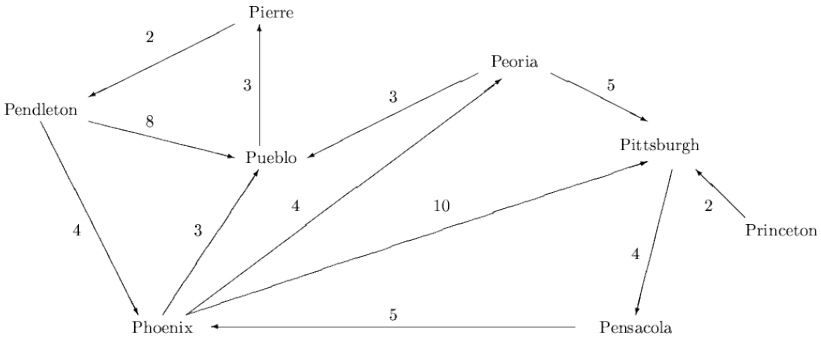
Chapter 13: Graphs and Matricies

Graphs and matrices occur in a large variety of applications in computer science. In this chapter, we provide a brief introduction to these concepts, illustrating the two most commonly used data structures for representing graphs. Example problems are then examined, which illustrate both the benefits and the limitations of each of these representations.

A graph, such as the one shown at right, is composed of *vertices* (or nodes) and *edges* (or arcs). Either may carry additional information. If a graph is being used to model real-world objects, such as a roadmap, the value of a vertex might represent the name of a city, whereas an edge could represent a road from one city to another.



Graphs come in various forms. A graph is said to be *directed* if the edges have a designed beginning and ending point; otherwise, a graph is said to be *undirected*. We will restrict our discussion in this chapter to directed graphs. A second source of variation is the distinction between a *weighted*, or labeled, graph and an *unweighted* graph. In a weighted graph each edge is given a numerical value. For example, the value might represent the distance of a highway connecting two cities, or the cost of the airfare between two airports. An unweighted graph carries no such information. The graph shown above is both directed and weighted.

1	0	0	1	0	0	0	1
0	1	0	1	0	0	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	0	1
1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	1	1	0
0	0	0	0	1	0	0	1

The two most common representations for a graph are an adjacency matrix and an edge list. An adjacency matrix represents a graph as a two-dimensional matrix. The vertices in the graph are numbered, and used as index values for the matrix. If the cities listed in the graph shown earlier are numbered in alphabetical order (Pendleton is 0, Pensacola is 1, and so on), then the resulting adjancency matrix is shown at left. Here a 1 value in position (i, j) indicates that there is a link between city i and j, and zero value indicates no such link. By convention, a node is always considered to be connected to itself.

The adjacency matrix representation has the disadvantage that it always requires $O(v^2)$ space to store a matrix with v verticies, regardless of the number of edges. An alternative representation, the edge list, stores only the edges and is thus advantageous if the graph is relatively sparse. The basic idea is that each vertex will hold a set of adjacent edges:

Pendleton: {Pueblo, Phoenix}
Pensacola: {Phoenix}

Peoria: {Pueblo, Pittsburgh}
Phoenix: {Pueblo, Peoria, Pittsburgh}
Pierre: {Pendleton}
Pittsburgh: {Pensacola}
Princeton: {Pittsburgh}
Pueblo: {Pierre}

This edge list can easily be stored in a dictionary, where the index is the vertex (say, the city named), and the value is a list of adjacent cities. If the graph is weighted the list is replaced by a second dictionary, again indexed by verticies, which contains the weight for each edge.

Adjacency Matrix Representation

The unweighted adjacency matrix representation of the graph given earlier can be defined in C as shown at right. We have used a symbolic constant to represent the size of the array.

```
# define N 8

int adjacency[N][N] = { { 1, 0, 0, 1, 0, 0, 0, 1},
                        { 0, 1, 0, 1, 0, 0, 0, 0},
                        { 0, 0, 1, 0, 0, 1, 0, 1},
                        { 0, 0, 1, 1, 0, 1, 0, 1},
                        { 1, 0, 0, 0, 1, 0, 0, 0},
                        { 0, 1, 0, 0, 0, 1, 0, 0},
                        { 0, 0, 0, 0, 0, 1, 1, 0},
                        { 0, 0, 0, 0, 1, 0, 0, 1} };
```

There are a number of questions one could ask concerning any particular graph. One of the fundamental problems is the question of *reachability*. That is, what is the set of vertices that can be reached by starting from a particular vertex and moving only along the given arcs in the graph. Conversely, are there any vertices that cannot be reached by traveling in such a fashion.

