of $u_1 u_2 \ldots u_m$, which contradicts the fact that this path has maximal length.

So, suppose that G is a directed acyclic graph. Let v_0 be a vertex in G that has no incoming edges. Then we can let v_0 be the first vertex in a topological sort of G. Now, consider the graph G_1 obtained by deleting from G vertex v_0 and all the edges that exit v_0. G_1 is a dag and so has a vertex v_1 that has no incoming edges in G_1. We can let v_1 be the second vertex in the topological sort of G. Next, we let G_2 be the graph obtained by deleting from G_1 v_1 and all edges that exit v_1. The next vertex, v_2, in the topological sort of G is selected as a vertex in G_2 that has no incoming edges in G_2. Continuing in the same way, we obtain a complete topological sort of G.[1] □

The last paragraph in this proof is actually an outline of an algorithm for finding a topological sort of a dag. However, this algorithm is not very efficient, and we can do better. We can use a version of recursive depth-first search to do topological sorting in run time $\Theta(|V| + |E|)$ (for an edge-list representation of the graph).

In a topological ordering of the vertices of a dag, all the descendents of a given vertex must come after that vertex. A recursive depth-first search starting from a vertex v first visits v and then recursively visits all the descendents of v. If we list the vertices in the order in which they are visited,

[1] This proof should really be phrased as a proof by mathematical induction.

they will be in topological order, since every vertex will be listed before its descendents. A single traversal, starting from a given vertex, will not necessarily visit every vertex in the graph (only the descendents of the starting vertex). To obtain a topological sort of all vertices, if there are unvisited vertices after the first traversal, we must do another traversal starting from an unvisited vertex and visiting only vertices that were not already visited in the first traversal. In the topological order, all vertices encountered during the second traversal *precede* all vertices encountered during the first traversal. Further traversals will be needed if there are still unvisited vertices. The topological sort at the beginning of this section could be generated by a traversal starting from vertex A, followed by a traversal starting from vertex C followed by a traversal starting from vertex E.

To implement this algorithm, we can use the following global variables, where $N = |V|$:

```
bool visited[N];   // Keep track of which vertices are visited.
int topoOrder[N];  // List vertices in topological order.
int count;         // Number of vertices placed in topoOrder.
```

We modify the recursive DFS procedure so that it places vertices into the *topoOrder* array. It is convenient to fill the array in reverse order, dropping each vertex into the array after all its descendents have been processed. This will allow successive traversals to place the vertices that they encounter into the array in the correct order.

```
void topoSortDFS( int v ) {  // Do a DFS starting from v.
   if ( ! visited[v] ) {
      visited[v] = true;
      for each edge from v to another vertex w {  // Pseudocode!
         topoSortDFS( w );
      }
      count++;
      topoOrder[ N-count ] = v;
   }
}
```

Finally, to implement the topological sort algorithm, we must first mark all vertices as unvisited and then do a series of traversals until all vertices have been visited:

```
for ( int v = 0; v < N; v++ )
   visited[v] = false;
count = 0;  // Start with no vertices in the topoOrder array
for ( int v = 0; v < N; v++ )
   topoSortDFS( v );
```

If this algorithm is applied to a directed acyclic graph, the result will be that the N vertices of the graph are stored in the array *topoOrder* in topological order.

9.3 Minimal Spanning Trees

In this section, we will be working with connected, weighted, undirected graphs. We assume throughout this section that all the weights are positive numbers. In many such graphs, the weight of an edge often represents a cost or distance. Let $G = (V, E)$ be such a graph. Suppose that we want to find a set of edges that connect all the vertices in the graph for the minimum possible cost. That is, we want to find a subset T of E such that the subgraph $G' = (V, T)$ is still connected and the sum of the weights of all the edges in T is as small as possible. We can note first of all that T must be acyclic. For suppose $v_1 v_2 \dots v_n$ is a cycle in T, and let e be any edge in this cycle. If we remove e from the graph G', it is clear that the resulting graph is still connected, and the total weight of all the edges in the resulting graph is less that the total weight of the edges in T. But this contradicts the minimality of T.

