Graph Algorithms

1 Preliminary Definitions

There is a lot of terminology associated with graphs and graph algorithms, and so we start by defining the terms we will use for the remainder of this chapter.

Definition 1. A ***graph*** $G = (V, E)$ is a set of ***vertices*** V and a set of ***edges*** E. Each edge $e \in E$ is a pair (v, w) where $v \in V$ and $w \in V$. Formally,$E \subseteq V \times V$, the Cartesian product of V with itself. Edges are also called ***arcs***.

Definition 2. A graph $G = (V, E)$ is a ***directed graph*** (a ***digraph***, for short) if E is a set of ordered pairs, in which case the edges are called ***directed edges***. A graph that is not directed is called an ***undirected graph***.

Every graph is therefore either directed or undirected. If it is not specified, i.e., if you see the word *graph* without any modifier, it is usually in a context in which it does not matter whether it is directed or undirected. Sometimes, sources might use the word *graph* without a modifier to mean an undirected graph. In these notes, the word *graph* will mean *any graph.*

In a digraph, if (v, w) is an edge then w is said to be ***adjacent*** to v, but not vice versa. In an undirected graph, if (v, w) is an edge then v is adjacent to w and w is adjacent to v. The directed edge (v, w) is said to be ***incident*** on the vertex w, and the undirected edge (v, w) is incident on both v and w.

Definition 3. A ***weighted graph*** is a graph (directed or undirected), in which edges have ***weights***, also called ***costs***. A weight is a numeric value assigned to an edge.

You can think of a weighted graph as having a weight, or cost, function that assigns an integer to each edge. Weighted graphs are very common because they model many real world relationships. For the remainder of this chapter, I will assume that if a graph is weighted, then there is a function $c : E \to \mathbb{Z}$ that assigns an integer, positive, negative, or zero, to each edge.

Example 4. Let $G = (V, E)$ be the graph in which V is the set of all American cities with airports, and E is the set of all edges (v, w) such that there is a scheduled flight from city v to city w by some commercial airline. The weight $c(v, w)$ assigned to the edge (v, w) might be

1. the air distance between v and w,
2. the air time between them, or
3. the price of a minimum price ticket.

Online reservation systems often let the user choose from among many criteria for weighting flights.

Example 5. Let $G = (V, E)$ be a computer network and let V be the set of routers and endpoint computers. E is the set of connections between routers and routers and between routers and computers. Weights might be the latency of a transmission across an edge, or the bandwidth of an edge. It might even be the length of the physical wire.

Definition 6. A ***path*** in a graph is a sequence of vertices, $v_1, v_2, v_3, \ldots, v_N$ such that for each i, $1 \leq i < N$, $(v_i, v_{i+1}) \in E$. The ***length*** of a path is the number of edges (not vertices) in the path, which is $N - 1$ if there are N vertices in the path.

A path of length 0 is a path from a vertex to itself with no edges, so it is just a single vertex. In contrast, (v, v) is a path of length 1, which is different from a path of length 0. A path (v, v) is called a ***loop***.

Definition 7. A ***simple path*** is a path such that all vertices are distinct except possible the first and the last.

Definition 8. A ***cycle*** in a directed graph is a path of length at least 1 such that the first and last vertices are the same. A cycle is simple if it is a simple path.

Definition 9. A ***cycle*** in an undirected graph is a path in which the first and last vertices are the same and the edges are distinct (to avoid the problem that a path like u, v, u is called a cycle even though the second edge is the same as the first edge.)

Definition 10. A graph is ***acyclic*** if it has no cycles. A directed acyclic graph is called a ***DAG*** for short.

Definition 11. An undirected graph is called ***connected*** if there is a path from every vertex to every other vertex. A directed graph with this property is called ***strongly connected***.

2 Graph Representations

There are two common representations.

2.1 Adjacency Matrix

A boolean or numeric-valued square matrix `A` with a row for every vertex. `A[u][v]` is true if there is an edge from `u` to `v`. If `A` is numeric valued, then `A[u][v]` would be the weight of the edge *(u,v)*. The following asymmetric boolean matrix represents the directed graph in Figure 1.

	0	1	2	3	4	5
0	0	0	0	1	0	0
1	0	0	1	0	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	1
4	1	0	0	0	0	0
5	0	1	0	0	0	0

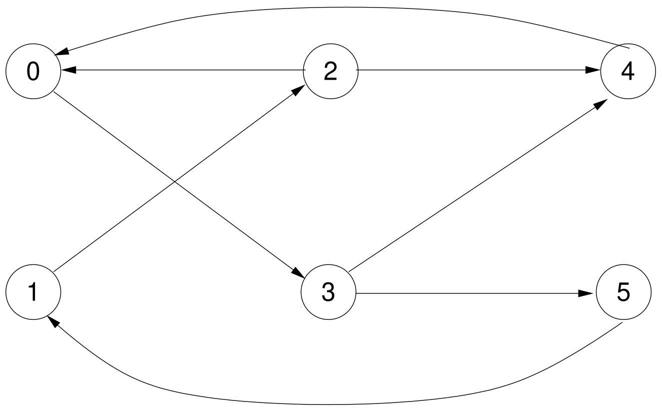


Figure 1: A directed graph.

Advantages

An adjacency matrix has a very simple implementation. That is about its only advantage.

Disadvantages

- Graphs are usually sparse, meaning that they have lots of zeros. There are N^2 entries in a matrix of N vertices, but the number of edges may only be a multiple of N.
- Many problems that have efficient solutions cannot be solved efficiently when this representation is used, because they will all tend to be $O(N^2)$.

2.2 Adjacency Lists

An adjacency list is a linear array with an entry for each vertex, such that each entry is a pointer to a list of vertices adjacent to that vertex. This is the more common representation because it is the most efficient for most purposes. Formally,

Definition 12. An ***adjacency list*** for a graph *G = (V,E)* is an array `A` of size $n = |V|$ such that, for each vertex $v_i \in V$, `A[i]` is a pointer to a linked list of nodes $(v_1, v_2, \ldots, v_k)$ such that (`A[i]`,v_j) is an edge in E for each $j = 1, 2, \ldots, k$.

