Chapter 2
Paths and Searching

Section 2.1 Distance

Almost every day you face a problem: You must leave your home and go to school. If you are like me, you are usually a little late, so you want to take the shortest route. How do you find such a route? Figure 2.1.1 models one such situation. Suppose vertex *h* represents your home, vertex *s* represents school and the other vertices represent intersections of the roads between home and school. Each edge represents a road and is labeled with its length (distance).

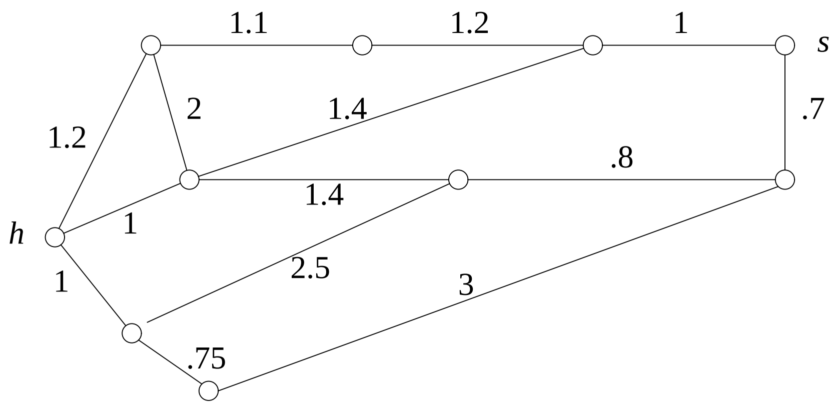

Figure 2.1.1. The model of routes to school.

Earlier, we discussed the idea of paths and of walking from one vertex to another. We also have seen that there may be many different paths between two vertices. I'm sure that ordinarily you feel it is best to take the shortest route between two places when you travel, and we are no different when it comes to graphs. But how do we measure the distance between two vertices in a graph? Certainly, following paths seems more efficient than following walks or trails since with paths we do not repeat any edges or vertices. Recall that we defined the length of a path to be the number of edges in the path. However, this definition treats each edge equally, as if each edge had length 1. This is inaccurate when the graph is modeling a road system, as in the problem just discussed. In order to adjust to more situations, we shall simply think of each edge as having a nonnegative label representing its length, and unless otherwise stated, that length will be 1. The *length of a path*, then, becomes the sum of the lengths of the edges in that path. The idea of distance stays the same no matter what the label; we merely let the *distance from x to y*, denoted $d(x, y)$, be the minimum length of an $x - y$ path in the graph. We now have a very general and flexible idea of distance that applies in many settings and is useful in many applications.

But how do we go about finding the distance between two vertices? What do we do to find the distance between all pairs of vertices? What properties does distance have? Does graph distance behave as one usually expects distance to behave? For that matter, how does one expect distance functions to behave? Let's try to answer these questions.

We restrict the length of any edge to a positive number, since this corresponds to our intuition of what length should be. In doing this, we can show that the following properties hold for the distance function *d* on a graph *G* (see exercises):

1. $d(x, y) \geq 0$ and $d(x, y) = 0$ if, and only if, $x = y$.
2. $d(x, y) = d(y, x)$.
3. $d(x, y) + d(y, z) \geq d(x, z)$.

These three properties define what is normally called a *metric function* (or simply a metric) on the vertex set of a graph. Metrics are well-behaved functions that reflect the three properties usually felt to be fundamental to distance (namely, properties $1-3$). Does each of these properties also hold for digraphs?

The *diameter,* denoted $diam(G)$, of a connected graph *G* equals $\max\limits_{u \in V} \max\limits_{v \in V} d(u, v)$, while the *radius* of *G*, denoted $rad(G)$, equals $\min\limits_{u \in V} \max\limits_{v \in V} d(u, v)$. Theorem 2.1.1 shows that these terms are related in a manner that is also consistent with our intuition about distance.

Example 2.1.1. We find the value of $d(x, y)$ for each pair of vertices *x*, *y* in the graph of Figure 2.1.2. These distances are shown in Table 2.1.1.

$d(x, y)$	a	b	c	d	e
a	0	1	2	1	3
b	1	0	1	2	2
c	2	1	0	1	1
d	1	2	1	0	2
e	3	2	1	2	0

Table 2.1.1 Table of distances in the graph of Figure 2.1.2.

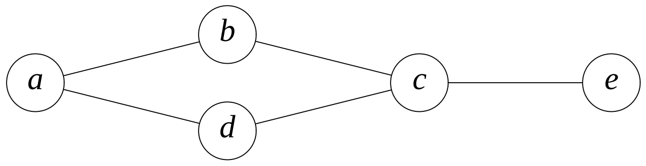

Figure 2.1.2. A graph of diameter 3 and radius 2.

Theorem 2.1.1 For any connected graph G,

$$rad(G) \leq diam(G) \leq 2rad(G).$$

Proof. The first inequality comes directly from our definitions of radius and diameter. To prove the second inequality, let $x, y \in V(G)$ such that $d(x, y) = diam(G)$. Further, let z be chosen so that the longest distance path from z has length $rad(G)$. Since distance is a metric, by property (3) we have that

$$diam(G) = d(x, y) \leq d(x, z) + d(z, y) \leq 2\ rad(G). \square$$

Another interesting application of distance occurs when you try to preserve distance under a mapping from one graph to another. A connected graph H is *isometric from* a connected graph G if for each vertex x in G, there is a 1-1 and onto function $F_x : V(G) \to V(H)$ that preserves distances from x, that is, such that $d_G(x, y) = d_H(F_x(x), F_x(y))$.

Example 2.1.2. The graph G_2 is isometric from G_1 (see Figure 2.1.3). The following mappings show that G_2 is isometric from G_1:

$$F_u\colon\ u \to d,\ w \to c,\ x \to a,\ v \to b$$
$$F_w\colon\ u \to a,\ w \to c,\ x \to d,\ v \to b$$
$$F_x = F_w$$
$$F_v\colon\ u \to a,\ w \to b,\ x \to d,\ v \to c.\ \square$$

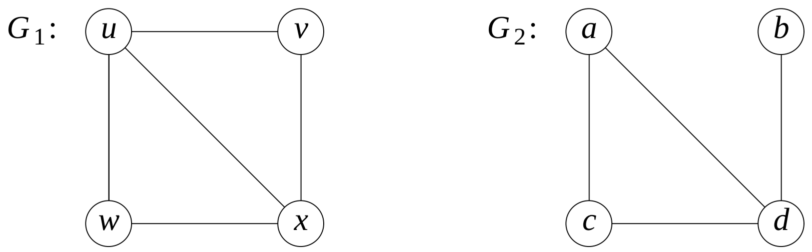

Figure 2.1.3. The graph G_2 is isometric from G_1 but not conversely.

Theorem 2.1.2 The relation isometric from is not symmetric, that is, if G_2 is isometric from G_1, then G_1 need not be isometric from G_2.

Proof. Graphs G_1 and G_2 of Figure 2.1.3 show a graph G_2 that is isometric from G_1.

To see that G_1 is not isometric from G_2, we note that each mapping F_i in the definition must preserve distances from the vertex i and there can be no mapping F_b with this property, because b has only one vertex at distance 1 from itself. □

We wish to develop efficient algorithms for computing distances in graphs. In order to do this, we begin by considering the simplest case, graphs with all edges of length 1. The first algorithm we present is from Moore [9] and is commonly called the *breadth-first search algorithm* or *BFS.* The idea behind a breadth-first search is fairly simple and straightforward. Beginning at some vertex x, we visit each vertex dominated by (adjacent from) x. Then, in turn, from each of these vertices, we visit any vertex that has not yet been visited and is dominated by the vertex we are presently visiting. We continue in this fashion until we have reached all vertices possible.

Algorithm 2.1.1 Breadth-First Search.
Input: An unlabeled graph $G = (V, E)$ with distinguished vertex x.
Output: The distances from x to all vertices reachable from x.
Method: Use a variable i to measure the distance from x, and label vertices with i as their distance is found.

```
1.  $i \leftarrow 0$.
```

2. Label x with "i."
3. Find all unlabeled vertices adjacent to at least one vertex with label i. If none is found, stop because we have reached all possible vertices.
4. Label all vertices found in step 3 with $i + 1$.
5. Let $i \leftarrow i + 1$, and go to step 3.

Example 2.1.3. As an example of the BFS algorithm, consider the graph of Figure 2.1.2. If we begin our search at the vertex d, then the BFS algorithm will proceed as follows:

1. Set $i = 0$.

2. Label d with 0.

3. Find all unlabeled vertices adjacent to d, namely a and c.

4. Label a and c with 1.

5. Set $i = 1$ and go to step 3.

3. Find all unlabeled vertices adjacent to one labeled 1, namely b and e.

4. Label b and e with 2.

5. Set $i = 2$ and go to step 3.

3. There are no unlabeled vertices adjacent to one labeled 2; hence, we stop. □

In essence, we can view the search as producing a *search tree,* using some edge to reach each new vertex along a path from x. In Figure 2.1.4 we picture two possible search trees for the previous example.