A connected, acyclic, undirected graph is called a **free tree**. If $G = (V, E)$ is a connected undirected graph, a **spanning tree** for G is a subgraph that is a tree and that includes all the vertices of G. If G is weighted, then we can look for a **minimal spanning tree**, that is, a spanning tree in which the total weight of all the edges is as small as possible. In this section, we will first consider some of the general properties of free trees. We will then look at an efficient algorithm for finding minimal spanning trees.

Free trees have several useful properties. Let $G = (V, T)$ be a free tree, and let u and v be two vertices in V. Then there is a unique simple path in G from u to v. We know that there is at least one simple path, since G is connected. Suppose that there were two distinct simple paths $u_1 u_2 \dots u_n$ and $v_1 v_2 \dots v_m$ with $u_1 = v_1 = u$ and $u_n = v_m = v$. Then there would be a cycle in the graph, contradicting the fact that the graph is acyclic. (To find the cycle, let u_i be the first vertex in the first path for which $u_i \neq v_i$, and let u_k be the first vertex in the first path such that $k > i$ and $u_k = v_j$ for some v_j in the second path. We know k exists since $u_n = v$ is in the second path. Then the cycle is $u_{i-1} u_i \dots u_k v_{j-1} v_{j-2} \dots v_{i-1}$, which is a path since $u_k = v_j$ and is a cycle since $u_{i-1} = v_{i-1}$.)

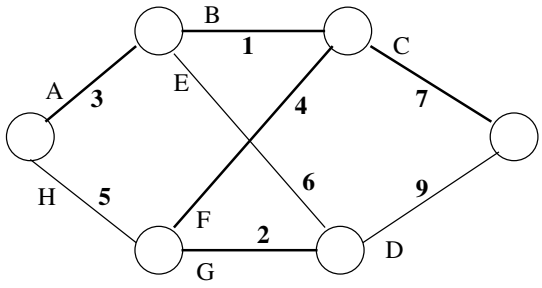
A free tree is called "free" because it has no root. However, if v is any vertex in a free tree, we can select v to be a root and any other vertex, u, becomes a descendent of v via the unique simple path from v to u. This gives the free tree an ordinary, rooted tree structure. From now on in this section, we will refer to a free tree simply as a tree.

Suppose that $G = (V, T)$ is a tree, that is, a connected, acyclic, undirected graph. Then $|T| = |V| - 1$. That is, in a tree, the number of edges is one less than the number of vertices. To prove this, let v_0 be any vertex. Let V' be the set obtained by removing v_0 from V. Then $|V'| = |V| - 1$. To prove that $|T| = |V| - 1$, we define a one-to-one correspondence $f \colon V' \to T$. Let $u \in V'$. We know that there is a unique simple path in G from v_0 to u. Since $v_0 \neq u$, this path contains at least one edge. Let e be the last edge in the simple path from v_0 to u. Then we define $f(u)$ to be e. f is well-defined because there is only one simple path from v_0 to u. Note that u is one of the vertices of the edge $f(u)$, and it is endpoint of this edge that is farther from the root. This implies that f is one-to-one, because for any edge, there is only one vertex that satisfies this condition. f is onto because an edge $e = \overline{u_1 u_2}$ must be the final edge in the simple path from v_0 to either u_1 or to u_2. (Let $v_0 v_1 \dots v_n$ be the simple path where $v_n = u_1$. If e is not the final edge in

this path, then it is the final edge in the path $v_0v_1 \dots v_nu_2$.)

Note that this implies that all spanning trees for any connected undirected graph contain the same number of edges, since the number of edges in any spanning tree is one less than the number of vertices in the graph.