There are two basic variations on this question. The *single source* question poses a specific initial starting vertex, and requests the set of vertices reachable from this vertex. The *all pairs* question seeks to generate this information simultaneously for all possible vertices. Of course, a solution to the all pairs question will answer the single source question as well. For this reason, we would expect that an algorithm that solves the all pairs problem will require at least as much effort as any algorithm for the single source problem.

```
void warshall (int a[N][N]) {
        int i,j,k;
        for (k = 0; k < N; k++) {
                for (i = 0; i < N; i++)
                        if (a[i][k] != 0)
                          for (j = 0; j < N; j++)
                                a[i][j] |= a[i][k] & a[k][j];
        }
}
```

The algorithm we will describe to illustrate the use of the adjacency matrix representation of graphs is known as *Warshall's algorithm*, after the computer scientist credited with its discovery. The heart of Warshall's algorithm is a triple of loops, which operate much like the loops in the classic algorithm for matrix multiplication. The key idea is that at each

iteration through the outermost loop (index k), we add to the graph any path of length 2 that has node k as its center.

The process of adding information to the graph is performed through a combination of *bitwise* logical operations. The expression `a[i][k] & a[k][j]` is true if there is a path from node i to node k, and if there is a path from node k to node j. By using the bitwise or-assignment operator we add this to position `a[i][j]`. The use of bitwise-or ensures that if there was a 1 bit already in the position, indicating we had already discovered a previous path, then the earlier information will not be erased.

It is easy to see that Warshall's algorithms is $O(n^3)$, where n is the number of nodes in the graph. It is less easy to give a formal proof of correctness for the algorithm. To do so is beyond the purpose of the text here; however, suggestions on how such a proof could be developed are presented in the exercises at the end of the chapter.

The weighted adjacency matrix version of Warshall's algorithm is known as Floyd's algorithm; again, named for the computer scientist credited with the discovery of the algorithm. In the weighted version of the adjacency matrix some value must be identified as representing "not connected". For instance, we could use a large positive number for this purpose. In place of the bitwise-and operation in Warshall's algorithm, an addition is used in Floyd's; and in place of the bitwise-or used to update the path matrix, Floyd's algorithm uses a minimum value calculation. Each time a new path through the matrix is discovered, the cost of the path is compared to that of the previously known best, and if it is shorter than the new cost is recorded.

Edge List Representation

The adjacency matrix representation has the disadvantage that it always requires $O(v^2)$ space to store a matrix with v vertices, regardless of the number of arcs. An alternative representation stores only the arcs, and is thus advantageous if, as is common, the graph is relatively sparse. The basic idea is for each vertex to maintain both a value, and a list of those vertices to which it is connected. Because the function calls necessary to manipulate lists and maps can get in the way of understanding the fundamental algorithms, the techniques here are presented in the form of pseudo-code, rather than working C program.

Reachability for the edge list representation is more commonly expressed as a single-source problem. That is, given a single starting vertex, produce the set of vertices that can be reached starting from the initial location. We can solve this problem using a technique termed *depth-first search*. The basic algorithm can be described as follows:

Depth-first search shortest distance algorithm:
To find the set of vertices reachable from an initial vertex v_s
Create and Initialize set of *reachable* vertices
Add v_s to a stack
While stack is not empty

Get and remove (pop) last vertex v from stack
If v is known to be reachable, discard
Otherwise, add v to set of reachable vertices, then
    For all neighbors, v_j, of v
    If v_j is not is set of reachable vertices, add to stack

At each step of processing one node is removed from the stack. If it is a node that has not previously been reported as reachable, then the neighbors of the node are pushed on the stack. When the stack is finally empty, then all nodes that can be reached will have been investigated.

Question: Simulate the execution of this algorithm on the graph given earlier, using Pierre as your starting location. What city is not reachable from Pierre?

The type of search performed by this algorithm is known as a *depth-first* search. This mirrors a search that might be performed by an ant walking along the edges of the graph, and returning to a previous node when it discovers it has reached a vertex it has already seen. What is interesting about this algorithm is that if we replace the data structure, the stack, with an alternative, the queue, we get an entirely different type of search. Now, this is known as a *breadth-first* search. A breadth-first search looks at all possible paths at the same time. A mental image for this type of search would be to pour ink at the starting vertex, and see where it travels along all possible paths. In Worksheet 41 you will explore the differences between depth and breadth first search.