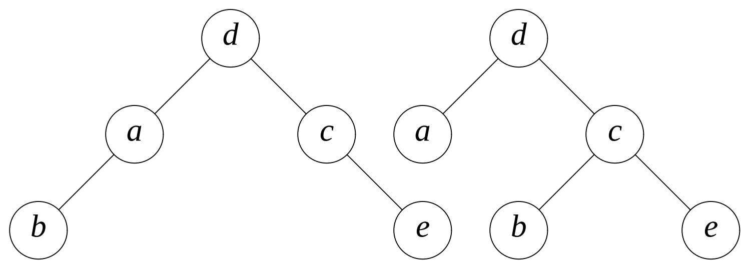

Figure 2.1.4. Two BFS search trees in the graph of Figure 2.1.2.

Theorem 2.1.3 When the BFS algorithm halts, each vertex reachable from x is labeled with its distance from x.

Proof. Suppose vertex $v = v_k$ has been labeled k. Then, by BFS (steps 3 and 4), there must exist a vertex v_{k-1}, which is labeled $k - 1$ and is adjacent to v_k, and similarly a vertex v_{k-2}, which is labeled $k - 2$ and is adjacent to v_{k-1}. By repeating this argument, we eventually reach v_0, and we see that $v_0 = x$, because x is the only vertex labeled zero. Then

$$x = v_0, v_1, \ldots, v_{k-1}, v_k = v$$

is a path of length k from x to v. Hence, $d(x, v) \leq k$.

In order to prove that the label on v is the distance from x to v, we apply induction on k, the distance from x to v. In step 2 of BFS, we label x with 0, and clearly $d(x, x) = 0$. Now, assume the result holds for vertices with labels less than k and let P: $x = v_0, v_1, \ldots, v_k = v$ be a shortest path from x to v in G. By the inductive hypothesis, $v_0, v_1, \ldots, v_{k-1}$ is a shortest path from x to v_{k-1}. By the inductive hypothesis, v_{k-1} is labeled $k - 1$. By the algorithm, when $i = k - 1$, v receives the label k. To see that v could not have been labeled earlier, suppose that indeed it had been labeled with $h < k$. Then there would be an $x - v$ path shorter than P, contradicting our choice of P. Hence, the result follows by induction. □

When the BFS algorithm is performed, any edge in the graph is examined at most two times, once from each of its end vertices. Thus, in step 3 of the BFS algorithm, the vertices labeled i are examined for unvisited neighbors. Using incidence lists for the data, the BFS algorithm has time complexity $O(|E|)$. In order to obtain the distances between any two vertices in the graph, we can perform the BFS algorithm, starting at each vertex. Thus, to find all distances, the algorithm has time complexity $O(|V|\ |E|)$. How does the BFS algorithm change for digraphs? Can you determine the time complexity for the directed version of BFS? Can you modify the BFS labeling process to make it easier to find the $x - v$ distance path?

Next, we want to consider arbitrarily labeled (sometimes these labels are called *weights*) digraphs, that is, digraphs with arcs labeled $l(e) \geq 0$. These labels could represent the length of the arc e, as in our home to school example, or the cost of traveling that route, or the cost of transmitting information between those locations, or transmission times, or any of many other possibilities.

When we wish to determine the shortest path from vertex u to vertex v, it is clear that we must first gain information about distances to intermediate vertices. This information is often recorded as a label assigned to the intermediate vertex. The label at intermediate vertex w usually takes one of two forms, the distance $d(u, w)$ between u and w, or sometimes the pair $d(u, w)$ and the predecessor of w on this path, $pred(w)$. The predecessor aids in backtracking to find the actual path.

Many distance algorithms have been proposed and most can be classified as one of two types, based upon how many times the vertex labels are updated (see [6]). In *label-setting* methods, during each pass through the vertices, one vertex label is assigned a value which remains unchanged thereafter. In *label-correcting* methods, any label may be changed during processing. These methods have different limitations. Label-setting methods cannot deal with graphs having negative arc labels. Label-correcting methods can handle negative arc labels, provided no negative cycles exist, that is, a cycle with edge weight sum a negative value.

Most label-setting or correcting algorithms can be recast into the same basic form which allows for finding the shortest paths from one vertex to all other vertices. Often what distinguishes these algorithms is how they select the next vertex from the candidate list C of vertices to examine. We now present a generic distance algorithm.

Algorithm 2.1.2a Generic Distance Algorithm.
Input: A labeled digraph $D = (V, E)$ with initial vertex v_1.
Output: The distance from v_1 to all other vertices.
Method: Generic labeling of vertices with label $(L(v), pred(v))$.

```
1.  For all $v \in V(D)$ set $L(v) \leftarrow \infty$.

2.  Initialize $C =$ the set of vertices to be checked.

3.  While $C \neq \emptyset$;
      Select $v \in C$ and set $C = C - v$.
      For all $u$ adjacent from $v$;
            If $L(u) > L(v) + l(vu)$;
                  $L(u) = L(v) + l(vu)$;
                  $pred(u) = v$;
                  Add $u$ to $C$ if it is not there.
```

One of the earliest label-setting algorithms was given by Dijkstra [1]. In Dijkstra's algorithm, a number of paths from vertex v_1 are tried and each time the shortest among them is chosen. Since paths can lead to new vertices with potentially many outgoing arcs, the number of paths can increase as we go. Each vertex is tried once, all paths leading from it are added to the list and the vertex itself is labeled and no longer used (label-setting). After all vertices are visited the algorithm is finished. At any time during execution of the algorithm, the value of $L(v)$ attached to the vertex v is the length of the shortest $v_1 - v$ path presently known.

Algorithm 2.1.2 Dijkstra's Distance Algorithm.
Input: A labeled digraph $D = (V, E)$ with initial vertex v_1.
Output: The distance from v_1 to all vertices.
Method: Label each vertex v with $(L(v), pred(v))$, which is the length of a shortest path from v_1 to v that has been found at that instant and the predecessor of v along that path.

```
1.  $L(v_1) \leftarrow 0$ and for all $v \neq v_1$ set $L(v) \leftarrow \infty$ and $C \leftarrow V$.
2.  While $C \neq \emptyset$;
      Find $v \in C$ with minimum label $L(v)$.
      $C \leftarrow C - \{ v \}$
      For every $e = v \rightarrow w$,
            if $w \in C$ and $L(w) > L(v) + l(e)$ then
            $L(w) \leftarrow L(v) + l(e)$ and $pred(w) = v$.
```

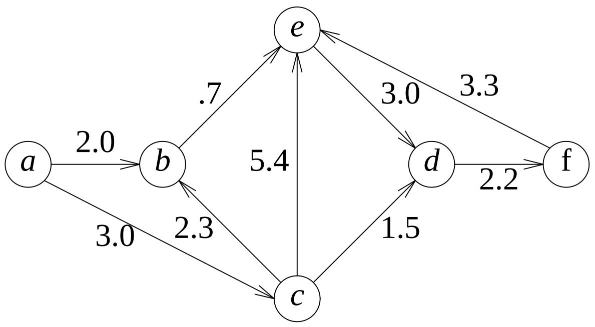

Figure 2.1.5. A digraph to test Dijkstra's algorithm.

Example 2.1.4. We now apply Dijkstra's algorithm to the digraph of Figure 2.1.5, finding the distances from vertex a. The steps performed and the actions taken are shown in the following table.

iteration	vertex	distances currently to vertices					
		a	b	c	d	e	f
initial		0	∞	∞	∞	∞	∞
1	a		2	3	∞	∞	∞
2	b			3	∞	2.7	∞
3	e			3	5.7		∞
4	c				4.5		∞
5	d						6.7
6	f						

Table 2.1.2 Distances from vertex *a* via Dijkstra's Algorithm.

We next verify that upon completion, Dijkstra's algorithm has labeled the vertices with their proper distances from *x*.

Theorem 2.1.4 If $L(v)$ is finite when Algorithm 2.1.2 halts, then $d(x, v) = L(v)$.

Proof. First we show that if $L(v)$ is finite, then there is a $v_1 - v$ path of length $L(v)$ in *D*. Since $L(v)$ is finite, its value must have changed during step 2 of the algorithm. Let *u* be the vertex used to label *v*, that is, $e = u \to v$ and $L(v) = L(u) + l(e)$. During this step, we delete *u* from *C*, and, hence, its label can never be changed again.

Now find the vertex *w* used to label *u* (this is just $pred(u)$) and again repeat this backtracking search. Eventually, we will backtrack to v_1, and we will have found a $v_1 - v$ path whose length is exactly $L(v)$. The backtracking search finds each time a vertex was deleted from *C*, and, thus, no vertex on the path can be repeated. Therefore, the path must backtrack to v_1, which has label 0.