If the graph is a weighted graph, then the node for each edge in the list would have a member variable that stores the weight.

Vertices often have labels or names, such as the name of the cities they represent. In this case the properties are stored in the array entry along with the pointer to the list. The graph in Figure 1 would have the following adjacency list:

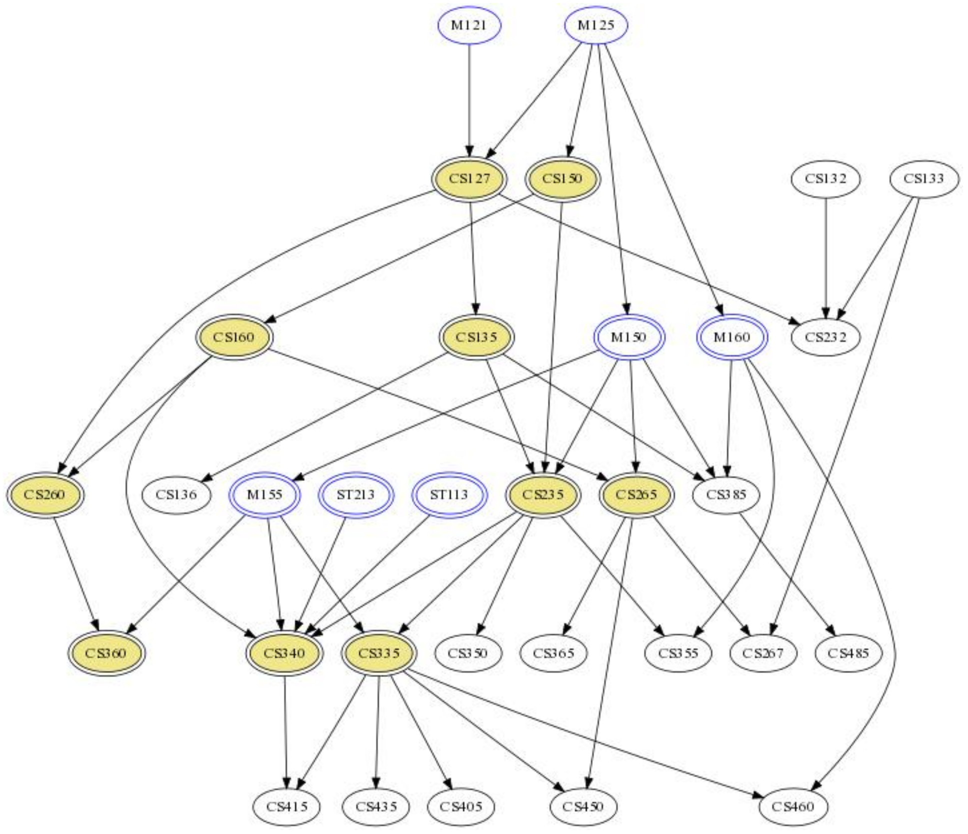
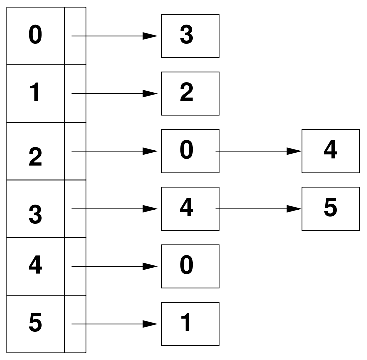


Figure 2: Prerequisites for CS Major in Hunter College



3 Graph Algorithms: Warming Up

We will examine a few of the simplest algorithms for solving problems related to graphs. We start with topological sorting.

3.1 Topological Sort of a Directed Acyclic Graph

Figure 2 is a snapshot of what the prerequisites for computer science courses were in Hunter College some years back. Most students will immediately realize that the set of courses required for the computer science major together with the prerequisite relation is a directed, and hopefully acyclic, graph. That is, if c_1 and c_2 are nodes in a graph that represent two difference courses, and there is a directed edge from $c_1$to c_2, then it means that c_1 is a prerequisite for c_2, and that therefore one cannot take c_2 until one has successfully completed c_1. So for example, according to Figure 2, there is a directed path from CS127 to CS415 of length 4, which means that after completing CS127, one needs at least 4 semesters to complete CS415.

Suppose that you are a very busy person and that you can take just one course in each semester. You seek to assign the numbers $1, 2, \ldots, N$ to the N courses you need to take to graduate with a major in Computer Science with the meaning that the course labeled k is to be taken in semester k. This is an example of a topological sort of the graph of courses. It is a *linear*, or *total*, ordering of the nodes of the graph such that, if there is a directed path from a node v to a node w in the graph, then v has a lower number than w in the ordering.

It should be pretty obvious that there are many possible topological sorts of the graph in Figure 2. When two or more nodes in the graph are not related, meaning that there is no directed path from one to the other, then their relative order in the topological sort does not matter and multiple topological sorts of the graph are possible. Any valid topological sort is a solution to the problem. In general, the edge set E in a graph defines a ***partial ordering*** relation on the vertices; it is not total. A topological sort imposes a ***total ordering*** that is consistent with the underlying partial ordering that the edges define.

To make the problem precise now, a topological sort of a graph $G = (V, E)$ with vertices $\{v_1, v_2, v_3, \ldots, v_N\}$ is an ordering of the vertices such that if there is a path from v_i to v_j in the graph, then v_i must precede v_j in the ordering. The output of a topological sort of a graph $G = (V, E)$ is an assignment of the numbers $1, 2, \ldots, N$ to the vertices.

A topological sort is not possible for graphs with cycles. If a graph has cycles, then for any two vertices v and w in a cycle,v precedes w and w precedes v. There is no way to create a sequence because if v precedes w, then w cannot be before v, and vice versa.

