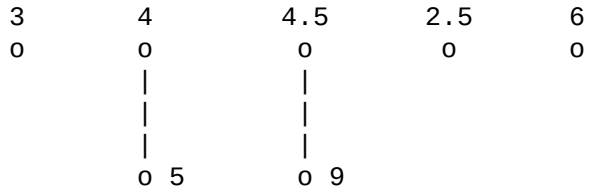
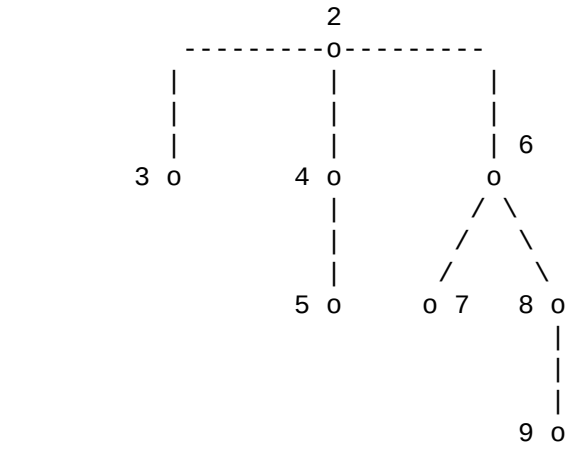


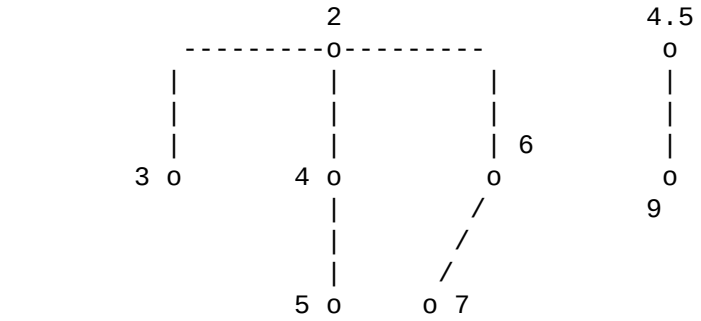
(d) After Deletemin.

Figure 3-18. Development of a Fibonacci Heap.

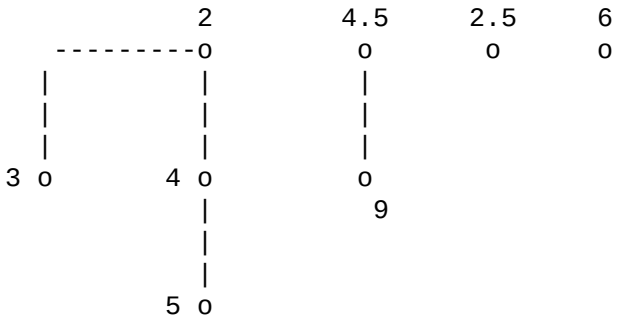
Section 3-5



(a) After First Deletemin.



(b) After Decrease-key 8 to 4.5.



(c) After Decrease-key 7 to 2.5.

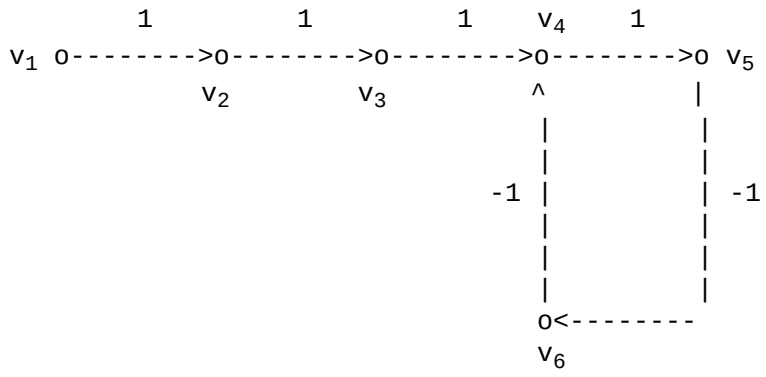


Figure 3-16. Negative Cycle Reachable from v_1.

SECTION 3-4

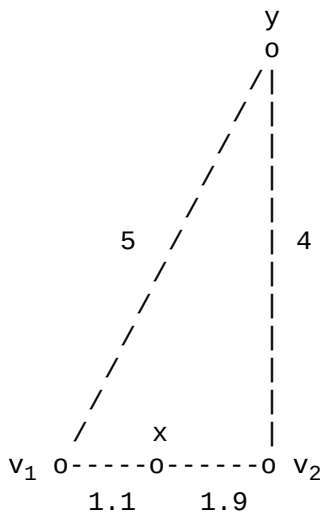


Figure 3-17. Example for Sedgewick-Vitter Algorithm.

SECTION 3-3

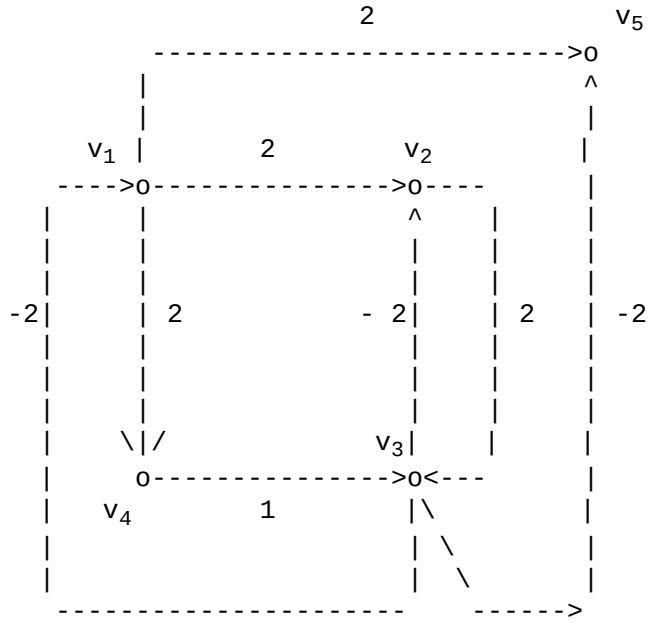


Figure 3-14. Weighted Graph G.