Now suppose that $G = (V, E)$ is a weighted, connected, undirected graph, and assume that all the weights are positive numbers. One algorithm for finding a minimal spanning tree for G is **Kruskal's algorithm**. This algorithm builds a minimal spanning tree by adding edges one at a time. The edges of G are considered in order of increasing weight. For each edge, that edge is added to the tree if and only if doing so will not introduce a cycle into the tree. The process is finished when all the vertices of G are contained in the tree. For example, in the following graph



the edges are considered in the order B, G, A, F, H, E, C, D. The first four edges in this list can be added to the tree without creating a cycle. Adding H would create the cycle $ABFH$, so it is not added to the tree. Adding E would create the cycle $BFGE$, so it is not added. C can be added without creating a cycle, and this completes the tree since all vertices are now part of the tree. The resulting tree is a minimal spanning tree. The edges of the tree are shown in the graph as thick lines.

Kruskal's algorithm is simple in concept, but we still need a procedure for checking whether adding an edge will introduce a cycle. One way to do this would be as follows: When considering an edge $\overline{uv}$ for inclusion in the tree, do a search starting from u, following only edges that are already in the tree. If this search visits vertex v, then there is already a path in the tree from u to v and adding the edge $\overline{uv}$ would introduce a cycle in the tree. There are more efficient ways to check for cycles, but I will not cover them here. Using one of the alternative methods for checking for cycles, it is possible to implement Kruskal's algorithm with a running time of $\Theta(|E| * \log(|E|))$.

Note that Kruskal's algorithm does not produce a uniquely determined result in the case where several edges have the same weight. Edges of the same weight can be considered in any order, and different orders can produce different trees. However, all the different spanning trees that can be produced by Kruskal's algorithm will have the same weight, and they will all be minimal spanning trees.

It is not obvious that the spanning trees produced by Kruskal's algorithm do in fact have minimal weight. I will not prove this fact here.

Prim's algorithm is another well-known algorithm for finding minimal spanning trees. In Prim's algorithm, as in Kruskal's, the tree is grown one edge at a time. In Kruskal's algorithm, the set of

edges that have been added at a given step can form a disconnected graph. In Prim's algorithm, the set of edges that have been added is always connected.

For Prim's algorithm, we need to keep track of the set of vertices that have been added so far to the tree. Let P be this set. We start by adding an arbitrary vertex v_0 to the tree. Then we repeat the following procedure until the tree is complete: Choose the shortest edge $e = \overline{uv}$ such that $u \in P$ and $v \notin P$; add e to the tree, and add v to P. The result is a minimal spanning tree, although I will not prove this here, and I will not explain how to implement the selection of e efficiently. Prim's algorithm is similar to Dijkstra's shortest path algorithm, which we look at in the next section.

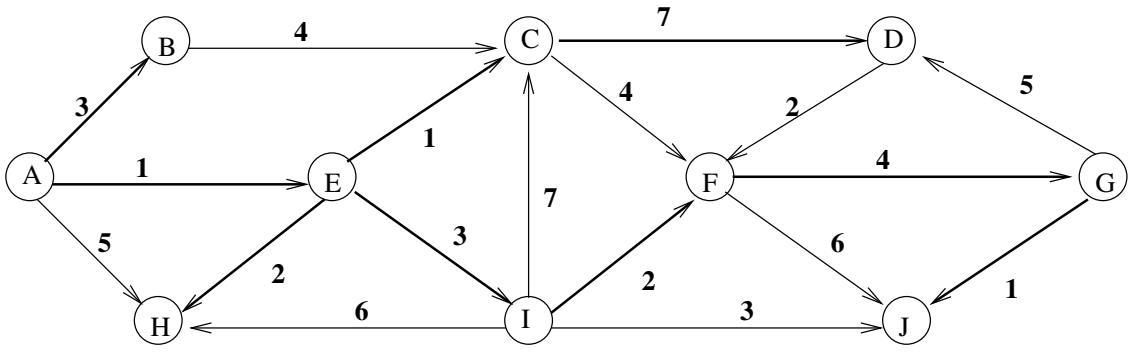
9.4 Shortest Paths

In this section, we consider directed weighted graphs in which all the weights are positive numbers. Given two vertices u and v in such a graph, we can try to find a minimal-cost path from u to v. That is, we want a path from u to v such that the sum of the weights of the edges in the path is as small as possible. Of course, we are not guaranteed that *any* path from u to v exists, but we observe that if there is a path, then the path of minimal weight will be a simple path.