The Algorithm

Define the ***in-degree*** of a node to be the number of edges incident upon the node. For example, in Figure 2, M121 has in-degree 0 and CS235 has in-degree 3. We now prove

Lemma 13. *If a graph has no cycles, then there is at least one node with in-degree 0.*

Proof. Suppose to the contrary that every node has in-degree greater than zero. Pick any node arbitrarily. Pick an edge incident on that node and visit the node at its source. Repeat this procedure of following the edge incident on the node in the reverse direction of the edge. If multiple edges are incident on a node, pick any one arbitrarily. Since every node has at least one such incoming edge, the path so followed is of infinite length, as there is always an edge to follow. But there are finitely many nodes in the graph, so some node must appear twice on this path, which implies that the path contains a cycle. □

Let v be any one of the nodes whose in-degree is zero. We assign v the number 1 and reduce the in-degree of all vertices adjacent to v by 1. Reducing the in-degree of an adjacent node by 1 is like removing the edge from v to the node, so if you were simulating this algorithm on paper with a picture of the graph, it would be best to do so using a pencil with an eraser! Next, we try to find another node whose in-degree is 0 and if we do, we repeat this procedure. If we cannot find a vertex with in-degree 0 but we have not visited every vertex, then there must be a cycle, because it means that every node in the remaining non-empty graph has at least one incoming edge, which means that the remaining graph has a cycle, by Lemma 13 above. The following listing is a pseudocode description of the resulting algorithm.

```
initialize the in-degrees of all vertices by scanning the
   adjacency lists

counter = 1;
while there are vertices in V yet to be processed
{
    let v be any vertex with in-degree == 0;
    if no such vertex exists,
        return CycleFound
    else {
        v.topologicalNumber = counter++;
        for each w that is adjacent to v,
            decrement its in-degree by 1
        }
}
```

Correctness We want to prove that if there is a path from v to w in the graph, then v is assigned a smaller number than w by the algorithm.

We can argue by induction on the length of a path from v to w. If the path has length 1, then (v, w) is an edge in the graph and w cannot be visited before v because the edge (v, w) will not be removed until v is visited, so *in-degree(w)* will not be 0 until after v is visited. This implies that the number assigned to w is greater than the number assigned to v.

Suppose that we have proved the claim for all paths of length less than n and let v, w be nodes such that the length of the path from v to w is $n > 1$. By assumption there is a vertex u such that $v, \ldots, u$ is a path of length $n - 1$ and (u, w) is an edge. By the hypothesis, the topological number assigned to v is less than the topological number assigned to u, and by the previous argument, the number assigned to u is less than that assigned to w, proving the claim for paths of length n.

Implementation and Performance Analysis The simplest way to implement this algorithm is to use a queue for storing the vertices whose in-degrees are zero. The in-degree of each vertex is computed by reading every adjacency list and incrementing the in-degree of vertex v every time that v appears in the adjacency list of some other vertex. This is *O(E)* since it looks at each edge once.

Having computed all in-degrees, all vertices with in-degree 0 are enqueued and a loop is entered. In the loop, the front of the queue is removed, it is given a topological number, its successors' in-degrees decremented, and if 0, they are enqueued. The resulting pseudocode follows.

```
// Find all nodes with in-degree 0
q.makeEmpty();
for each vertex v in V
    if ( v.indegree == 0 )
```

```
        q.enqueue(v);

// Start processing with a count of 1
counter = 1;
while ( ! q.empty() )
{
    v = q.dequeue();    // v has in-degree 0
    v.topologicalNumber = counter++;

    for each w that is adjacent to v {
        // (v,w) is an edge in the graph
        w.indegree--;
        if ( w.indegree == 0 )
            q.enqueue(w);
    }
}
if ( counter != |V| )
    return CycleFound; // or throw an exception
```

This algorithm requires $O(|E|+|V|)$ time. This is because, in the while-loop, each edge is examined at most once and each vertex is processed at most once. Whichever is larger is what dominates the running time.

Example

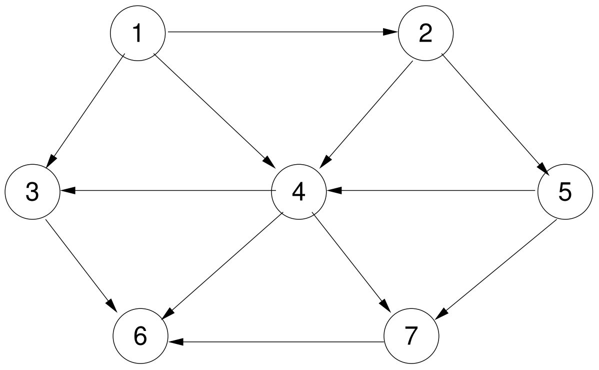
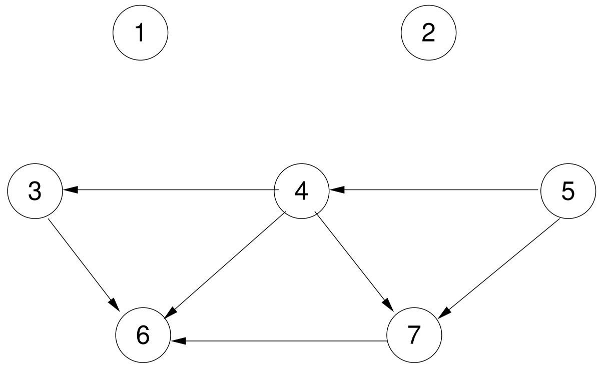


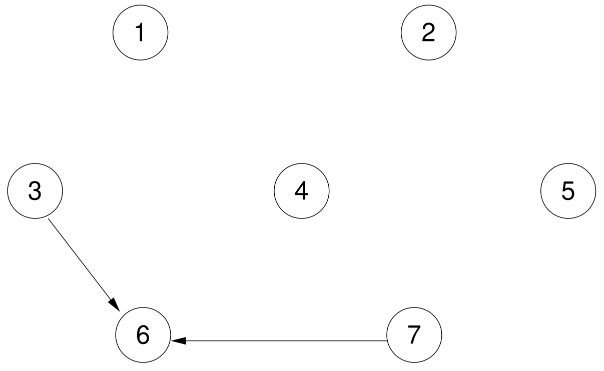
Figure 3: Digraph to be topologically sorted.