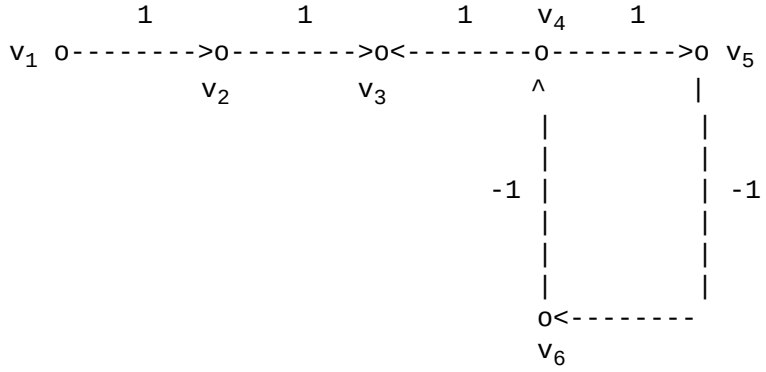
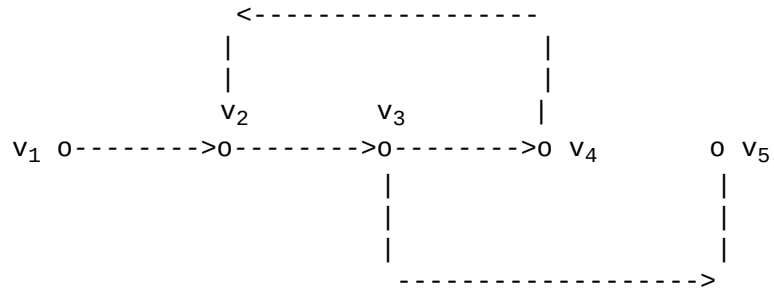
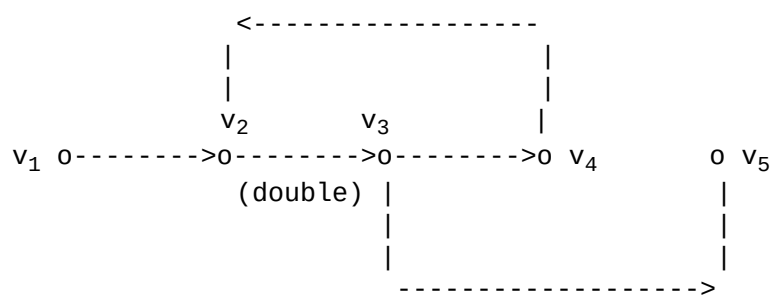


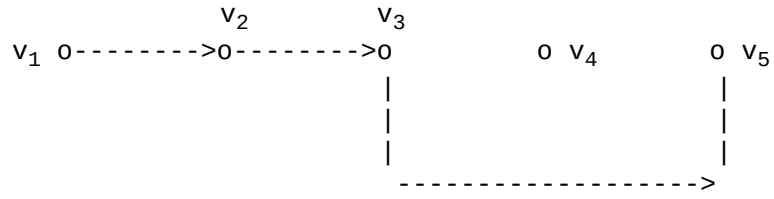
Figure 3-15. Negative Cycle Unreachable from v_1.



(a) Graph G



(b) Walk v_1-v_2-v_3-v_4-v_2-v_3-v_5.
(The edge v_2-v_3 is used twice.)



(c) Walk with Cycle Removed.

Figure 3-12. Reducing a Walk to a Path.

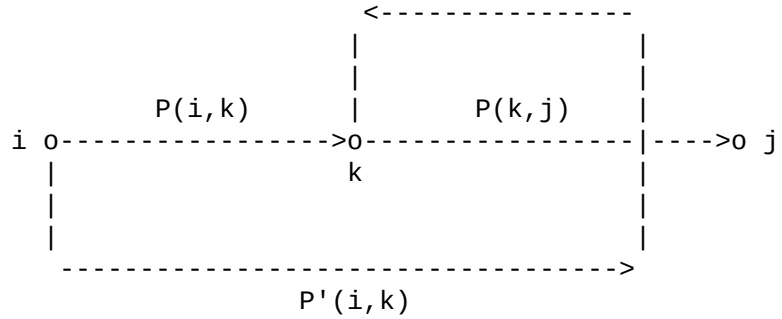


Figure 3-13. Subpath Optimality.

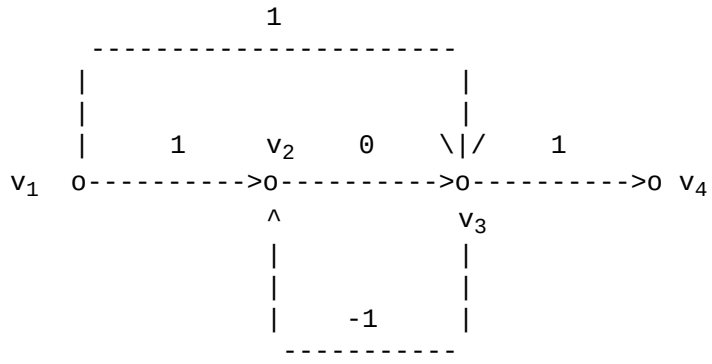


Figure 3-9. Shortest v_1-v_4 Walk < Shortest v_1-v_4 Path.