Next, to see that $L(v) = d(v_1, v)$, we proceed by induction on the order in which we delete vertices from *C*. Clearly, v_1 is the first vertex deleted, and $L(v_1) = 0 = d(v_1, v_1)$. Next, assume that $L(u) = d(v_1, u)$ for all vertices *u* deleted from *C* before *v*.

If $L(v) = \infty$, let *w* be the first vertex chosen with an infinite label. Clearly, for every vertex *v* remaining in *C*, $L(v) = \infty$ and for all $u \in V - C$ (that is, those already deleted from *C*), $L(u)$ must be finite. Therefore, there are no arcs from $V - C$ to *C*, and since $v_1 \in V - C$ and $w \in C$, there are no $v_1 - w$ paths.

Next, suppose $L(v)$ is finite. We already know there exists a $v_1 - v$ path *P* of length $L(v)$; hence, it must be the case that $L(v) \geq d(v_1, v)$. In order to see that *P* is a shortest path, suppose that

$$P_1 : v_1, v_2, \ldots, v_k = v$$

is a shortest $v_1 - v$ path and let $e_i = v_i \to v_{i+1}$. Then

$$d(v_1, v_i) = \sum_{j=1}^{i-1} l(e_j).$$

Suppose v_i is the vertex of highest subscript on P_1 deleted from C before v. By the inductive hypothesis

$$L(v_i) = d(v_1, v_i) = \sum_{j=1}^{i-1} l(e_j).$$

If $v_{i+1} \neq v$, then $L(v_{i+1}) \leq L(v_i) + l(e_i)$ after v_i is deleted from C. Since step 2 can only decrease labels, when v is chosen, $L(v_{i+1})$ still satisfies the inequality. Thus,

$$\begin{aligned} L(v_{i+1}) &\leq L(v_i) + l(e_i) \\ &= d(v_1, v_i) + l(e_i) \\ &= d(v_1, v_{i+1}) \leq d(v_1, v). \end{aligned}$$

If $d(v_1, v) < L(v)$, then v should not have been chosen. If $v_{i+1} = v$, the same argument shows that $L(v) \leq d(v_1, v)$, which completes the proof. □

We now determine the time complexity of Dijkstra’s algorithm. Note that in step 2, the minimum label of C must be found. This can certainly be done in $|C| - 1$ comparisons. Initially, $C = V$, and step 4 reduces C one vertex at a time. Thus, the process is repeated $|V|$ times. The time required in step 2 is then on the order of $\sum_{i=1}^{|V|} i$ and therefore is $O(|V|^2)$.

Step 2 uses each arc once at most, so it requires at most $O(|E|) = O(|V|^2)$ time. The entire algorithm thus has time complexity $O(|V|^2)$. If we want to obtain the distance between any two vertices, we can use this algorithm once with each vertex playing the role of the initial vertex x. This process requires $O(|V|^3)$ time.

Can you modify Dijkstra’s algorithm to work on undirected graphs?

What can we do to further generalize the problems we have been considering? One possibility is to relax our idea of what the edge labels represent. If we consider these labels as representing some general value and not merely distance, then we can permit these labels to be negative. We call these generalized labels *weights*, and we denote them as $w(e)$. The "distance" between two vertices x and y will now correspond to the minimum sum of the edge weights along any $x - y$ path.

We noted earlier that label-setting methods failed when negative arc labels were allowed. To see why this happens in Dijkstra's Algorithm, consider the example below.

Example 2.1.5

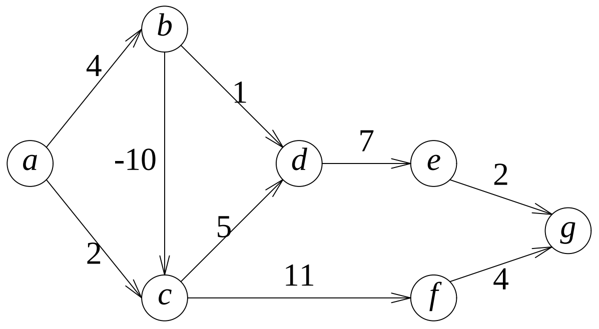

Figure 2.1.6. A digraph with negative arc weights allowed.

The chart below shows the changes made to the labels during the execution of Dijkstra's algorithm on the digraph of Figure 2.1.6. As you can see, the label set at 2 for vertex *c* is never corrected to it's proper value. The fact it can actually decrease because of negative weights is the problem.

iteration	vertex	current distances of vertices						
		a	b	c	d	e	f	g
init		0	∞	∞	∞	∞	∞	∞
1	a		4	2	∞	∞	∞	∞
2	c		4		7	∞	13	∞
3	b				5	∞	13	∞
4	d					12	13	∞
5	e						13	14
6	f							14
7	g							

However, the path a, b, c, d, e, g is never explored and has distance 8. □

Unfortunately, Dijkstra's algorithm can fail if we allow the edge weights to be negative. However, Ford [4, 5] gave an algorithm for finding the distance between a distinguished vertex x and all other vertices when we allow negative weights. This algorithm features a label correcting method to record the distances found.

Initially, x is labeled 0 and every other vertex is labeled ∞. Ford's algorithm, then, successively refines the labels assigned to the vertices, as long as improvements can be made. Arcs are used to decrease the labels of the vertices they reach. There is a problem, however, when the digraph contains a cycle whose total length is negative (called a *negative cycle*). In this case, the algorithm continually traverses the negative cycle, decreasing the vertex labels and never halting. Thus, in order to properly use Ford's algorithm, we must restrict its application to digraphs without negative cycles.

Algorithm 2.1.3 Ford's Distance Algorithm.
Input: A digraph with (possibly negative) arc weights $w(e)$, but no negative cycles.
Output: The distance from x to all vertices reachable from x.
Method: Label correcting.

```
1. $L(x) \leftarrow 0$ and for every $v \neq x$ set $L(v) \leftarrow \infty$.
2. While there is an arc $e = u \rightarrow v$ such that $L(v) > L(u) + w(e)$
        set $L(v) \leftarrow L(u) + w(e)$ and $pred(v) = u$.
```

Example 2.1.5. Now we apply Ford's algorithm to the digraph of Figure 2.1.7, which has some negative weights, but no negative cycles. We will find the distances from vertex a.

iteration	arc	current label					
		a	b	c	d	e	f
init		0	∞	∞	∞	∞	∞
1	$a \rightarrow b$	0	2.0	∞	∞	∞	∞
2	$b \rightarrow c$	0	2.0	4.3	∞	∞	∞
3	$c \rightarrow e$	0	2.0	4.3	∞	4.0	∞
4	$c \rightarrow d$	0	2.0	4.3	5.8	4.0	∞
5	$d \rightarrow e$	0	2.0	4.3	5.8	2.8	∞
6	$d \rightarrow f$	0	2.0	4.3	5.8	2.8	8.0

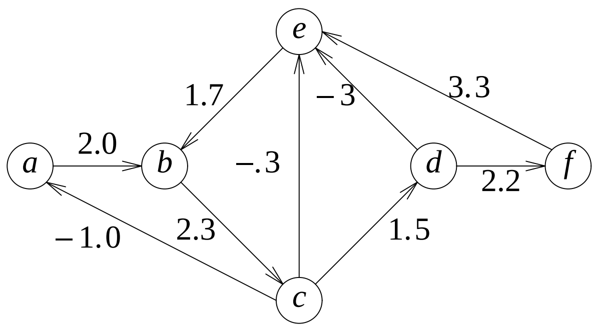

Figure 2.1.7. A digraph to test Ford's algorithm.

The next result assures us that Ford's algorithm actually produces the desired distances.

Theorem 2.1.5 For a digraph D with no negative cycles, when Algorithm 2.1.3 halts, $L(v) = d(x, v)$ for every vertex v.

In order to compute the time complexity of Ford's algorithm, we begin by ordering the arcs of D, say $e_1, e_2, \ldots, e_{|E|}$. We then perform step 2 of the algorithm by examining the arcs in this order. After examining all the arcs, we continue repeating this process until one entire pass through the arc list occurs without changing any label. If D contains no negative cycles, then it can be shown that if a shortest $x - v$ path contains k arcs, then v will have its final label by the end of the kth pass through the arc list (see exercises). Since $k \leq |V|$, Ford's algorithm then has time complexity bounded by $O(|E|\ |V|)$. Further, we can detect a negative cycle by seeing if there is any improvement in the labels on pass $|V|$.

Can you modify Ford's algorithm to make it easier to find the path?

Unfortunately, Ford's algorithm can only be used on digraphs. The problem with graphs is that any edge $e = xy$ with negative weight causes the algorithm to continually use this edge while decreasing the labels of x and y.