Consider the graph in Figure 3. Two topological orderings of it are 1, 2, 5, 4, 7, 3, 6 and 1, 2, 5, 4, 3, 7, 6. Let us apply the algorithm to it. The in-degrees of all nodes are computed and we see that node 1 has in-degree 0.

We assign a 1 to node 1, and remove edges (1,2), (1,3), and (1,4). Node 2 now has in-degree 0, so we assign 2 to it and remove the edges (2,4) and (2,5). The resulting graph is



Node 5 now has in-degree 0 so it is assigned 3, and the edges (5,4) and (5,7) are removed. This results in node 4 having in-degree 0, so it is given number 4 and edges (4,3), (4,6), and (4,7) are removed. The graph is now



The last few steps assign 3 the number 5, deleting edge (3,6), then assigning number 6 to node 7, deleting (7,6), and assigning 7 to node 6.

3.2 Minimum Spanning Trees

Definition 14. A ***spanning tree*** for an undirected graph[1] $G = (V, E)$ is a graph G' $= (V, E$') such that G' is a tree.

In other words, G' has the same set of vertices as G, but edges have been removed from E so that the resulting graph is a tree. This amounts to saying that G' is acyclic. It is called a spanning tree because every node is connected to every other node, so the tree spans all of the nodes. Removal of any single edge from a spanning tree causes the graph to be unconnected. A spanning tree is ***minimum*** if there is no other spanning tree with smaller cost. If the graph is unweighted, then the cost is just the number of edges. If it has weighted edges, then the cost is the sum of the edge weights of the edges in the spanning tree. See Figure 4 for an example.

An example of an application of spanning trees is for finding the least length of wiring that can be used to wire the electrical connections in a building.

[1]For directed graphs the problem is still defined but we will not consider them.

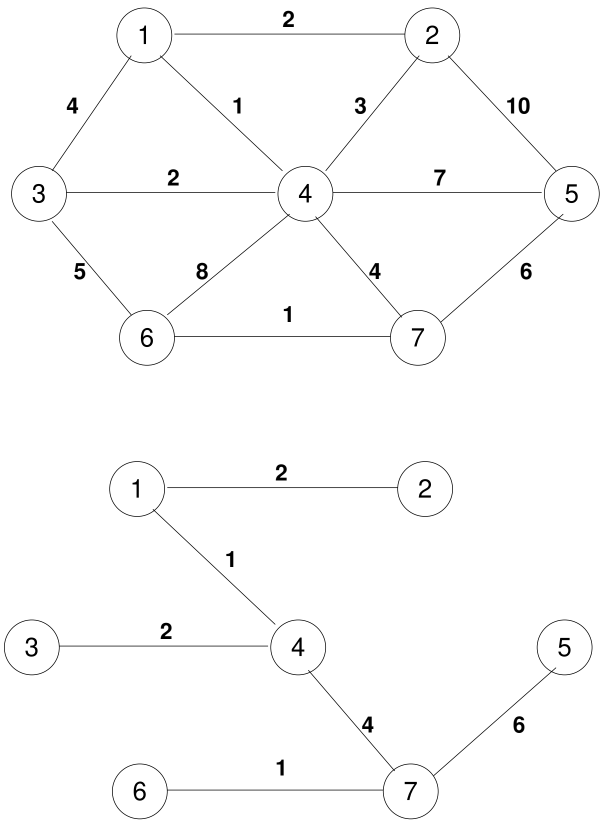


Figure 4: An undirected graph and a minimum spanning tree for it.

The problem of finding a minimum spanning tree amounts to finding the set of edges of which this tree is composed. After all, the spanning tree is the same vertex set as the graph, but a subset of the edges. So the idea is to design an algorithm that selects which edges belong in the tree. A relatively simple algorithm for finding a minimum spanning tree is ***Kruskal's Algorithm***. It is a greedy algorithm in that it always tries to find the best solution at each step. Not all greedy approaches work. Here it does. It uses the *Union-Find* algorithm from Chapter 8. Its "greediness" is in always choosing an edge with least weight among the possible choices of edge.

Kruskal's Algorithm

The algorithm starts out with a forest in which each vertex is a tree by itself. Then it looks for the edge with least weight and connects its two vertices into a tree with two vertices. If there is more than one such edge, any of them can be chosen. It then looks for another edge that is least weight among those that remain and connects the two vertices together if they are not already part of the same tree. If they are in the same tree, then adding this edge would form a cycle, so the edge is not added. The algorithm continues this procedure until there is just one tree, i.e., every vertex is in a single tree. This tree will be minimum cost because the edges were added in order of ascending cost.

The algorithm represents the trees as sets of vertices. To form a new tree from two existing trees, a *union* operation is applied to the two sets that represent these trees. When given an edge *(v,w)* in the graph, the algorithm uses the *find* operation to find in which sets v and w each belong.

Therefore, a disjoint set ADT is used to represent the sets of vertices. A heap is used for picking the minimum weight edge at each step. The algorithm is presented in pseudocode in the following listing.

```
1
2  void kruskal()
3  {
4      Vertex u,v;                    // Vertex is the type of vertex labels
5      Edge e;                        // Edge is a struct containing (
           source_vertex, target_vertex, weight)
6      DisjointSet s(NUM_VERTICES );  // the set of trees
7      PriorityQueue<Edge> h( NUM_EDGES );  // the edges, with smallest
           weight having highest priority
8      SetType uset, vset;            // SetType is the range of indices 0..
           NUM_VERTICES
9      list<Edge> edgelist;           // list of edges in the minimum
           spanning tree
10
11     readGraphIntoHeapArray(h);     // read the edge set into the array
           that gets heapified
12     h.buildHeap();                 // now heapify based on edge weights
13
14     int edgesAccepted = 0;
15     while ( edgesAccepted < NUM_VERTICES -1 ) {
16         e = h.deleteMin( );   // assume e = (u,v)
17         uset = s.find(u);
18         vset = s.find(v);
19         if ( uset != vset ) {
20             // add the edge to the list of edges in the tree and increment
                   count
21             edgelist.pushback(e);
22             edgesAccepted++;
23             s.union(uset, vset);
24         }
25     }
26 }
```

Performance Analysis The algorithm takes $O(|E|)$ steps to build the heap in lines 11 and 12. Since the while-loop iterates in the worst case once for each edge in the graph, it can take $O(|E|)$ deleteMin, find, and union operations to build the minimum spanning tree. Each deleteMin takes $\log|E|$ steps, so this is $O(|E|\log|E|)$ steps. The find and union operations take less time than the deleteMin, so they do not increase the amortized running time. Since $|E| = O(|V^2|)$ and $\log(|V|^2) \in O(\log|V|)$, this is $O(|E|\log|V|)$.