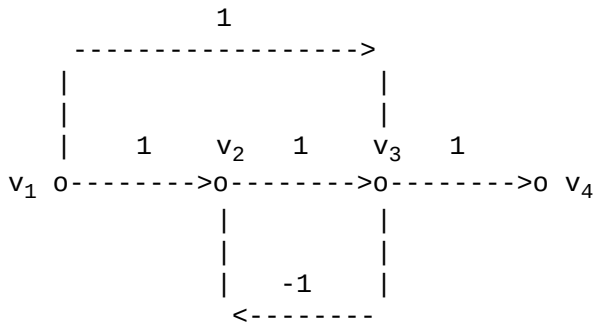


Figure 3-10. Shortest v_1-v_4 Walk = Shortest v_1-v_4 Path.

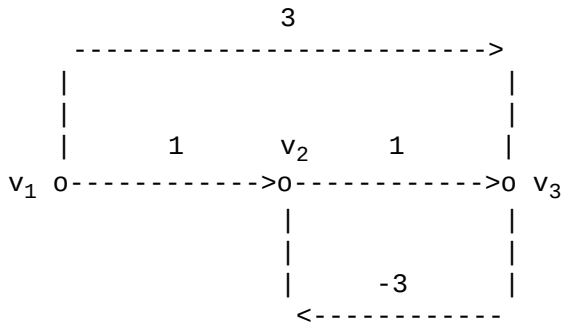


Figure 3-11. Subpath Property Fails.

0	2	2	M
M	0	M	2
M	M	0	1
-2	-2	0*	0

(b) SD after stage 1.

1	2	3	4
1	2	3	4
1	2	3	4
1	2	1*	4

(b') SP after stage 1.

0	2	2	4*
M	0	M	2
M	M	0	1
-2	-2	0	0

(c) SD after stage 2.

1	2	3	2*
1	2	3	4
1	2	3	4
1	2	1	4

(c') SP after stage 2.

0	2	2	3*
M	0	M	2
M	M	0	1
-2	-2	0	0

(d) SD after stage 3.

1	3*	3	3
4*	2	4*	2
4*	4*	3	4
1	2	1	4

(d') SP after stage 3.

0	1*	2	3
0*	0	2*	2
-1*	-1*	0	1
-2	-2	0	0

(e) SD after stage 4.

1	2	3	4
1	2	3	4
1	2	3	4
1	2	1*	4

(e') SP after stage 4.

Figure 3-8. Trace of SD and SP for G.

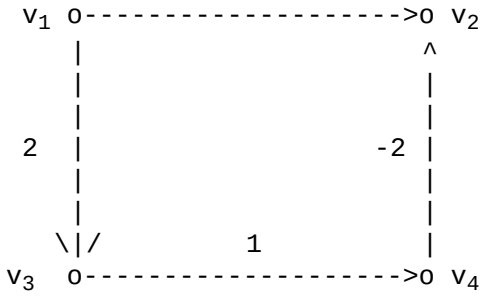


Figure 3-6. SP(1,2,4) = 3.

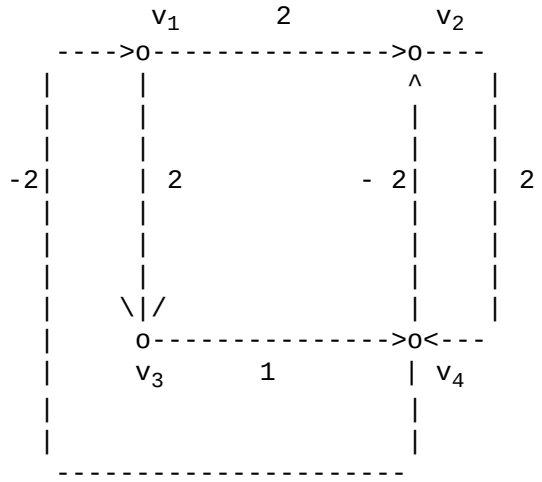
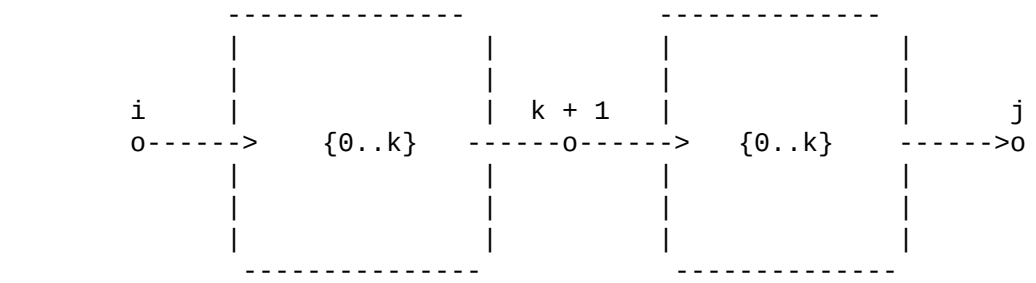