We conclude this section with a $O(|V|^3)$ algorithm from Floyd [3] for finding all distances in a digraph. Floyd's algorithm also allows negative weights. To accomplish Floyd's algorithm, we define for $i \neq j$:

$$d^0(v_i, v_j) = \begin{cases} l(e) & \text{if } v_i \to v_j \\ \infty & \text{otherwise} \end{cases}$$

and let $d^k(v_i, v_j)$ be the length of a shortest path from v_i to v_j among all paths from v_i to v_j that use only vertices from the set $\{ v_1, v_2, \ldots, v_k \}$. The distances are then updated as we allow the set of vertices used to build paths to expand. Thus, the d^0 distances represent the arcs of the digraph, the d^1 distances represent paths of length at most two that include v_1, etc. Note that because there are no negative cycles, the $d^k(v_i, v_i)$ values will remain at 0.

Algorithm 2.1.4 Floyd's Distance Algorithm.
Input: A digraph $D = (V, E)$ without negative cycles.
Output: The distances from v_i to v_j.
Method: Constant refinement of the distances as the set of excluded vertices is decreased.

```
1.  $k \leftarrow 1$.
2.  For every $1 \le i, j \le n$,
        $d^k(v_i, v_j) \leftarrow \min \{ d^{k-1}(v_i, v_j),\ d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j) \}$.
3.  If $k = |V|$, then stop;
        else $k \leftarrow k + 1$ and go to 2.
```

Example 2.1.6. We show one pass of Floyd's algorithm on the digraph of Figure 2.1.8. The d^0 and d^1 distances are shown in Tables 2.1.3 and 2.1.4:

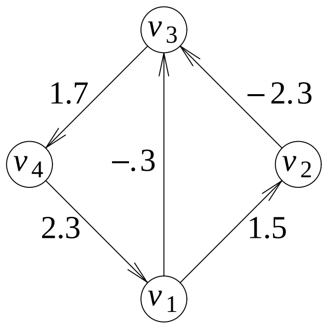

Figure 2.1.8. A digraph for Floyd's Algorithm.

d^0	v_1	v_2	v_3	v_4
v_1	0	1.5	-.3	∞
v_2	∞	0	-2.3	∞
v_3	∞	∞	0	1.7
v_4	2.3	∞	∞	0

Table 2.1.3 The d^0 distances.

We now find the d^1 distances as:

$$d^1(v_i, v_j) \leftarrow \min \{ d^0(v_i, v_j), \ d^0(v_i, v_1) + d^0(v_1, v_j) \}.$$

d^1	v_1	v_2	v_3	v_4
v_1	0	1.5	-.3	∞
v_2	∞	0	-2.3	∞
v_3	∞	∞	0	1.7
v_4	2.3	3.8	2.0	0

Table 2.1.4 The d^1 distances.

Many other algorithms exist to find distances in graphs. However, the techniques used in these algorithms often depend upon higher data structures or sorting techniques and are beyond the scope of this text. The interested reader should see the fine article by Gallo and Pallottino [6].

Section 2.2 Connectivity

In Chapter 1 we defined the idea of a connected graph, that is, a graph in which there is a path between any two vertices. In the last section, while constructing algorithms for finding distances, we actually developed algorithms for testing if a graph is connected. For example, if we apply BFS to a graph $G = (V, E)$, and we are able to visit every vertex of V, then G is connected. This follows by observing that for an arbitrary pair of vertices u and w, there is a path from x to w and from x to u; so these two paths can be used together to form a path from u to w.

There are other techniques for determining if a graph is connected. Perhaps the best known of these is the approach of Tremaux [11] (see also [7]) known as the *depth-first search.* The idea is to begin at a vertex v_0 and visit any vertex adjacent to v_0, say v_1. Now visit any vertex that is adjacent to v_1 that has not yet been visited. Continue to perform this process as long as possible. If we reach a vertex v_k with the property that all its neighbors have been visited, we backtrack to the last vertex visited prior to going to v_k, say v_{k-1}. We then try to visit new vertices neighboring v_{k-1}. If we can find an unvisited neighbor of v_{k-1}, we visit it. If we cannot find such a vertex, we again backtrack to the vertex visited immediately before v_{k-1}, say v_{k-2}. We continue visiting new vertices where possible and backtracking where necessary until we backtrack to v_0 and there are no unvisited neighbors at v_0. At this stage we have visited all possible vertices reachable from v_0, and we stop.

The set of edges used in the depth-first search from v_0 are the edges of a tree. Upon returning to v_0, if we cannot continue the search and the graph still contains unvisited vertices, we may choose such a vertex and begin the algorithm again. When all vertices have been visited, the edges used in performing these visits are the edges of a forest (simply a tree when the graph is connected).

The depth-first search algorithm actually partitions the edge set into two sets T and B: those edges T contained in the forest (and, hence, used in the search), which are usually called *tree edges,* and the remaining edges $B = E - T$, called *back edges.* The set B of back edges can be further partitioned as follows: Let B_1 be the set of back edges that join two vertices x and y, along some path from v_0 to y in the depth-first search tree that begins at v_0. Let C be the set of edges in B that join two vertices joined by a unique tree path that contains v_0. The edges of C are called *cross edges,* since they are edges between vertices that are not descendents of one another in the depth-first search tree.

Although it is not necessary in the search, we shall number the vertices v with an integer $n(v)$. The values of $n(v)$ represent the order in which the vertices are first encountered during the search. This numbering will be very useful in several problems that we will study later.

Algorithm 2.2.1 Depth-First Search.
Input: A graph $G = (V, E)$ and a distinguished vertex x.
Output: A set T of tree edges and an ordering $n(v)$ of the vertices.
Method: Use a label $m(e)$ to determine if an edge has been examined.
Use $p(v)$ to record the vertex previous to v in the search.

```
1.  For each $e \in E$, do the following: Set $m(e) \leftarrow$ "unused."
    Set $T \leftarrow \emptyset$, $i \leftarrow 0$.
    For every $v \in V$, do the following: set $n(v) \leftarrow 0$.
2.  Let $v \leftarrow x$.
3.  Let $i \leftarrow i + 1$ and let $n(v) \leftarrow i$.
4.  If $v$ has no unused incident edges, then go to 6.
5.  Find an unused edge $e = uv$ and set $m(e) \leftarrow$ "used."  Set $T \leftarrow T \cup \{ e \}$.
    If $n(u) \neq 0$, then go to 4;
          else $p(u) \leftarrow v$, $v \leftarrow u$ and go to 3.
6.  If $n(v) = 1$, then halt; else $v \leftarrow p(v)$ and go to 4.
```

There is an alternate way of expressing recursive algorithms that often simplifies their description. This method involves the use of procedures. The idea of a procedure is that it is a "process" that can be repeatedly used in an algorithm. It is started with certain values, and the variables determined within the procedure are local to it. The procedure may be halted at some stage by reference to another procedure or by reference to itself. Should this happen, another version of the procedure is invoked, with new parameters passed to it, and all values of the old procedure are saved, until this procedure once again begins its work. We now demonstrate such a procedural description of the depth-first search algorithm.

Algorithm 2.2.2. Recursive Version of the Depth-First Search.
Input: A graph $G = (V, E)$ and a starting vertex v.
Output: A set T of tree edges and an ordering of the vertices traversed.

```
1.  Let $i \leftarrow 1$ and let $F \leftarrow \emptyset$.  For all $v \in V$, do the following:  Set $n(v) \leftarrow 0$.
2.  While for some $u$, $n(u) = 0$, do the following: DFS($u$).
3.  Output $T$.
```

(The recursive procedure DFS is now given.)

Procedure DFS(v)

```
1.  Let $n(v) \leftarrow i$ and $i \leftarrow i + 1$.
2.  For all $u \in N(v)$, do the following:
     if $n(u) = 0$, then $T \leftarrow T \cup \{ e = uv \}$
     DFS($u$)
```

```
end DFS
```

The depth-first search is an extremely important tool, with applications to many other algorithms. Often, an algorithm for determining a graph property or parameter is dependent on the structure of the particular graph under consideration, and we must search that graph to discover this structural dependence. We shall have occasion to make use of the depth-first search as we continue to build other more complicated algorithms.

We now have a variety of ways of determining if a graph is connected. However, when dealing with graphs, you realize quickly that some graphs are "more connected" than others. For example, the path P_n seems much less connected than the complete graph K_n, in the sense that it is much easier to disconnect P_n (remove one internal vertex or edge) than it is to disconnect K_n (where removing a vertex merely reduces K_n to K_{n-1}).

Denote by $k(G)$ the *connectivity of G*, which is defined to be the minimum number of vertices whose removal disconnects G or reduces it to a single vertex K_1. Analogously, the edge connectivity, denoted $k_1(G)$, is the minimum number of edges whose removal disconnects G. If G is disconnected, then $k(G) = 0 = k_1(G)$. If $G = K_n$, then $k(G) = n - 1 = k_1(G)$, as we must remove $n - 1$ vertices to reduce K_n to K_1, and we can disconnect any vertex by removing the $n - 1$ edges incident with it.