We will consider the shortest path algorithm known as Dijkstra's algorithm. Given a vertex v_0, Dijkstra's algorithm will find a minimal weight path from v_0 to each vertex in the graph that is reachable from v_0. Although we will work with directed graphs, Dijkstra's algorithm also applies to undirected graphs. We just have to treat each edge $\overline{uv}$ in the undirected graph as a pair of directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$ with the same weight.

If $v_0 v_1 \dots v_n$ is a minimal weight path from v_0 to v_n, it must also be true that $v_0 v_1 \dots v_i$ is a minimal weight path from v_0 to v_i for $i = 0, 1, \dots, n-1$. This follows because if there is a shorter path from v_0 to v_i, then we could replace $v_0 v_1 \dots v_i$ in the path $v_0 v_1 \dots v_n$ to obtain a shorter path from v_0 to v_n.

If you look at all the shortest paths that start from v_0 and lead to other nodes in the graph, they form a tree. Dijkstra's algorithm builds up this tree one edge at a time. At each step, it adds an edge that joins a new vertex onto the tree. The edge that is chosen is the one for which the path to the newly added vertex is as short as possible. Consider the following graph, where the starting vertex, v_0, is vertex A:



We start with vertex A and no edges, and we look at all edges that connect A to other vertices in the graph. That is, we consider adding $\overrightarrow{AB}$, $\overrightarrow{AE}$, or $\overrightarrow{AH}$ to the tree. We choose the shortest of these, $\overrightarrow{AE}$, to add to the tree. The tree now contains vertices A and E. We look at all edges that connect one of these vertices to a vertex that is not yet in the tree, that is, at $\overrightarrow{AB}$, $\overrightarrow{AH}$, $\overrightarrow{EC}$, $\overrightarrow{EH}$, and $\overrightarrow{EI}$, and we select the edge that produces the shortest path from A. In this case, we select $\overrightarrow{EC}$ which produces the path AEC of length 2, and add it to the tree. The tree now contains A, E, and I. In the next step, we consider the edges $\overrightarrow{AB}$, $\overrightarrow{AH}$, $\overrightarrow{CD}$, $\overrightarrow{CF}$, $\overrightarrow{EH}$, $\overrightarrow{EI}$, $\overrightarrow{IH}$, and $\overrightarrow{IJ}$. Two of these edges, $\overrightarrow{AB}$ and $\overrightarrow{EH}$, produce paths of length 3. We can choose to add either of these edges. Continuing in this way, we eventually get the tree made up of the thick lines in the graph. This tree contains a minimal weight path from A to every other vertex in the graph.

Dijkstra's algorithm is actually more clever than this about deciding which edge to add at each step in the process. For each vertex u, Dijkstra's algorithm keeps track of the weight of the "shortest known path" to u. These values are stored in an array, L, with one location for each vertex. For a vertex v_i, that is already in the tree, $L[i]$ is the weight of the minimal weight path from the starting vertex to that v_i. For a vertex not in the tree, $L[i]$ is the weight of the shortest path that consists of edges in the tree followed by an edge that joins v_i to the tree. If no such path exists, then $L[i]$ is infinity. At each step of the algorithm, the vertex that is chosen to be added to the tree is the v_j that has the smallest value of $L[j]$ among all vertices not in the tree. When the vertex v_j is added to the tree, some of the $L[i]$ values have to be updated. Namely, for any edge $\overrightarrow{v_jv_i}$ for which v_i is not in the tree, we have to check whether the path to v_i that passes through v_j is shorter than the previously shortest known path to v_i. The length of the path through v_j to v_i is $L[j]$ plus the weight of $\overrightarrow{v_jv_i}$. If this value is less than the current value of $L[i]$, then it replaces that value.