4 Shortest Path Algorithms

We assume that the graphs in question are directed graphs. There are a few different types of shortest path problems. The simplest one is the ***single-source shortest path*** problem. The most general statement of the problem is the ***shortest weighted path*** problem. This is one of the hardest versions of the problem. A weighted path is a path in a weighted graph. The weight of

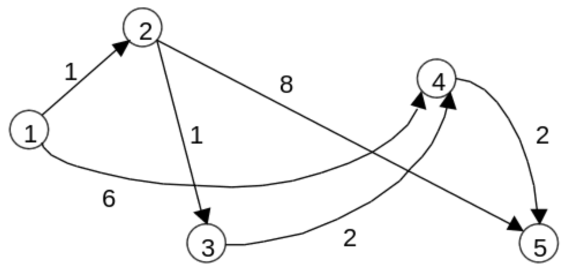


Figure 5: Sample graph

a path is the sum of the weights of the edges on the path. A ***weighted path*** from a vertex s to a vertex v is a shortest weighted path if there is no other path in the graph from s to v that has shorter weight. For convenience, when we say a ***shortest path*** we mean a shortest weighted path, not a path with the fewest edges. The distance between two vertices is the weight of a shortest path between the vertices.

4.1 Single Source Shortest Path Problem

> Given a weighted graph $G = (V, E)$, and a distinguished vertex s, find the shortest weighted path between s and every other vertex in the graph.

If we let the weight of every edge be 1, then this statement of the problem is reduced to finding the paths whose lengths are least. Edgar Dijkstra proposed an algorithm to solve the weighted graph version of this problem provided that edges do not have negative edge weights. In his algorithm, it does not matter whether the graph is directed or undirected. The output of the algorithm is a list of the shortest (weighted) distances to each vertex. If the output needs to include the set of paths to each vertex, then the algorithm can be modified to record this information.

The idea of the algorithm is to maintain a temporary set of vertices, T, with the property that the shortest path from s to every vertex in T has been correctly determined, and to enlarge this set iteratively. This set can be thought of as the *known set*, because for any vertex in this set, the shortest path from s to that vertex is *known to be correct*. Initially, only the source vertex s will be in T, since the distance from s to s is 0. In each iteration, a new vertex is added to T. When the size of the known set is equal to $|V|$, the algorithm stops.

The algorithm also maintains, for each vertex v not in T, a ***temporary least distance***, $d(v)$, from s. The value $d(v)$ is the weight of the shortest path from s to v that, except for v, passes only through vertices in T. The algorithm may discover as it proceeds that $d(v)$ is too large, and it will reduce it when that happens. We call $d(v)$ an ***estimated distance***.

After initializing the set T and recording the initial estimated distances $d(v)$ to each node, the algorithm enters a loop. In each iteration of the loop, a vertex $v \in V - T$ with minimal $d(v)$ is added to T, and the distances $d(u)$ to each $u \in V - T$ are updated. Since the set T starts with just the vertex s in it, and in each iteration, a vertex that is not in T is added to it, the algorithm must iterate exactly $|V - 1|$ times, and after $|V - 1|$ iterations of the loop, all vertices have been added to T and the algorithm terminates.

In the description of the algorithm that follows, we assume first that the cost of each edge is denoted by a cost function $c(v, w)$, which is defined on each edge (v, w) in the edge set E. This function needs to be extended to a function $c'(v, w)$ that is defined on all pairs of vertices $v, w \in V$ as follows, even if there is no edge (v, w) in E. This function is defined as follows:

$$c'(v,w) \quad = \begin{cases} 0 & if\ v = w \\ c(v,w) & if\ v \neq w \wedge (v,w) \in E \\ \infty & if\ v \neq w \wedge (v,w) \notin E \end{cases}$$

In other words, the cost of an edge (v, w) is infinite if the edge does not exist. In an actual implementation, a special value could denote when an edge does not exist. The cost of all other edges is simply the weight of the edge itself.

Algorithm

The following listing is a pseudocode description of the algorithm.

```
// Initialize the function d(v) by setting d(v) to the cost of
    the edge from s to v and setting d(s) to 0:

for each v in V {
    d(v) = c'(s,v);      // set initial distances
}
// Implies that d(s) has been set to 0

// Initialize the set T to contain only the vertex s
T = {s};

// Iterate until every vertex from V has been added to T
while ( T != V ) {
    choose a vertex v in V - T with least d(v);
    set T = T  + {v};                          // add v to T
    // update the distances from s to each vertex not yet placed
        into T. The only vertices whose distance might change are
        the ones that are adjacent to v.
    for all vertices u in V - T that are adjacent to v {
        // if the current distance to u is larger than the
            distance from s to v and then from v to u, then
            replace d(u) by the  new, smaller distance.
        if ( d(u) > d(v) + c'(v,u) )
            d(u) = d(v) + c'(v,u);
    }
}
```

When we find the vertex v whose $d(v)$ is minimal among all vertices not yet in T, we look at all vertices adjacent to it, and for each one u, if the weight of the path from s to u going through v is less than it was without going through v, we decrease the weight of its potential shortest path so that it is the weight of the path from s to u through v.