The weighted graph version of this algorithm is known as Dijkstra's algorithm, again in honor of the computer scientist who first discovered it. Dijkstra's algorithm uses a priority queue instead of a stack. It returns a dictionary of vertex/distance pairs. Because the priority queue orders values on shortest distance, it is guaranteed to find the shortest path. Dijkstra's algorithm can be described as follows:

Dijkstra's Shortest Path Algorithm:
To find the shortest distance to vertices reachable from an initial vertex v_s
Initialize dictionary of *reachable* vertices with v_s having cost zero,
and add v_s to a priority queue with distance zero
While priority queue is not empty
    Get and remove (pop) vertex v with shortest distance from priority queue
    If v is known to be reachable, discard
    Otherwise, add v with given cost to dictionary of reachable vertices
        For all neighbors, v_j, of v
        If v_j is not is set of reachable vertices, combine cost of reaching v
        with cost to travel from v to v_j, and add to priority queue

It is important that you add a vertex to the set of reachable nodes only when it is removed from the collection. As you will see when you complete worksheet 42, the cost when a vertex is first encountered in the inner loop may not be the shortest path possible.

Because a priority queue is used, rather than the stack as in the earlier algorithm, a shorter path that is subsequently uncovered will nevertheless be placed earlier in the queue, and thus will be the first to be removed from the queue.

Other Graph Algorithms

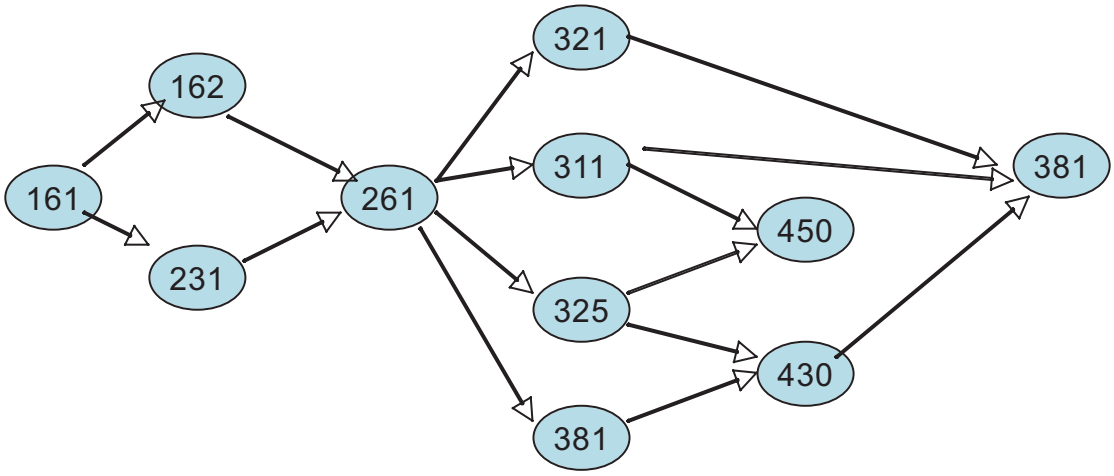
As we noted at the start of this chapter, graphs and matrices appear in a wide variety of forms in computer science applications. In this section we will present a number of other classic graph algorithms.

Greedy Algorithms

Many problems, particularly search problems, can be phrased in a form so that at each step there is a choice of many alternatives. The greedy method is a heuristic that provides guidance in how to construct a solution in these situations. The greedy technique says simply to take the next step that seems best from the current choices (that is, without resorting to any long-term planning). Dijkstra's algorithm is a classic greedy algorithm. At each step, we simply select the move that gets us to a new destination with the least amount of work.

Topological Sorting

Oftentimes a graph is used to represent a sequence of tasks, where a link between one task and the next indicates that the first must be completed before the second. When discussing university courses this is often termed a prerequisite chain. The following graph, for example, shows a typical table of courses and their prerequisites.



A topological order is a listing of vertices (in this case, courses), having the property that if b is a successor to a, then b occurs later in the ordering than a. For example, that you do not take any course without first taking its prerequisites. In effect, the topological ordering imposes a linear order on a graph. Of course, the topological order may not be unique, and many orderings may satisfy the sequence property.

A topological ordering can be found by placing all the vertices into a bag. The bag is then searched to find a vertex that has no successor. (As long as the graph has no cycles, such

a vertex must exist). This will be the end of our sorted list. The vertex is removed, and the bag examined once more. The result will be a topological sort.

Spanning Tree

A tree is a form of graph. Although all trees are graphs, not all graphs are trees. For example, trees have no cycles, and no vertex in a tree is pointed to by more than one other vertex. A spanning tree of a graph is a subset of the edges that form a tree that includes all the vertices.

To construct a spanning tree you can perform a depth-first search of the graph, starting from a given starting node. If you keep the list of neighbor nodes in a priority queue, as in dijkstra's algorithm, you can find the lowest-cost spanning tree. This can be accomplished by modifying dijkstra's algorithm to record an edge each time it is added to the set of reachable nodes.

The Traveling Salesman

Another graph problem that is of both theoretical and practical interest is that of the traveling salesman. In this problem a weighted graph represents the cost to travel between cities. The task is to find a path that touches every city, crosses no city more than once, and has least total cost.