We say G is *n-connected* if $k(G) \geq n$ and *n-edge connected* if $k_1(G) \geq n$. A set of vertices whose removal increases the number of components in a graph is called a *vertex separating set* (or *vertex cut set*) and a set of edges whose removal increases the number of components in a graph is called an *edge separating set* (or *edge cut set*). If the context is clear, we will simply use the term *separating set* (or *cut set*). If a cut set consists of a single vertex, it is called a *cut vertex* (some call it an *articulation point*), while if the cut set consists of a single edge, this edge is called a *cut-edge* or *bridge*.

Paths can be used to describe both cut vertices and bridges.

Theorem 2.2.1 In a connected graph G:

1. A vertex v is a cut vertex if, and only if, there exist vertices u and w ($u, w \neq v$) such that v is on every $u - w$ path of G.
2. An edge e is a bridge if, and only if, there exist vertices u and w such that e is on every $u - w$ path of G.

Proof. To prove (1), let v be a cut vertex of G. If u and w are vertices in different components of $G - v$, then there are no $u - w$ paths in $G - v$. However, since G is connected, there are $u - w$ paths in G. Thus, v must lie on every $u - w$ path in G.

Conversely, suppose that there exist vertices u and w in G such that v lies on every $u - w$ in G. Then in $G - v$, there are no $u - w$ paths, and so $G - v$ is disconnected. Thus, v is a cut vertex of G.

To prove (2), let e be a bridge of G. Then $G - e$ is disconnected. If u and w are vertices in different components of $G - e$, then there are no $u - w$ paths in $G - e$. But, since G is connected, there are $u - w$ paths in G. Thus, e must be on every $u - w$ path of G.

Conversely, if there exist vertices u and w such that e is on every $u - w$ path in G, then clearly in $G - e$ there are no $u - w$ paths. Hence, $G - e$ is disconnected, and, hence, e is a bridge. □

The depth-first search algorithm can be modified to detect the *blocks* of a graph, that is, the maximal 2-connected subgraphs. The strategy of the algorithm is based on the following observations which are stated as a sequence of lemmas.

Lemma 2.2.1 Let G be a connected graph with DFS tree T. If vw is not a tree edge, then it is a back edge.

Lemma 2.2.2 For $1 \leq i \leq k$, let $G_i = (V_i, E_i)$ be the blocks of a connected graph G. Then

1. For all $i \neq j$, $V_i \cap V_j$ contains at most one vertex.
2. Vertex x is a cut vertex if, and only if, $x \in V_i \cap V_j$ for some $i \neq j$.

Lemma 2.2.3 Let G be a connected graph and $T = (V, E_1)$ be a DFS search tree for G. Vertex x is a cut vertex of G if, and only if,

1. x is the root and x has more than one child in T, or
2. x is not the root and for some child s of x, there is no back edge between a descendant of s (including s itself) and a proper ancestor of x.

Suppose we perform a DFS on G and number the vertices $n(v)$ as usual. Further, suppose we define a lowpoint function LP(v) as the minimum number of a vertex reachable from v by a path in T followed by at most one back edge. Then we can use the lowpoint function to help us determine cut vertices.

Lemma 2.2.4 If v is not a root of T (a DFS tree for G) then v is a cut vertex if, and only if, v has a child s in T with $LP(s) \geq n(v)$.

These observations are illustrated in Figure 2.2.1, where tree edges are shown as dashed lines.

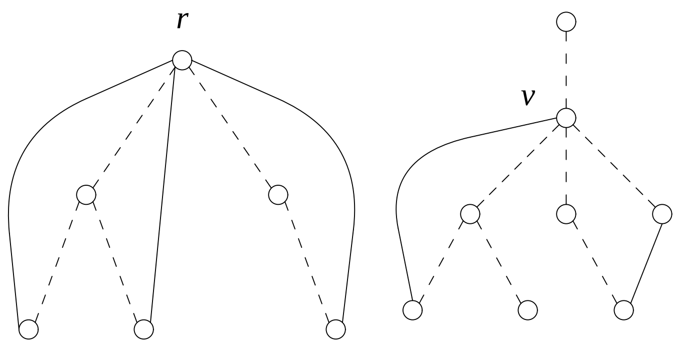

Figure 2.2.1. Cut vertex r as root and cut vertex v, not the root.

In order to identify the blocks of a graph, we really must identify the cut vertices. In order to make use of our observations, we must employ a device called a *stack* to help us. We record values on a stack and retrieve these values in the reverse order of their placement on the stack. The name stack comes from the idea that this structure behaves much like a stack of trays in a cafeteria. The first trays placed on the stack are the last ones taken (from the top) for use. Thus, stacks are often described as first in-last out devices.

Algorithm 2.2.3 Finding Blocks of a Graph.
Input: A connected graph $G = (V, E)$.
Output: The edges in each block of G.
Method: A modified DFS.

1. Set $T \leftarrow \emptyset$, $i \leftarrow 1$, and for all $v \in V$ set $n(v) \leftarrow 0$.
2. Select $v_0 \in V$ and call $UDFS(v_0)$
3. When vertex w is encountered in UDFS, push the edge vw on the stack (if it is not already there).

4. After discovering a pair vw such that w is a child of v and $LP(w) \geq n(v)$, pop from the stack all edges up to and including vw. These are the edges from a block of G

Upgraded DFS - Procedure $UDFS(v)$

```
1.  $i \leftarrow 1$
2.  $n(v) \leftarrow i, LP(v) \leftarrow n(v),\ i \leftarrow i + 1$.
3.  For all $w \in N(v)$, do the following:
        If $n(w) = 0$, then
            add $uv$ to $T$
            $p(w) \leftarrow v$
            $UDFS(w)$
            If $LP(w) \geq n(v)$, then a block has been found.
            $LP(v) \leftarrow \min\ (LP(v),\ LP(w))$
        Else If $w \neq p(v)$, then
            $LP(v) \leftarrow \min\ (LP(v),\ n(w))$
```

A simple inequality from Whitney [12] relates connectivity, edge connectivity and the minimum degree of a graph.

Theorem 2.2.2 For any graph G, $k(G) \leq k_1(G) \leq \delta(G)$.

Proof. If G is disconnected, then clearly $k_1(G) = 0$. If G is connected, we can certainly disconnect it by removing all edges incident with a vertex of minimum degree. Thus, in either case, $k_1(G) \leq \delta(G)$.

To verify that the first inequality holds, note that if G is disconnected or trivial, then clearly $k(G) = k_1(G) = 0$. If G is connected and has a bridge, then $k_1(G) = 1 = k(G)$, as either $G = K_2$ or G is connected and contains cut vertices. Finally, if $k_1(G) \geq 2$, then the removal of $k_1(G) - 1$ of the edges in an edge-separating set leaves a graph that contains a bridge. Let this bridge be $e = uv$. For each of the other edges, select an end vertex other than u or v and remove it from G. If the resulting graph is disconnected, then $k(G) < k_1(G)$. If the graph is connected, then it contains the bridge e and the removal of either u or v disconnects it. In either case, $k(G) \leq k_1(G)$, and the result is verified. □

Can you find a graph G for which $k(G) < k_1(G) < \delta(G)$?

Bridges played an important role in the last proof. We can characterize bridges, again taking a structural view using cycles.

Theorem 2.2.3 In a graph G, the edge e is a bridge if, and only if, e lies on no cycle of G.

Proof. Assume G is connected and let $e = uv$ be an edge of G. Suppose e lies on a cycle C of G. Also let w_1 and w_2 be distinct arbitrary vertices of G. If e does not lie on a $w_1 - w_2$ path P, then P is also a $w_1 - w_2$ path in $G - e$. If e does lie on a $w_1 - w_2$ path P', then by replacing e by the $u - v$ (or $v - u$) path of C not containing e produces a $w_1 - w_2$ walk in $G - e$. Thus, there is a $w_1 - w_2$ path in $G - e$ and hence, e is not a bridge.

Conversely, suppose $e = uv$ is an edge of G that is on no cycle of G. Assume e is not a bridge. Then, $G - e$ is connected and hence there exists a $u - v$ path P in $G - e$. Then P together with the edge e produces a cycle in G containing e, a contradiction. □

With the aid of Theorem 2.2.3, we can now characterize 2-connected graphs. Once again cycles play a fundamental role in the characterization. Before presenting the result, we need another definition. Two $u - v$ paths P_1 and P_2 are said to be *internally disjoint* if

$$V(P_1) \cap V(P_2) = \{ u, v \}.$$

Theorem 2.2.4 (Whitney [12]). A graph G of order $p \geq 3$ is 2-connected if, and only if, any two vertices of G lie on a common cycle.

Proof. If any two vertices of G lie on a common cycle, then clearly there are at least two internally disjoint paths between these vertices. Thus, the removal of one vertex cannot disconnect G, that is, G is 2-connected.