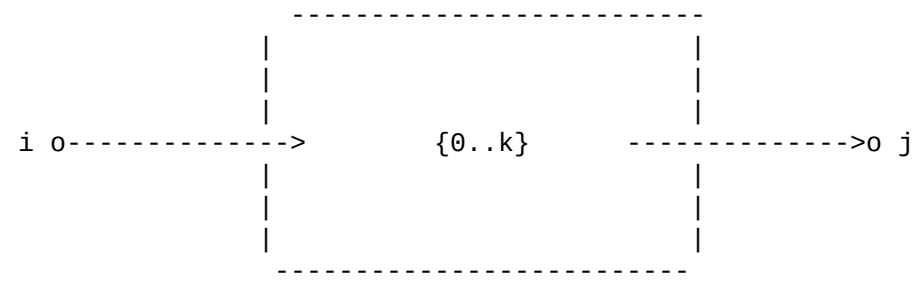
Figure 3-7. Weighted Graph G.

0	2	2	M	1	2	3	4
M	0	M	2	1	2	3	4
M	M	0	1	1	2	3	4
-2	-2	M	0	1	2	3	4

(a) Initial SD Matrix. (a')Initial SP Matrix.



(a) Vertex k + 1 Lies on Optimal Stage k + 1 Path from i to j.



(b) Vertex k + 1 Not on Optimal Stage k + 1 Path from i to j.

Figure 3-4. Dynamic Programming Alternatives.

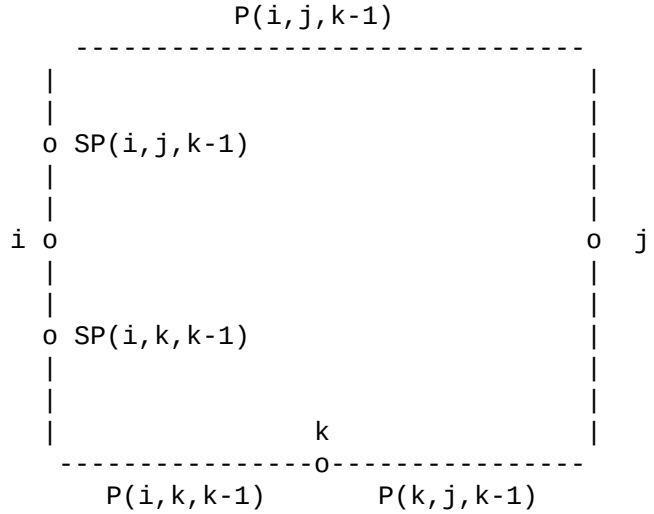


Figure 3-5. SP(i.j.k) = SP(i,j,k-1) or SP(i,k,k-1).

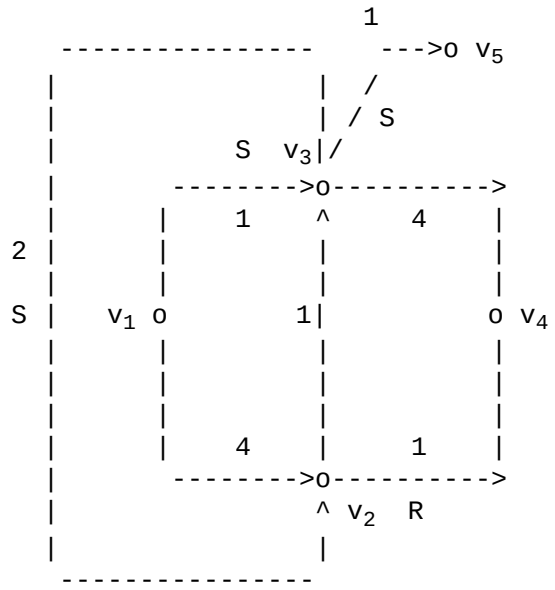


Figure 3-3c. Third Nearest Vertex Found.

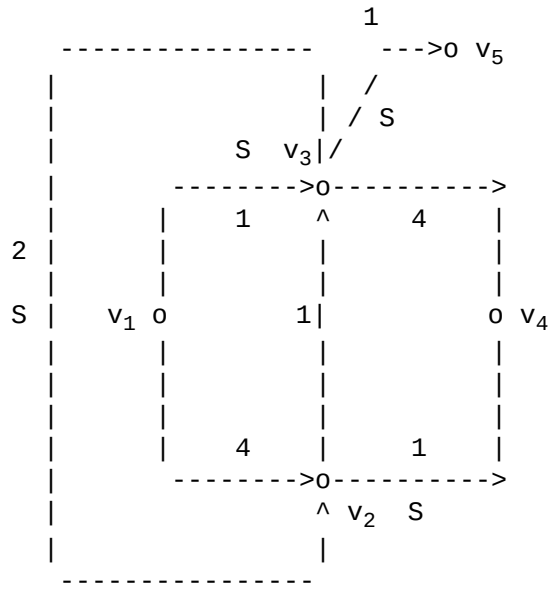


Figure 3-3d. Fourth Nearest Vertex Found.

Figure 3-3. Trace of Dijkstra's Algorithm for Digraph of Figure 3-2.