Example

Given the graph in Figure 6, the table below shows how the set T and the values $d(v)$ change in each iteration when $s = 0$. The columns show the state at the end of the iteration, not before.

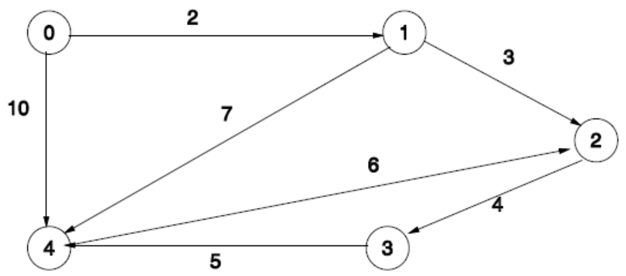


Figure 6: Example for Dijkstra's Algorithm

Iteration	T	v	d(v)	d(0)	d(1)	d(2)	d(3)	d(4)
initial	{0}			0	2	∞	∞	10
1	{0,1}	1	2	0	2	5	∞	9
2	{0,1,2}	2	5	0	2	5	9	9
3	{0,1,2,3}	3	9	0	2	5	9	9
4	{0,1,2,3,4}	4	9	0	2	5	9	9

Implementation Issues

The graph should be represented by an adjacency list, because within the loop, we have to find the set of nodes adjacent to the chosen vertex v. An adjacency list representation makes it possible to visit the nodes that are adjacent to v in constant time. I.e., if there are m nodes adjacent to v, it will take m steps to visit them.

Most graphs are sparse – the number of edges is roughly $|V|$. For these graphs, it makes sense to use a priority queue to store the vertices in $V - T$, and to use a *deleteMin* operation to pick the vertex from $V - T$ whose distance $d(v)$ is smallest. We do not need to create an actual set T; we can maintain a boolean array with an entry for each vertex in V; the entry would be false if v is not in T and true if it is in T. To decide whether a vertex u that is adjacent to v is in the set T, we just inspect its boolean value in this array.

The hard part is efficiently updating the function $d(v)$ for all vertices not in T. Suppose that we use a binary heap for the priority queue. If we assume that the adjacency list entry for a vertex stores the index of the vertex in the binary heap, or –1 if it is not in the heap, then each time that we need to decrease the value of $d(u)$ for a vertex u that is adjacent to v and in the heap, we look up its index in the heap from the adjacency list, e.g.

```
k = A[u].heapindex;
```

and then modify the value of $d(u)$ in the heap with something like

```
heap[k].distance = d(v) + c'(v,u);
```

where the right hand side is the distance from s to v plus the weight of the edge from v to u, obtained from the adjacency list. Now this is not enough, as the heap element has changed value and must be percolated up. So we follow the update with a *percolateUp* operation.

There is an alternative strategy that uses less time but more storage and is also more complicated:

when a vertex u has to be updated, instead of actually finding it in the heap, changing its value, and percolating it up, we just create a new instance of it with smaller key value and insert it into the heap, where it will rise above its current position. Of course now there would be multiple copies of it in the heap, but it doesn't matter if we keep track of whether it has been moved into the set T – we just ignore the ghost copies whose bodies have moved into the set T. The smaller ones will always reach the top before the larger ones. The downside of this strategy is that the heap can get big, $O(|E|)$, to be precise. The details are left to the reader.

Correctness

The following proof is an example of a proof by induction in which one needs to strengthen the inductive hypothesis in order to prove it. Sometimes the obvious induction hypothesis is some assertion $P(n)$ that needs to be proved for all $n \geq 0$, but in order to do so, we need to add constraints to P to climb up the inductive step, so to speak. In other words, we supplement $P(n)$ with an assertion $Q(n)$ so that we have to prove $P(0) \wedge Q(0)$, and if $P(n) \wedge Q(n)$ then $P(n+1) \wedge Q(n+1)$. In this case the "n" will be the size of the set T, i.e., $|T| = n$. The hypothesis to be proved is two-fold:

1. for every vertex v in the set T, $d(v)$ is the weight of a shortest path from s to v, and
2. for every vertex w in the set $V - T$, $d(w)$ is the weight of a shortest path from s to w that lies completely in T, except for w itself.

The first part states that whenever a vertex has been placed into T, $d(v)$ is the weight of the absolute shortest path from s to v, regardless of the route it takes. The second part states that, for those vertices w not yet placed into T, $d(w)$ is the weight of the shortest path from s to w that does not visit any vertex outside of T except for w. It is not necessarily the weight of a shortest path from s to w.

Base case. $|T| = 1$. Since s is the only vertex in T and its distance to itself is 0, and since $d(s)$ is initialized to 0, part 1 is true. For all w in $V - \{s\}$, $d(w)$ is initialized in lines 4-6 to be the weight of the edge from s to w if it exists, or infinity. The edge (s, w) must be the shortest path from s to w, because there are no negative edges in the graph and so a path to w that passed through another vertex would not have less weight. Therefore, part 2 is true.

Inductive step. Assume that both parts of the hypothesis are true when $|T| = k$ and that v is the $k+1^{st}$ vertex added to T in line 14. The algorithm chooses v to be the vertex in $V - T$ whose estimated distance from s, $d(v)$, is least among all vertexes $w \notin T$.

Suppose that $d(v)$ is not the weight of a shortest path from s to v. Before we added v to T, the inductive hypothesis was true for T, so part 2 of the hypothesis tells us that $d(v)$ was the weight of a shortest path to v that lies completely in T except for v, so if $d(v)$ is not now the weight of an absolute shortest path, then there is a shorter path from s to v that is not completely contained in T except for v. Let p be such a shorter path. Let $D(p)$ be the total weight of this path p. Then $D(p) < d(v)$ by our assumption.