$L[i]$ is initialized to zero for the starting vertex and to ∞ for all other vertices. At the completion of the algorithm, $L[i]$ will still be ∞ for any vertex v_i that is not reachable from the starting vertex. For any other vertex, $L[i]$ will be the weight of the minimal weight path from the starting vertex to v_i. Dijkstra's algorithm can be coded as follows:

```
bool inTree[N];   // For recording which vertices are in the tree.
double L[N];      // Best known distances from start to i.
int pred[N];      // pred[i] is the node that precedes vertex
                  //    i on the shortest path from the starting
                  //    vertex to i.  pred[i] is -1 if no path
                  //    has  yet bee found.

for (int i = 0; i < N; i++) {
   inTree[i] = false;
   L[i] = INFINITY;
   pred[i] = -1;
}
int nextVertex = v;  // where v is the starting vertex.
```

```
L[v] = 0;            // path length from v to itself.

while (nextVertex != -1) {

      // We add nextVertex to the tree and update
      // any values of L[i] for which the path from
      // nextVertex to i is shorter than the previous
      // best value for i.  We assume that weight[i][j]
      // is the weight of the edge from vertex i to
      // vertex j if such an edge exists, or is
      // INFINITY if there is no edge from i to j.
      // (That is, weight[N][N] is the adjacency matrix
      // for the weighted graph.)

   inTree[nextVertex] = true;
   for each edge from nextVertex to a vertex w { // Pseudocode!
      if ( ! inTree[w] )
         if ( L[nextVertex] + weight[nextVertex][w] < L[i] ) {
            L[i] = L[nextVertex] + weight[nextVertex][w];
            pred[i] = nextVertex;  // Predecessor of i on
         }                         //   the new shortest path.
   }

      // Now find the next vertex to be added.  We want
      // the vertex that has the smallest value of L[i]
      // among the vertices that are not in the tree.
      // If all vertices are in the tree, we are done.
      // Also, if all vertices not in the tree have
      // L[i] = INFINITY, then none of the remaining
      // vertices are reachable from the starting vertex,
      // and we are done in this case also.

   nextVertex = -1;
   double min = INFINITY;
   for (int i = 0; i < N; i++) {
      if ( ! inTree[i]  &&  L[i] < min ) {
         nextVertex = i;
         min = L[i];
      }
   }
   // We are done if nextVertex is still equal to -1.
}
```

The array *pred* is an optional feature of this algorithm. It can be used to recover the actual minimal weight path from the starting vertex to any other vertex. If $L[i]$ is not ∞ at the completion of the algorithm, then $pred[i]$ is the index of the vertex that precedes v_i on the shortest path from the starting vertex to v_i. We can find the complete path to v_i by tracing these predecessor links backwards from each vertex to its preceding vertex, all the way back to the starting vertex.

The while loop in Dijkstra's algorithm is executed at most once for each vertex in the graph, and the body of the first for loop is executed at most once for each edge in the graph. Assuming that we are using an edge-list representation for the graph, this for loop contributes $\mathcal{O}(|E|)$ to the run time of the algorithm The second for loop, however, takes up $\Theta(|V|)$ time each time it is run, and it is run $\mathcal{O}(|V|)$ times, so the total run time for this for loop is $\mathcal{O}(|V|^2)$. Since $|E|$ is also $\mathcal{O}(|V|^2)$, the running time for this algorithm is $\mathcal{O}(|V|^2)$. It is $\Theta(|V|^2)$ in the worst case, which occurs when every vertex in the graph is reachable from the starting vertex, since the while loop runs $|V|$ times in this case.

(It is possible to improve the worst case running time to $\Theta(|E| * \log(|E|))$ by using a better method for locating the minimum value of $L[i]$. The improved method stores the vertices in a data structure that is a variation on a priority queue. The priority of a vertex v_i is given by $L[i]$, and we order the priority queue so that the item with the smallest priority will be the first one removed. However, this version of the priority queue must support an operation that was not covered in Chapter 3: We must be able to decrease the priority of an item that is already in the queue. This capability is used when the value of $L[i]$ is adjusted in the algorithm. This *decrease_priority* operation can be implemented in $\Theta(\log(n))$ time for a priority queue that contains n items.)