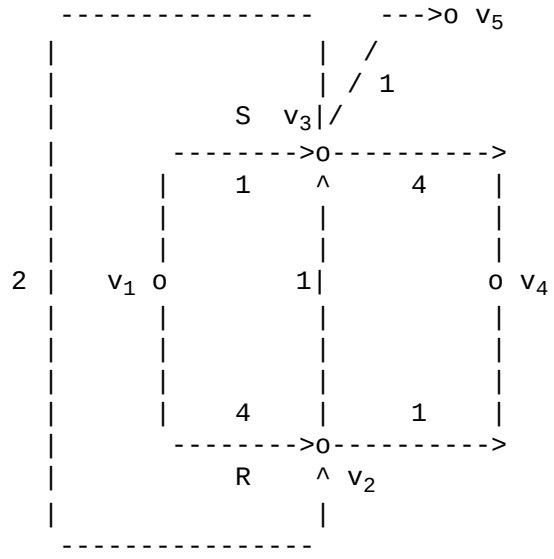


Figure 3-3a. First Nearest Vertex Found.

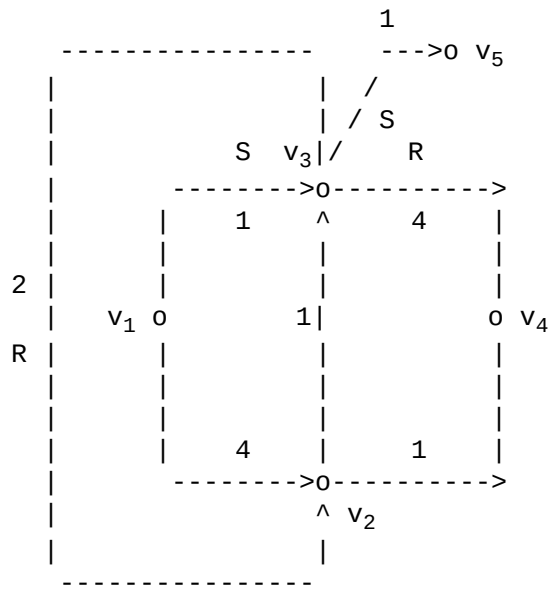


Figure 3-3b. Second Nearest Vertex Found.

SECTION 3-1

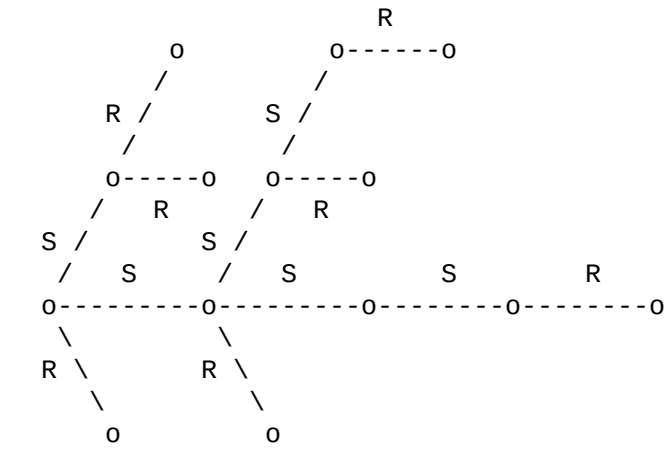


Figure 3-1. Search Tree.

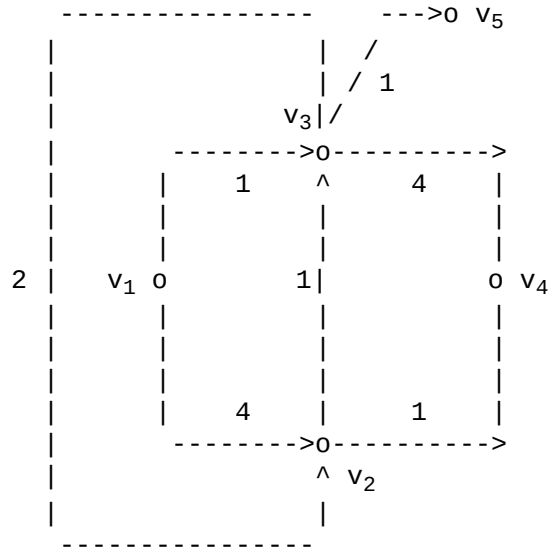


Figure 3-2. Weighted Digraph G.

(10) What difference(s) are there in the way Floyd's and Ford's algorithms react to negative cycles?

(11) If the shortest path from i to j is not unique, which path is selected by Floyd's algorithm?

(12) What graphical objects or invariants do we calculate if we replace min by max in Floyd's algorithm?

(13) Adapt Floyd's algorithm to find shortest paths avoiding a given set of vertices.

(14) Can you adapt Floyd's algorithm to find a shortest cycle on a given vertex? Explain.

(15) Suppose G(V,E) has a cut-vertex. How can you adapt Floyd's algorithm to capitalize on the structure of the graph?

(16) Prove the following specialized version of Floyd's algorithm (Warshall's algorithm) correctly computes the transitive closure of a digraph G(V,E):

```
   for k = 1 to |v| do
      for i = 1 to |v| do
         for j = 1 to |v| do
             Set A(i,j) to ( A(i,j) or (A(i,k) and A(k,j)) ),
```

where **and**/**or** bit operations are performed on the |v| by |v| adjacency matrix A interpreted as a bit matrix.

(17) What happens if Ford's algorithm is applied to a graph? What if the edge weights are restricted to be positive?

(18) The *transitive closure* of G(V,E) has an edge between every pair of vertices that are connected in G. The *transitive reduction* of G(V,E) is a digraph with the same transitive closure as G, but with as few edges as possible. We can also define a *minimal reduction* of G as a digraph with the same transitive closure as G and containing no smaller digraph with the same property. Is a transitive reduction unique? Is a minimal reduction unique? What is the relation between a transitive reduction and a minimal reduction. Can you design an algorithm to find a transitive reduction (Aho, Garey, and Ullman (1972))?