This path p contains a vertex that is not in T, so let w be the first vertex in the path from s to v that is not in T. In other words, we can write $p = (s, \ldots, w, \ldots, v)$ where all vertices in p preceding w are in T. The sub-path of p from s to w lies completely within T except for w itself. By the inductive hypothesis, part 2, $d(w)$ must be the weight of a shortest path from s to w that lies completely within T. Clearly $d(w) \leq D(p)$ because the path from s to w is a sub-path of the path from s to v, and the remaining edges have non-negative weights. Then $d(w) \leq D(p) < d(v)$, which contradicts the fact that v was chosen to be the vertex in $V - T$ whose estimated distance $d(v)$ was least. Therefore the path p does not exist, and $d(v)$ is the weight of the absolute shortest path from s to v. Vertex v is added to T in line 15 and so becomes part of T when $|T| = k + 1$.

To complete the inductive step we need to show that the two parts of the hypothesis remain true after v is added to T and after lines 16 to 19 are executed. We assumed part 1 was true before we added v and the only new vertex in T is v, and we just proved that when it was added to T, $d(v)$ was the weight of the shortest path from s to v in the graph. So part 1 remains true when $|T| = k + 1$. What remains is to show that part 2 is still true. Suppose it is no longer true. Then there is a vertex $w \in V - T$ for which $d(w)$ is no longer the weight of the shortest path from s to w that lies completely in T, except for w itself. Either this vertex is adjacent to v or it is not.

Suppose w is adjacent to v. By the hypothesis, $d(w)$ was the weight of a shortest path from s to w that was completely within T before v was added to T. If $d(w)$ is no longer the shortest distance from s to w, then there must be a shorter path to w within T, because $d(w)$ is either the same or it is smaller. Since the only change to T is that v is now part of it, the only way that there can be a shorter path to w is if it passes through v. But in this case, $d(w)$ is assigned its new weight in lines 18 to 19 and the hypothesis holds true. If $d(w)$ is still the shortest distance, then no update is made and it stays the same. In either case, $d(w)$ is the weight of a shortest path from s to w completely within T except for w. So the hypothesis remains true if w is adjacent to v.

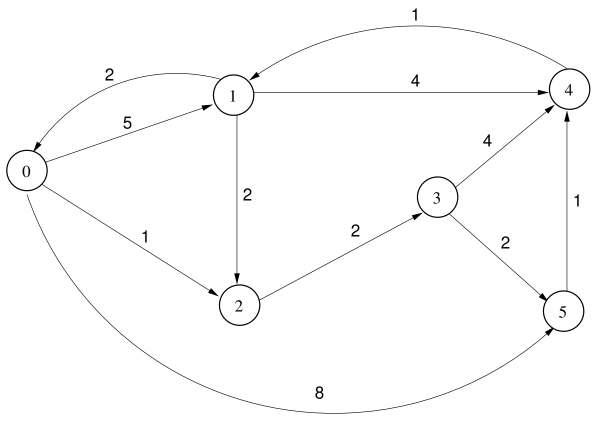
Suppose the vertex w is not adjacent to v. Then adding v to T did not change the set of paths from s to w; there is no shorter path to w from s passing through only vertices in T. The algorithm does not change $d(w)$ in this case. Therefore, $d(w)$ remains the weight of the shortest path from s to w that lies completely in T except for w. This shows that the inductive hypothesis remains true when the size of T has increased by 1, which proves that the algorithm is correct, because it stops when $T = V$ and $V - T$ is empty. QED.

Performance Analysis

The running time of the algorithm greatly depends on how it is implemented. If the vertices are stored in a priority queue based on the values $d(v)$, then choosing the vertex v with least $d(v)$, combined with removing it from $V - T$ is a *deleteMin* operation requiring $O(\log |V|)$ steps. The time to update the values $d(u)$ for each u adjacent to v is also $O(\log |V|)$, assuming that the update is performed using the implementation described above (the update can be accomplished by doing a `percolateUp(u)` on the vertex in the heap, which is at worst $O(\log |V|)$.) The number of updates is in the worst case $|E|$, because each edge is examined only once. Therefore, the total cost of the updates is $O(|E| \log |V|)$. The total number of *deleteMin* operations is $|V|$ because the loop executes $O(|V|)$ times, so the total cost of the *deleteMins* is $O(|V| \log |V|)$. The total running time is therefore $O(|E| \log |V| + |V| \log |V|)$.

4.2 The All-Pairs Shortest Path Problem

Sometimes we want to compute the shortest paths between all pairs of vertices. One example of this is in producing a chart that contains the minimum distances or costs of traveling between any pair of cities in a given geographic region, as in Example 4. The weights associated with edges might be travel time or the cost of an airline ticket. For this reason the pairs are ordered pairs, since these costs are not symmetric, and the collection of cities with inter-city distances will be represented by a weighted, directed graph.



$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 5 & 1 & \infty & \infty & 8 \\
1 & 2 & 0 & 2 & \infty & 4 & \infty \\
2 & \infty & \infty & 0 & 2 & \infty & \infty \\
3 & \infty & \infty & \infty & 0 & 4 & 2 \\
4 & \infty & 1 & \infty & \infty & 0 & \infty \\
5 & \infty & \infty & \infty & \infty & 1 & 0
\end{array}
$$

Figure 7: A weighted, directed graph and its adjacency matrix representation.

Figure 7 contains a weighted, directed graph. It shows that the distance from vertex 0 to vertex 1 is 5, for example, and that the direct distance from vertex 3 to vertex 4 is 4. But notice that the distance from vertex 3 to vertex 4 obtained by traveling through vertex 5 is $1 + 2 = 3$, illustrating that the shortest distance between two vertices is not necessarily the one along the least number of edges. The ***all-pairs, shortest-path problem*** is, given a graph G, to find the lengths of the shortest paths between every pair of vertices in the graph.

The first step in designing an algorithm to solve this problem is deciding whuch representation to use for the graph. For this particular problem, the best representation is an adjacency matrix, which we denote A. If the graph has an edge (i, j), then $A_{i,j}$ will be the weight of the edge. If there is no edge, conceptually we want to assign ∞ to the entry $A_{i,j}$. When we do arithmetic, we would use the rule that $\infty + c = \infty$ for all constants c. In Figure 7, the adjacency matrix has ∞ in all entries that correspond to non-existent edges. In an actual application, we would have to use a very large number instead of ∞.