Conversely, let G be a 2-connected graph. We use induction on $d(u, v)$ to prove that any two vertices u and v must lie on a common cycle.

If $d(u, v) = 1$, then since G is 2-connected, the edge uv is not a bridge. Hence, by Theorem 2.2.3, the edge uv lies on a cycle. Now, assume the result holds for any two vertices at a distance less than d in G and consider vertices u and v such that $d(u, v) = d \geq 2$. Let P be a $u - v$ path of length d in G and suppose w precedes v on P. Since $d(u, w) = d - 1$, the induction hypothesis implies that u and w lie on a common cycle, say C.

Since G is 2-connected, $G - w$ is connected and, hence, contains a $u - v$ path P_1. Let z (possibly $z = u$) be the last vertex of P_1 on C. Since $u \in V(C)$, such a vertex must exist. Then G has two internally disjoint paths: one composed of the section of C from u to z not containing w together with the section of P_1 from z to v, and the other composed of the other section of C from u to w together with the edge wv. These two paths thus form a cycle containing u and v. □

A very powerful generalization of Whitney's theorem was proved by Menger [8]. Menger's theorem turns out to be related to many other results in several branches of discrete mathematics. We shall see some of these relationships later. Although a proof of Menger's theorem could be presented now, we postpone it until Chapter 4 in order to better point out some of these relationships to other results.

Theorem 2.2.5 (Menger's theorem). For nonadjacent vertices u and v in a graph G, the maximum number of internally disjoint $u - v$ paths equals the minimum number of vertices that separate u and v.

Theorem 2.2.4 has a generalization (Whitney [12]) to the k-connected case. This result should be viewed as the global version of Menger's theorem.

Theorem 2.2.6 A graph G is k-connected if, and only if, all distinct pairs of vertices are joined by at least k internally disjoint paths.

It is natural to ask if there is an edge analog to Menger's theorem. This result was independently discovered much later by Ford and Fulkerson [5] and Elias, Feinstein and Shannon [2]. We postpone the proof of this result until Chapter 4.

Theorem 2.2.7 For any two vertices u and v of a graph G, the maximum number of edge disjoint paths joining u and v equals the minimum number of edges whose removal separates u and v.

Section 2.3 Digraph Connectivity

The introduction of direction to the edges of a graph complicates the question of connectivity. In fact, we already know there are several levels of connectivity possible for digraphs. To help clarify the situation even further, we define a $u - v$ *semiwalk* to be a sequence $u = v_1, v_2, \ldots, v_k = v$, where for each $i = 1, 2, \ldots, k$, either $v_i \rightarrow v_{i+1}$ or $v_i \leftarrow v_{i+1}$ is an arc of the digraph. Can you define a $u - v$ semipath? A semipath may be a directed path in the digraph, or it may not. However, a semipath would be a $u - v$ path in the underlying graph. We say a digraph D is

1. *weakly connected* if every two vertices of D are joined by a semipath
2. *unilaterally connected (or unilateral*) if for every two vertices u and v, there is a directed $u - v$ path or a directed $v - u$ path in D
3. *strongly connected* (or *strong*) if all pairs of vertices u and v are joined by both a $u - v$ and a $v - u$ directed path.

If D satisfies none of these conditions, we say D is *disconnected.* As you might expect, each type of digraph connectivity can be characterized in terms of spanning semiwalks or paths. The proof of Theorem 2.3.1 is similar in nature to that of connectivity in graphs, and so it is omitted.

Theorem 2.3.1 Let D be a digraph. Then

1. D is weakly connected if, and only if, D contains a spanning semiwalk
2. D is unilateral if, and only if, D contains a spanning walk
3. D is strong if, and only if, D contains a closed spanning walk.

We say a vertex u is *reachable from* v if there exists a directed $v - u$ path in the digraph. The set of all vertices that are reachable from v is denoted as $R(v)$. The relation "mutually reachable" is an equivalence relation (see exercises); hence, this relation partitions the vertex set into classes $V_1, V_2, \ldots, V_k$ $(k \geq 1)$. Since the vertices u and v are in the same equivalence class if, and only if, D contains both a $u - v$ and $v - u$ directed path, the subgraphs $S_i = < V_i >$ have come to be called the *strong components* of D. Despite the fact that the strong components of D partition the vertex set of D, they do not necessarily partition the arc set. This fact can be seen in the example in Figure 2.3.1.

The term *strong component* is still appropriate, even when *D* is weakly connected or unilateral, since if S_1 and S_2 are two strong components, all arcs between these strong components are directed in one way, from S_1 to S_2 or from S_2 to S_1. Thus, there will always be vertices in one of these components that cannot reach any vertex of the other component.

Tarjan [10] developed an algorithm for finding the strongly connected components of a digraph. This algorithm makes use of the digraph $S_D = (V_S, E_S)$, called the *superstructure* of $D = (V, E)$, where

$$V_S = \{S_1, S_2, \ldots, S_k\} \text{ and}$$

$$E_S = \{ e = S_i \to S_j \mid i \neq j \text{ and } x \to y \in E \text{ where } x \in S_i \text{ and } y \in S_j \}.$$

Note that the digraph S_D must be acyclic, for if it were not, then all strongly connected components on some cycle of S_D would form one strongly connected component of *D*, contradicting the way the strong components were chosen. Now, we see that since S_D is acyclic, some vertex, say S_j, must have outdegree zero.

Suppose that we perform a depth-first search on *D*. Let *v* be the first vertex of S_j to be visited during this search. Since all the vertices of S_j are reachable from *v*, the depth-first search will never backtrack from *v* until all of the vertices in S_j have been visited. Thus, the number $n(u)$ assigned to each vertex *u* as it is first reached during the DSF ensures that each vertex of S_j has a number at least as large as $n(v)$. Since there are no arcs out of S_j, no vertex outside of S_j is visited from the time we first encounter *v* until we finally backtrack from *v*. Our only remaining problem is to determine when we actually perform a backtrack on the first vertex encountered in that strong component.

In order to solve this problem, we again turn to the bookkeeping lowpoint function. Here, the *lowpoint of v*, denoted $LP(v)$, is the least number $n(u)$ of a vertex *u* reachable from *v* using a (possibly empty) directed path consisting of tree arcs followed by at most one back arc or cross arc, provided *u* is in the same strong component as *v*. This definition seems circular. To find strong components, we need the lowpoint, and to find the lowpoint, we need to know the strong components. Tarjan [10] eliminated this problem by using a stack. The vertices visited are stored on a stack in the order in which they are reached during the search. For each vertex we also record whether it is on the stack, using the function onstack (with values of true or false). Again we traverse the digraph using a depth-first search.

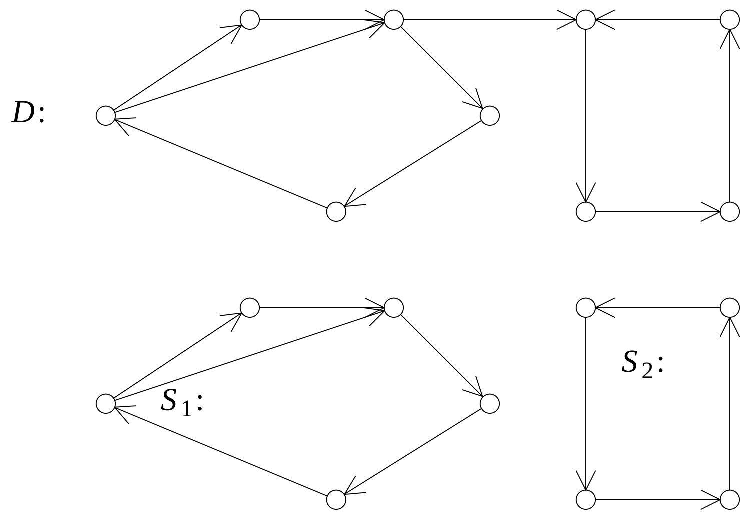

Figure 2.3.1. A digraph D and its strong components S_1 and S_2.

Algorithm 2.3.1 Finding Strongly Connected Components.
Input: A digraph D.
Output: The vertices of the strong components of D.

```
1. Set $i \leftarrow 1$ and empty the stack.
2. For all $v \in V$, do the following:
       $n(v) \leftarrow 0$
       onstack($v$) $\leftarrow$ false.
3. While $n(v) = 0$ for some $v$, do the following: strongcomp($v$).
```

Procedure strongcomp(v)

```
1. Set $n(v) \leftarrow i$, $LP(v) \leftarrow n(v)$ and $i \leftarrow i + 1$.
2. Place $v$ on the stack and set onstack($v$) $\leftarrow$ true.
3. For all $u \in N(v)$, do the following:
4.         If $n(u) = 0$, then do the following:
5.                 strongcomp($u$)
```

```
6.                  $LP(v) \leftarrow \min \{ LP(v), LP(u) \}$
7.            Else if $n(u) < n(v)$ and onstack($u$) = true,
                  then $LP(v) \leftarrow \min \{ LP(v), n(u) \}$.
8.   If $LP(v) = n(v)$ then delete and output the stack from the top down through $v$,
     and for each such vertex $w$, set onstack($w$) $\leftarrow$ false.
```

Can you determine the complexity of the strong component algorithm?