(19) Suppose each edge weight in a weighted graph gives the probability of failure of the edge and that we wish to find the most reliable path between a given pair of vertices. That is, find a path with the least probability of failure. Could Dijkstra's or the other shortest path algorithms be adapted to this problem?

(20) Compare the performance of Dijkstra's algorithm with that of the Sedgewick-Vitter algorithm for a class of randomly generated connected euclidean graphs.

(21) Design an algorithm to find the k shortest paths in a network with positive edge weights.

See Clarkson (1987) for the visibility graph, where faster algorithms for its calculation are described. The euclidean shortest path algorithm is from Sedgewick and Vitter (1986) who also give experimental results. See Fredman and Tarjan (1987) for Fibonacci heaps and their application to Dijkstra's algorithm. See Tarjan (1983) for a sophisticated discussion of shortest path problems and further references. Skiscim and Golden (1987) present generalized Dijkstra's algorithms for the k shortest paths in a euclidean network. Lipton and Tarjan (1977) give a planar shortest path algorithm; see Melhorn (1984) for a discussion. See Edmonds and Fulkerson (1968) and Edmonds and Karp (1972) for the minimax path edge length generalization of Dijkstra's problem and Aho, Garey, and Ullman (1972) for the transitive reduction problem, both of which are referred to in the exercises. Italiano (1986) gives an algorithm for transitive closure in a dynamic environmnent using lazy insertion that has amortized performance O(n log n) for a digraph of order n.

CHAPTER 3: EXERCISES

(1) Modify Dijkstra's algorithm to find all shortest paths between a given pair of vertices.

(2) Construct examples where Dijkstra's algorithm works correctly in the presence of a negative edge(s) and where it works incorrectly in the presence of a negative edge(s).

(3) How could you adapt Dijkstra's algorithm to the case where the vertices as well as the edges have weights which contribute to the "length" of a path?

(4) Let G(V,E) be a weighted digraph as in Dijkstra's problem, and let x and y be the source and target vertices for the problem. If a new edge is added to G, can the information provided by the Dijkstra traversal be efficiently updated, that is, without running the whole algorithm again? Consider as well the related problems of accounting for a change in the weight of an existing edge, or the deletion of an edge.

(5) Consider the same problem as in the previous question, except that now the modification is that a new vertex (and its set of incident edges) is added to G. Consider as well the related problem of deleting an existing vertex.

(6) There are N^2 shortest paths in a weighted digraph of order N. If the weight of a single edge changes, what is the maximum number of shortest paths that are affected?

(7) Modify Dijkstra's algorithm to find a shortest path between x and y such that the length of the maximum length edge on the path is minimized (Edmonds and Fulkerson (1968) and Edmonds and Karp (1972)).

(8) How well does the following greedy shortest path algorithm perform: always extend the estimated shortest-path-tree by adding to it the shortest edge from the tree to a nontree vertex.

(9) Design an algorithm for finding the shortest cycle in a positive weighted digraph.

point, the rank of y must have been at least i - 1 at that point also. Subsequently, by the cascading cut convention, y could have lost at most one child without being cut from x. Therefore, since y is still a child of x, its rank must be at least i - 2. If we denote the minimum size of a subtree rooted at a node of rank k by S_k; then by the previous observation $S_k \geq S_0 +$ ... +Sk-2 + 2. The Fibonacci numbers F_k satisfy the recurrence relation Fk+2 = F_2 + ... + F_k + 2, and so $S_k \geq$ Fk + 2. It is well-known that the Fibonacci numbers satisfy the inequality Fk + 2 $\geq f^k$, whence the theorem follows.

To complete the analysis of the heap operations, we need to modify the definition of the *potential* of a Fibonacci heap which we now define to be the number of trees in the heap plus twice the number of marked nonroot nodes. With this definition of potential, the amortized time analysis for the create, findmin, meld, insert, and deletemin operations is the same as before. For the decrease-key operation, we argue as follows: k cascade-cuts increase the number of trees by k but decrease the number of nonmarked nonroot nodes by k, so the net change in potential is -k. This counterbalances the actual time cost of the k cascade cuts. Thus, the O(1) amortized bound still holds. A more detailed description of the effect on the potential is as follows. Each operation increases the potential by at most 3 minus twice the number of cascade cuts. This arises from the following contributions: plus 1 (for the new tree introduced by the first cut), plus 2 (since the root of the new tree may have been a marked nonroot node), plus 1 for each cascade cut (each of which introduces a new tree), minus 2 for each cascade cut (since each cut reduces the number of nonmarked nonroot nodes), for an overall change in potential of at most 3 minus the number of cascade-cuts. Thus, the amortized time is still O(1), just as for the simpler situation before we introduced cascade cuts. The delete operation similarly takes O(log n) amortized time. The performance of the Fibonacci heaps is summarized in the following theorem.

THEOREM (FIBONACCI HEAP PERFORMANCE) Let F be a Fibonacci heap which is initially empty, and suppose we perform an arbitrary sequence of heap operations on F. Then, the total actual time of the operations is at most the total amortized time of the operations. The amortized time for the deletemin and delete operations is O(log n). The amortized times for the remaining heap operations are O(1).

Figure 3-18 here

Figure 3-18 illustrates the development of a Fibonacci heap. We assume the heap is initially empty and that elements with keys equal to 1 through 9 are inserted. Figure 3-18a illustrates the effect of a subsequent deletemin on this nine element heap. Figure 3-18b illustrates the effect of a decrease-key operation, where the key of the item with key equal to 8 is reduced to 4.5. Figure 3-16c illustrates the effect of reducing the key 7 to 2.5. Observe that the item with key 6 has lost two children by this point, which triggers a cascade cut of the key 6 item from its parent, the tree root with key 2. A final deletemin is shown in Figure 3-18d, at which point there are five nodes on the root list.