The adjacency matrix representation uses an amount of storage proportional to $O(|V|^2)$. Other graph representations use storage that is asymptotically smaller. When the graph is sparse, meaning that the number of edges is small in comparison to n^2, this representation may not be suitable. But for this particular problem, the big advantage is that it is constant time access to each edge weight, and this advantage outweighs the high cost of storage. This is why we say it is the best representation in this case. A convenience of using the matrix is that we can use it to store the final, shortest distances between all pairs of vertices. In other words, when the algorithm terminates, the matrix entry A_{ij} will contain the shortest distance between vertex i and vertex j .

There are several different algorithms to solve this problem; however, we will present Floyd's algorithm. ***Floyd's algorithm*** is also known as the *Floyd-Warshall* algorithm. It was discovered

	0	1	2	3	4	5
0	0	5	1	∞	∞	8
1	2	0	2	∞	4	10
2	∞	∞	0	2	∞	∞
3	∞	∞	∞	0	4	2
4	∞	1	∞	∞	0	∞
5	∞	∞	∞	∞	1	0

Table 1: The shortest paths matrix after one iteration

independently by Robert Floyd and Stephen Warshall in 1962[2]. This algorithm has one restriction: it will not work correctly if there are negative cycles in the graph. A ***negative cycle*** is a cycle whose total edge weight is negative. We will assume there are no such cycles in our graph. After all, distances between cities cannot be negative. Floyd's algorithm runs in $\Theta(|V|^3)$ time regardless of the input, so its worst case and average case are the same. A pseudo-code description is in Listing 1 below.

Listing 1: Floyd's algorithm for all-pairs shortest paths.

```
// let A be a n by n adjacency matrix
for k = 0 to n-1
    for i = 0 to n-1
        for j = 0 to n-1
            if ( A[i,j] > A[i,k] + A[k,j] )
                 A[i,j] = A[i,k] + A[k,j];
            end if
        end for
    end for
end for
```

Initially, the length of the shortest path between every pair of vertices `i` and `j` is just the weight of the edge from `i` to `j` stored in `A[i,j]`. In each iteration of the outer loop, every pair of vertices `(i,j)` is checked to see whether there is a shorter path between them that goes through vertex `k` than is currently stored in the matrix. For example, when `k = 0`, the algorithm checks whether the path from `i` to `0` and then `0` to `j` is shorter than the edge `(i,j)`, and if so, `A[i,j]` is replaced by the length of that path. It repeats this for each successive value of k. With the graph in Figure 7, until `k = 5`, the entry `A[3,4]` has the value 4, as this is the weight of the edge `(3,4)`. But when `k = 5`, the algorithm will discover that `A[3,5] + A[5,4] < A[3,4]`, so it will replace `A[3,4]` by that sum. In the end, the lengths of the shortest paths have replaced the edge weights in the matrix. Note that this algorithm does not tell us what the shortest paths are, only what their lengths are. The result of applying Floyd's algorithm to the matrix in Figure 7 is shown in Table 2. This particular graph has the property that there is a path from every vertex to every other vertex, which is why there are no entries with infinite distance in the resulting matrix. It is an example of a strongly connected graph.

[2]Its history is even more complicated than this, as it was also discovered by Bernard Roy earlier, but for finding the transitive closure of a matrix.

	0	1	2	3	4	5
0	0	5	1	3	6	5
1	2	0	2	4	4	6
2	8	6	0	2	5	4
3	6	4	6	0	3	2
4	3	1	3	5	0	7
5	4	2	4	6	1	0

Table 2: The shortest-paths matrix for the graph in Figure 7.

5 Graph Traversals

5.1 Depth-First Search

A very natural way to traverse a graph, directed or undirected, is analogous to a pre-order traversal of a binary tree and it is called a ***depth-first search***, or ***DFS*** for short. DFS is a backtrack-style traversal that visits a node and then recursively visits all nodes adjacent to it that have not yet been visited. To prevent the algorithm from going into an endless loop by revisiting nodes, when a node is visited for the first time it is marked.

A very general algorithm that does nothing other than traverse the graph is

```
for all vertices v in V
    v.visited = false;
void Graph::DFS( Vertex v)
{
    v.visited = true;
    for each vertex w that is adjacent to v
        if ( !w.visited)
            DFS(w);
}
```

If an undirected graph is not connected, this will not visit every node. Similarly, if a directed graph is not strongly connected, it will not visit every node. The solution is to use a numbering scheme. Initially every vertex will have its number set to 0. When it is visited by DFS, it will get a non-zero number. If after processing all vertices, some still have zero as their number, they were not visited and must be in a part of the graph that was not connected to where the DFS started. DFS can be called for each vertex whose number is zero. The following algorithm illustrates this idea and also introduces the concept of ***tree edges*** and ***back edges***.

```
void Graph::DFS(Vertex v, Vertex u)
{
    // u is the parent of v in the search, i.e., (u,v) is the edge

    i += 1;
    v.number = i;
    for all w adjacent to v
        if (w.number == 0) {
            // this is a tree edge
            Tree_edges = Tree_edges + (v,w);
            DFS(w,v);
        }
        else if (w.number < v.number && w != u ) {
            // this is a back edge
            Back_edges = Back_edges + (v,w);
        }
}

// Assume that 0 represents a non-existent node
// i.e., all nodes have labels greater than 0
void Graph::DFS()
{
    i = 0;
    Tree_edge  = {};
    Back_edges = {};
    for all v in V
        v.number = 0;

    for all v in V
        if ( v.number == 0 )
            DFS(v,0);
}
```

Depth-first search essentially creates a tree whose root is the starting vertex of the search. If a graph is undirected, depth-first search creates a directed graph that is a tree. The direction of the edges is the direction in which the search proceeded. If an edge is traversed it is a tree edge. If DFS discovers an edge that cannot be traversed because it would visit a vertex that has already been visited, then that edge becomes a back edge. Back edges point from descendants to ancestors in the tree.