This problem is interesting because there is no known polynomial time algorithm to find a solution. The problem belongs to a large class of problems termed NP, which stands for nondeterministic polynomial. The nondeterministic characterization comes from the fact that one can *guess* a solution, and in polynomial time check to determine whether it is the correct solution. But the nature of NP problems gets even more curious. The traveling salesman problem is what is known as NP-complete. This is a large collection of problems that has the property that if any of them could be solved in polynomial time, they all could. Unfortunately, nobody has ever been able to find a fast algorithm for any of these; nor has anybody been able to prove that such an algorithm cannot exist.

Study Questions

1. What are the two components of a graph?

2. What is the difference between a directed and an undirected graph?

3. What is the difference between a weighted and an unweighted graph?

4. What is an adjacency matrix?

5. For a graph with V vertices, how much space will an adjacency matrix require?

6. what are the features of an edge-list representation of a graph?

7. How much space does an edge-list representation of a graph require?

8. What are the two varieties of reachability questions for a graph?

9. What is the asymptotic complexity of Warhsall's algorithm?

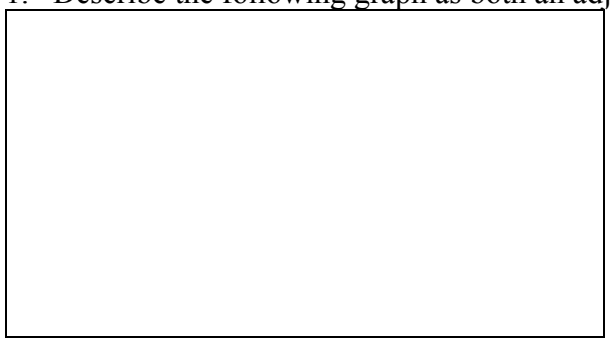
10. What problem is being addressed using Dijkstra's algorithm?

11. Why is it important that Dijkstra's algorithm stores intermediate results in a priority queue, rather than in an ordinary stack or queue?

12. In your own words, give an informal description of the difference between depth and breadth first search.

Exercises

1. Describe the following graph as both an adjacency matrix and an edge list:



2. Construct a graph in which a depth first search will uncover a solution (a path from one vertex to another) in fewer steps than will a breadth first search. You may need to specify an order in which neighbor vertices are placed into the container (e.g., lowest to highest). Construct another graph in which a breadth-first search will uncover a solution in fewer steps.

Analysis Exercises

1. Notice that the depth-first reachability algorithm did not specify what order neighboring cells should be placed into the container. Does the order matter? You can experiment by trying two different order, say listing the neighbors lowest-to-highest, and then highest-to-lowest.

2. A proof of correctness for Warshall's algorithm is based around the following induction technique. Prove, for each value of k, that if there is a path from node I to node j that does not go through any vertex with number higher than k, then a[i][j] is 1.

It is easy to establish the base case. For the induction step, assume that we are on the kth iteration, and that all paths that traverse only vertices labeled k or less have been marked. Assume that a path from vertex I to j goes through vertex k, and uses no vertices numbered larger than k. Argue why at the end of the loop the position a[i][j] will be set to 1.

3. A proof of correctness for the depth first reachability argument can be formed using induction, where the induction quantity is the length of the smallest path from the source to the destination. It is easy to argue that all vertices for which the shortest path is length 1 will be marked as reachable. For the induction step, argue why if all vertices with shortest path less than n are marked as reachable, then all vertices whose shortest path is length n will also be marked as reachable.

4. Give an example to illustrate why Dijkstra's algorithm will not work if there is an edge with a negative weight that is part of a cycle.

5. Rather than using induction, a proof of correctness for Dijkstra's algorithm is more easily constructed using proof by contradiction. Assume that the algorithm fails to find the shortest path of length greater than 1 from the source node to some node v, but instead finds a path of longer length. Because the path is larger than 1, there must be a next to last node along this path. Argue, from the properties of the priority queue, why this node must be processed before the node in question. Then argue why this will show that the shorter path will be discovered before the longer path that was uncovered.

6. Suppose you have a graph representing a maze that is infinite in size, but there is a finite path from the start to the finish. Is a depth first search guaranteed to find the path? Is a breadth-first search? Explain your answer.

7. Use the web to learn more about NP complete problems. What are some other problems that are NP complete? Why is the fact that factoring large prime number is NP complete a very practical importance at the present time?

Programming Projects

1. Using one of the representations for sets described in this text, provide an implementation of the depth-first shortest path algorithm.

2. Produce an implementation of Dijkstra's shortest path algorithm.

On the Web

Wikipedia has a collection of links to various algorithms under an entry “List of Algorithms”. Algorithms described include most of those mentioned in this chapter, as well as many others.