CHAPTER 3: REFERENCES AND FURTHER READING

potential by k, counterbalancing the cost in actual time. Consequently, the amortized time remains at most O(log n). Observe that the actual time of a deletemin may even be linear in n; for example, consider the very first deletemin operation after a sequence of insertions into an empty heap. But, the running average tracked by the amortized time is guaranteed to be at most O(log n).

It remains to describe the implementation of the decrease-key and delete(h,i) operations, and to establish the logarithmic upper bound on the maximum rank of a tree.

To decrease the key of an item u by a positive amount, we first locate the node for the item, which we can directly find using the auxiliary array; decrease its key; break the pointer to its parent, if any; and adjust its parent's rank and child pointer. The node u becomes a root and may even be the new minimum root. All this takes O(1) actual time. Later, we will introduce a refinement, which is necessary to ensure the O(log n) rank bound assumed in the analysis of deletemin. This refinement may trigger a chain of additional actions when the link between u and its parent is broken, but there will be a countervailing effect on a correspondingly modified version of the potential, so that the amortized time bound will still be O(1). Delete(h,i) is implemented similarly. We merely break the link between item i and its parent; combine the list of the children of item i with the original root list. This all takes O(1) time, except if item i happens to be the minimum root, in which case the delete operation degenerates into a deletemin operation with O(log n) amortized time.

The refinement of decrease-key and delete needed to ensure the proper time bounds is as follows. A linking operation makes a node u a child of a node v. As soon as u loses two of its children, we apply the following rule: cut the link between u and its parent v, thus making u a root. Whenever this condition occurs, the induced cutting action can obviously trigger a chain of such cuts, called a *cascade of cuts*. A simple artifice allows us to implement the test for this situation efficiently. Thus, whenever we link a pair of nodes u and v, making u a child of v, we turn off the mark bit in the child u. On the other hand, whenever we cut the pointer from a node u to its parent v, if the parent v is an unmarked nonroot node, we turn on the mark bit in v (to indicate v has just lost one child); while if the parent v is a marked nonroot node (which indicates v has previously lost one child), we apply the cutting rule by cutting the edge between v and the parent of v.

The effect of the cutting rule is to ensure that the size of any tree in a Fibonacci heap is at least exponential in its rank, as demonstrated by the following theorem; f denotes the golden ratio (1 + û5)/2.

THEOREM (FIBONACCI RANK BOUND) A node of rank k in a Fibonacci heap has at least f^k descendants.

The proof is as follows. Consider the children of a node x of rank k in the order in which they became children of x. Observe that the i^{th} child y of x has rank at least i - 2. For, by definition, when y was first linked to x, both x and y had the same rank, and, since x had rank at least i - 1 at that

time. Meld(h_1, h_2) combines the root lists of h_1 and h_2 into a single list and sets the new minimum to the smaller of the original minima in O(1) time. Insert(h,i) inserts element i in heap h by first creating a new heap h' consisting solely of item i, and then melding h and h'. The deletemin, decrease-key, and delete(h,i) operations are more complex to implement.

Deletemin(h) deletes the minimum element from the Fibonacci heap h. It first removes the node u containing the minimum (root) element. This leaves the children of u without a parent, and so the list of children of u is combined with the existing root list, making each child of u a new root. We then apply a series of linking operations on the resulting heap ordered trees, where each root determines a tree. Thus, if any two trees (roots) have equal rank, we link them by making the root with the larger key point to the root with the smaller key, breaking ties arbitrarily, and updating any effected fields, including the rank of the new root. This linking process is repeated until no two trees of equal rank remain. The new root list and the new minimum root can be identified on the fly. The worst-case time of this operation may be greater than O(log n). However, the amortized time of the operation, which we shall define shortly, can be shown to be O(log n). We can efficiently find and link trees of equal rank by augmenting the heap with an additional array of pointers A, of size at most O(log n). The components of A are initially nil. As we scan the root list, and encounter a root u of rank i, we look up A(i). If A(i) is nil, we make A(i) point to u. If A(i) already points to the root v of another tree of rank i, we link the trees rooted at u and v, making the larger root point to the smaller root, creating a tree of rank i + 1, and setting A(i) to nil. We then continue the scanning and linking process by trying to insert the new tree at A(i + 1), iterating in this manner until we eventually create a tree T of some rank j not equal to the rank of any existing tree. At that point, we make A(j) point to the root of T. This process is repeated until all the rooted trees have roots of distinct rank.

We introduce the amortized time of a heap operation as a way of modelling the uneven performance of an operation whose average performance is much better than its worst case performance. First, we define the *potential* of a Fibonacci heap to be the number of trees it contains. The *amortized time* of a heap operation is then defined as the actual execution time of the operation plus the change in potential the operation causes. The *total amortized time* of a sequence of heap operations is the total actual execution time of the operations plus the net change they cause in potential. If the Fibonacci heap initially has zero trees, the net change in potential is necessarily nonnegative; so the total amortized time is an upper bound on the total actual time of the sequence of operations.

The amortized time of the create, findmin, meld, and insert operations are each O(1), since the actual time of each operation is O(1), and only the insert operation affects the potential, increasing it by 1.