Section 2.4 Problem Solving and Heuristics

Suppose you are confronted with the following problem to solve (what else is new?). You have two water jugs, a 4-gallon jug and a 3-gallon jug. Neither jug has any measure markings on it. There is also a water pump that you can use to fill the jugs. Your problem, however, is that you want exactly 2 gallons of water in the larger jug so that you can make your secret recipe. How can you solve this problem?

The solution of this particular problem can actually help us see what some of the general techniques for problem solving are like. These techniques revolve around a search (often on a computer) for the solution among all possible situations the problem can produce. We may not even be sure that a solution exists when we begin this search or even know what the structure of the graph model happens to be.

There are many approaches one might take to problem solving, but any organized approach certainly encompasses the following points:

- Define the problem precisely. This includes precise specifications of what the initial situation will be as well as what constitutes an acceptable solution to the problem.
- Analyze the problem. Some feature or features can have a tremendous impact on the techniques we should use in solving the problem. Understand the "legal moves" you can make to try to find the solution.
- Choose the best technique and apply it.

We have just defined the water jug problem. We know that in the initial situation, both jugs are empty. We also know that a solution is found when there are exactly 2 gallons of water in the 4-gallon jug. Next, we must analyze the problem to try to determine what techniques to apply.

We can perform several "legal" operations with the water jugs:

- We can fill either jug completely from the pump.
- We can pour all the water from one jug into the other jug.
- We can fill one jug from the other jug.
- We can dump all the water from either jug.

Why have we restricted our operations in these ways? The answer is so that we can maintain control over the situation. By restricting operations in this way, we will always know exactly how much water is in either jug at any given time. Note that these operations also ensure that our search must deal with only a finite number of situations, since each jug can be filled to only a finite number of levels.

We have just examined an important concept, the idea of "moving" from one situation to another by performing one of a set of *legal moves.* What we want to do is to move from our *present state* (the starting vertex in the state space model) to some other state via these legal moves. Just what constitutes a legal move is dependent on the problem at hand and should be the result of our problem analysis.

The digraph model we have been building should now be apparent. Represent each state that we can reach by a vertex. There is an arc from vertex *a* to vertex *b* if we can move from state *a* to state *b* via some legal move. Our digraph and the collection of legal moves that define its arcs, constitute *the state space* (or the *state graph*) of the problem. Let's determine the state space of the water jug problem.

Label the state we are in by the ordered pair (j_1, j_2), which shows the amount of water in the 4-gallon and 3-gallon jugs, respectively. The initial state is then labeled $(0, 0)$. From this state we can move to either $(4, 0)$ or $(0, 3)$. Since we can move from these states back to $(0, 0)$, for simplicity we will join these vertices by one undirected edge representing the symmetric pair of arcs. We picture this in Figure 2.4.1.

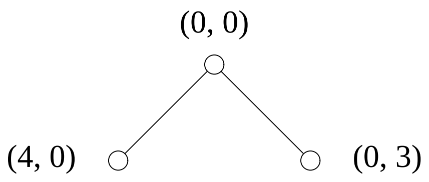

Figure 2.4.1. Early states we can reach.

From $(4,\ 0)$ we can move to $(0,\ 0)$, $(4,\ 3)$ or $(1,\ 3)$; while from $(0,\ 3)$ we can move to $(0,\ 0)$, $(4,\ 3)$ or $(3,\ 0)$. Thus, we have the situation in Figure 2.4.2.

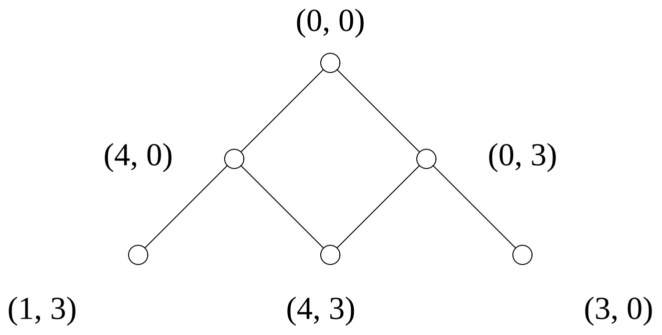

Figure 2.4.2. Two levels of the search.

It makes no sense for us to follow arcs that would return us to states we have already determined are not solutions; therefore, we will not bother to include arcs like the one from $(3,\ 0)$ to $(0,\ 0)$. As we continue to generate new states, we will eventually reach the graph in Figure 2.4.3.

We can see that this graph contains several possible solutions as well as several different paths from the initial vertex to these solutions. Thus, there are many ways to solve this particular problem. In fact, since any path from $(0,\ 0)$ to any vertex $(2,\ j_2)$ demonstrates a set of legal moves necessary to produce an acceptable solution, we will content ourselves with finding any one of these paths.

Our diagram representing the development of the state space also serves to point out another important fact. Rarely would we be presented with the state space and asked to find a path from the initial vertex to the solution. Instead, we would begin at a start vertex s and generate those vertices reachable from s in one move. If no solution is present, we begin generating neighbors of these vertices as we search for a solution. Thus, our search amounts to a *blind search*, that is, we cannot see the entire graph, only the local neighbors of the vertex we are presently examining. Luckily, we have already learned two search techniques that are essentially blind searches. Both the breadth-first search and the depth-first search are designed to operate in exactly this fashion.

There are some problems with using these search techniques on state spaces. The fundamental difficulty is that we have no idea how large the state space may actually be, and, therefore, we have no idea how many vertices we might have to examine. Especially in the case of the breadth-first search, where entire new levels of neighbors are being produced at one time, the amount of data we need to handle may increase exponentially. In the case of either search, there may simply be too many intermediate

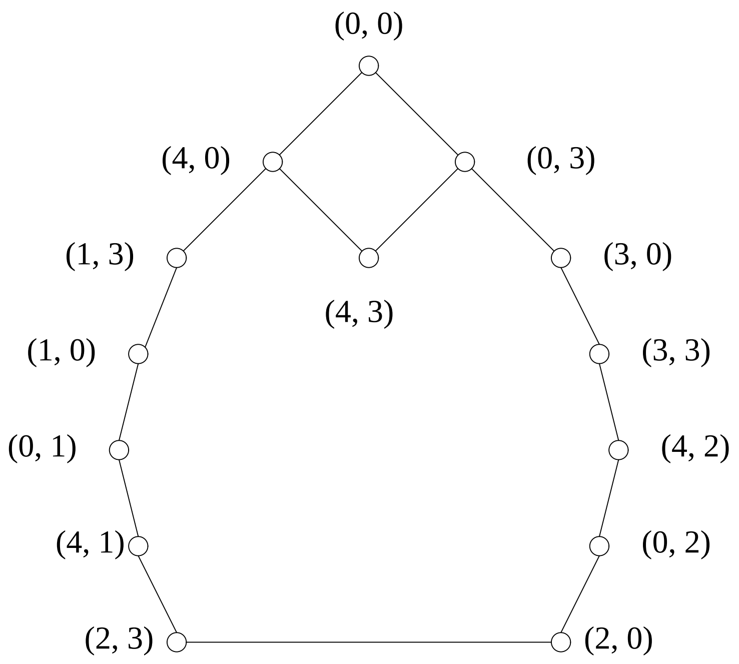

Figure 2.4.3. Many levels of the search (not all edges shown).

vertices to examine to be able to find any solution.

Often, we must sacrifice the completeness of a systematic search and construct a different control mechanism, one that hopefully will produce an acceptable solution in a reasonable amount of time. There are two approaches that are often tried:

- Perhaps it is the case that an exact answer is not necessary, that is, something close will be good enough for our purpose. Thus, we search for the best possible "approximate" answer we can find within a reasonable amount of time.
- Suppose, instead, that we sacrifice the guarantee (provided we have ample space and time) of finding a solution, because we make decisions during the search that mean we will never actually examine all vertices. If these decisions are good ones, we may work our way to a solution faster simply because we avoided taking a "wrong turn" in our search. This approach is clearly more dangerous because we may conclude our search with nothing more than what we started with. But, when it is clear that we may not be able to complete a thorough search anyway, the risk is often worth taking.

A *heuristic* is a technique that improves the efficiency of a search, while possibly sacrificing claims of completeness. Heuristics are very much like guided tours; they are intended to point out the highlights and eliminate time wasted on unnecessary events. Some heuristics are of more help than others, and usually this is a problem-dependent characteristic. However, several general heuristic techniques have become popular.

One general technique is called the *nearest neighbor algorithm.* The idea is that we examine all unvisited neighbors of some vertex and next visit the neighbor that most satisfies some test criterion. Our hope is that the test criterion will point us more rapidly toward a solution.

For example, suppose we apply the following heuristic to the water jug problem: During a DFS search we shall next visit the neighboring vertex whose label (j_1, j_2) has j_1 closest to the desired value of two. Under these conditions our search would proceed as follows:

- From $(0, 0)$ we generate $(4, 0)$ and $(0, 3)$. Since either has first coordinate within two of the goal, we randomly select the first as our next state.
- From $(4, 0)$ we generate the neighbors $(1, 3)$ and $(4, 3)$, and since $(1, 3)$ is best, we move there.
- From $(1, 3)$ we generate in turn (as there is only one neighbor each time) the sequence $(1, 0)$, $(0, 1)$, $(4, 1)$ and $(2, 3)$ and, thus, solve the problem.

In performing the above process, we examined less than half of the vertices in the state space and, thus, speeded our finding of a solution by a considerable amount. The general process we applied has been given the name *best-first search* because we used the "best" neighbor as our next choice.

Other modifications in these techniques are possible. Suppose that we decide to move only to the unvisited neighbor that produces the greatest improvement in our position, relative to some heuristic test. This technique is called *hill climbing.* We test all unvisited neighbors of the present vertex and using this information move to the vertex of greatest improvement. As you might already suspect, there are some obvious potential problems with this approach.

The major issue is what we do if the search reaches a vertex that is not a solution, but from which there are no neighbors that improve our position. There are several ways that this could happen. A *local maximum* is achieved if the present vertex is not a solution and all neighbors fail to improve our position. If, in fact, all the neighbors are essentially equivalent to our present vertex we say that a *plateau* has been reached. A far worse situation would be that the state space was not a connected graph and we were in a component that contained no solutions. Then, no simple move would ever achieve our goal.

Typical strategies for dealing with these problems are:

- Backtrack to a previous vertex and start the search again.
- Make a "big jump" to a new vertex, possibly by making two or more moves without regard to the heuristic test.
- Apply two or more moves all the time, using several levels of vertices to try to determine the next state. This process is called *lookahead.*

We have discussed a variety of options for trying to search for solutions to difficult problems. The central theme in each is that the set of possible solutions can be viewed as a (possibly infinite) graph or digraph. If we are able to search this graph exhaustively, we will find a solution, provided one exists.

However, if we cannot perform an exhaustive search, we are not necessarily doomed to failure. Creative heuristic tests can be (and often are) a great deal of help. The descriptions here are by no means a complete list of "standard heuristics" (if any such thing exists), but merely an indication that we should not immediately abandon a search when the graph model seems too large to handle.

Exercises

1. Show that graph distance is a metric function. Is distance still a metric function on labeled or weighted graphs?
2. Modify the BFS labeling process to make it easier to find the $x - v$ distance path.
3. Modify the BFS algorithm to find the distance from x to one specified vertex y.
4. Develop a recursive version of the BFS algorithm.
5. What modifications are necessary to make Dijkstra's algorithm work for undirected graphs?
6. Prove that the relation "is connected to" is an equivalence relation on the vertex set of a graph.
7. Show that if G is a connected graph of order p, then the size of G is at least $p - 1$.
8. Characterize those graphs having the property that every one of their induced subgraphs is connected.

9. Continue Example 2.1.6 by finding the tables for d^2, d^3 and d^4.
10. Prove that every circuit in a graph contains a cycle.
11. Prove that if G is a graph of order p and $\delta(G) \geq \frac{p}{2}$, then $k_1(G) = \delta(G)$.
12. Suppose that G is a (p, q) graph with $k(G) = n$ and $k_1(G) = m$, where both n and m are at least 1. Determine what values are possible for the following:
$$k(G - v),\ k_1(G - v),\ k(G - e),\ k_1(G - e).$$
13. Let G be an n-connected graph and let $v_1, v_2, \ldots, v_n$ be distinct vertices of G. Suppose we insert a new vertex x and join x to each of $v_1, v_2, \ldots, v_n$. Show that this new graph is also n-connected.
14. Prove that if G is an n-connected graph and $v_1, \ldots, v_n$ and v are $n + 1$ vertices of G, then there exist internally disjoint $v - v_i$ paths for $i = 1, \ldots, n$.
15. Show that if G contains no vertices of odd degree, then G contains no bridges.
16. Prove Theorem 2.1.5.
17. Prove Lemma 2.2.1.
18. Prove Lemma 2.2.2.
19. Prove Lemma 2.2.3.
20. Prove Lemma 2.2.4.
21. Prove Theorem 2.3.1.
22. Apply Algorithm 2.3.1 to the digraph D of Figure 2.3.1.
23. In applying Ford's algorithm to a weighted digraph D that contains no negative cycles, show that if a shortest $x - v$ path contains k arcs, then v will have its final label by the end of the kth pass through the arc list.
24. Modify the labeling in Ford's algorithm to make backtracking to find the distance path easier.
25. Show that G contains a path of length at least $\delta(G)$.
26. Show that G is connected if, and only if, for every partition of $V(G)$ into two nonempty sets V_1 and V_2, there is an edge from a vertex in V_1 to a vertex in V_2.
27. Show that if $\delta(G) \geq \frac{p - 1}{2}$, then G is connected.

28. Show that any nontrivial graph contains at least two vertices that are not cut vertices.

29. Show that if G is disconnected, then $\overline{G}$ is connected.

30. Show that if G is connected, then either G is complete or G contains three vertices x, y, z such that xy and yz are edges of G but $xz \notin E(G)$.

31. A graph G is a *critical block* if G is a block and for every vertex v, $G - v$ is not a block. Show that every critical block of order at least 4 contains a vertex of degree 2.

32. A graph G is a *minimal block* if G is a block and for every edge e, $G - e$ is not a block. Show that if G is a minimal block of order at least 4, then G contains a vertex of degree 2.

33. The *block index* $b(v)$ of a vertex v in a graph G is the number of blocks of G to which v belongs. If $b(G)$ denotes the number of blocks of G, show that
$$b(G) = k(G) + \sum_{v \in V(G)} (b(v) - 1).$$

34. Three cannibals and three missionaries are traveling together and they arrive at a river. They all wish to cross the river; however, the only transportation is a boat that can hold at most two people. There is another complication, however; at no time can the cannibals outnumber the missionaries (on either side of the river), for then the missionaries would be in danger. How do they manage to cross the river?

35. Prove that there can be no solution to the three cannibals and three missionaries problem that uses fewer than eleven river crossings.

36. Does a four-cannibal and four-missionary problem make sense? If so, explain this problem and try to solve it.

37. Three wives and their jealous husbands wish to go to town, but their only means of transportation is an RX7, which seats only two people. How might they do this so that no wife is ever left with one or both of the other husbands unless her own husband is present?

38. The 8-puzzle is a square tray in which are placed eight numbered tiles. The remaining ninth square is open. A tile that is adjacent to the open square can slide into that space. The object of this game is to obtain the following configuration from the starting configuration:

start

2	8	3
1	6	4
7		5

goal

1	2	3
8		4
7	6	5

How does this problem differ from those studied earlier? Can you build a mechanism into the rules that handles this difference?

39. A problem-solving search can proceed forward (as we have done) or backward from the goal state. What factors should influence your decision on how to proceed?

References

1. Dijkstra, E. W., A Note on Two Problems in Connection with Graphs. *Numerische Math.*, (1959), 269 – 271.
2. Elias, P., Feinstein, A. and Shannon, C. E., A Note on the Maximum Flow Through a Network. *IRE Trans. Inform. Theory,* IT-2(1956), 117 – 119.
3. Floyd, R. W., Algorithm 97: Shortest Path. *Comm. ACM,* 5(1962), 345.
4. Ford, L. R., *Network Flow Theory.* The Rand Corporation, P-923, August, 1956.
5. Ford, L. R. and Fulkerson, D. R., Maximal Flow Through a Network. *Canad. J. Math.,* 8(1956), 399 – 404.
6. Gallo, G. and Pallottino, S., Shortest Path Methods: A Unifying Approach. *Math. Programming Study* 26(1986), 38-64.
7. Hopcroft, J. and Tarjan, R., Algorithm 447: Efficient Algorithms for Graph Manipulation. *Comm. ACM,* 16(1973), 372 – 378.
8. Menger, K., Zur Allgemeinen Kurventheorie. *Fund. Math.,* 10(1927), 95 – 115.
9. Moore, E. F., The Shortest Path Through a Maze. *Proc. Iternat. Symp. Switching Th.,* 1957, Part II, Harvard Univ. Press, (1959), 285 – 292.

10. Tarjan, R., Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.,* 1(1972), 146 – 160.

11. Tremaux: see Lucas, E., *Recreations Mathematiques.* Paris, 1982.

12. Whitney, H., Congruent Graphs and the Connectivity of Graphs. *Amer. J. Math.,* 54(1932), 150 – 168.