The deletemin operation takes amortized time O(log n). For, it takes O(1) actual time to remove the minimum element and combine its children with the other roots. Assuming the maximum rank is O(log n), this increases the potential by at most O(log n); so the amortized time may be O(log n). The actual time to perform k linking operations is O(k), but each linking operation reduces the potential of the heap by 1; so k links reduce the

```
            Set Pred(w) to v
            Change (Reached,w)

End_Procedure_Sedgewick_Vitter
```

FIBONACCI HEAPS AND DIJKSTRA'S ALGORITHM

We showed how to enhance the performance of Dijkstra's algorithm using heaps in Section 3-1. We can improve its performance even further using a sophisticated implementation of a heap called a Fibonacci heap. This leads to an O(|v| log |v| + |E|) implementation of Dijkstra's algorithm for a digraph G(V,E). This improvement arises from the ability of a Fibonacci heap to realize the basic heap operations in O(log |v|) (average) time each, and the special change-key (strictly, decrease-key) operation required by Dijkstra's algorithm in O(1) (average) time. The performance estimates of each of the Fibonacci heap operations will be given not in terms of worst-case times, but in terms of the so-called amortized time of each operation, which is essentially a running average of the time for the operation. Despite this, the performance estimate for the resulting implementation of Dijkstra's algorithm is still in terms of worst-case times. Thus, Dijkstra's algorithm uses the heap operations which require O(log |v|) amortized time at most O(|v|) times each, and it uses the O(1) change-key operation at most O(|E|) times, whence the O(|v| log |v| + |E|) performance estimate follows.

A heap is usually represented as a one-dimensional array. The representation of a Fibonacci Heap is considerably more complex and is defined in terms of a collection of heap ordered trees. A *heap ordered tree* is a rooted tree with keys arranged in heap order: the smallest key is at the root of the tree, and the root of any subtree is the smallest key in that subtree. A *Fibonacci Heap* is a disjoint collection of heap ordered trees. Each node u in a Fibonacci heap has four pointers: a pointer to the tree parent of u (which is nil if u is a root); a pointer to one of the children of u; and two other pointers that serve to embed u in a doubly linked circular list consisting of all the siblings of u. The roots of the separate heap ordered trees are also arranged in a doubly linked circular list. Each node u contains a field Rank(u) which gives the number of children of u. Although there is no a priori bound on the rank of any node, the heap operations are implemented in such a way that the maximum rank is at most O(log n), for a heap with n nodes. Each node also contains a mark bit used to facilitate certain of the heap operations. A separate minimum pointer points to the root containing the smallest key. Finally, just as for the heap representation described in Section 3-1, we assume we can directly access the nodes of the heap by an auxiliary array of pointers.

We will consider the following heap operations: create, findmin, meld, insert, deletemin, decrease-key, and delete. The meld(h_1, h_2) operation combines the Fibonacci heaps h_1 and h_2 into a single Fibonacci heap. The Decrease-key operation decreases the key of an item in the heap by a positive amount. The Delete(h,i) operation deletes item i from the heap. Several of the operations are trivial to implement. Create merely returns a pointer to an empty heap in O(1) time. Findmin merely accesses the minimum pointer in O(1)

There are two reasons for the superior performance of the Sedgewick-Vitter algorithm. First, the algorithm terminates as soon as y is reached. Second, the heuristic distance estimates used to select the next neighbor to enter S are designed to tend to make the search tree grow in the direction of y. This is only a tendency and not a strictly guaranteed effect.

Refer to Figure 3-17 for an example. When the search tree Reached is first extended from x, Heur_Dist(v_1) is 6.1, while Heur_Dist(v_2) is only 5.9. Thus, the first vertex added to S after x is v_2, even though v_1 is closer to x. Consequently, the shortest path to y will be identified on the next iteration of the algorithm, in contrast to Dijkstra's algorithm where all the vertices closer to x than y are finalized before the shortest distance to y is determined.

Figure 3-17 here

The algorithm follows. The utilities and data structures are similar to those for Dijkstra's algorithm. An additional field Heur_Dist is included for each vertex. Reached is implemented as a heap which is ordered with respect to Heur_Dist.

```
Procedure Sedgewick_Vitter (G, x, y)

(* Find the shortest distance from x to y in G *)

var  G: Graph
     v,w,x,y:  1..|v|
     M: Integer constant
     Reached:  Vertex Heap
     Delete, Member, Dijkstra: Boolean function

Set Pred(w) to 0, for each vertex w in V(G)
Set Dist(x) to 0
Set Dist(w) to M (large), for each vertex w <> x in G
Set Heur_Dist(x) to Euclid_Dist(x,y)
Create (Reached), Insert (Reached,x)

while  Delete (Reached, v)  do

  for  each neighbor w of v  do

    if   Pred(w) = 0 (ie, w unreached)

    then Set Dist(w) to Dist(v) + Length(v,w)
         Set Heur_Dist(w) to Dist(w) + Euclid_Dist(w,y)
         Set Pred(w) to v
         if  w = y  then  return
         Insert (Reached, w)

    else if Member(Reached,w) and* Dist(w) > Dist(v)+ Length(v,w)

         then Set Dist(w) to Dist(v) + Length(v,w)
              Set Heur_Dist(w) to Dist(w) + Euclid_Dist(w,y)
```

Figure 3-16, on the other hand, shows the kind of negative cycle that makes the algorithm fail. In this example, and regardless of the order in which the edges are scanned, the distances to v_2 and v_3 are correctly estimated after two iterations. However, the algorithm assigns increasingly lower estimated distances to v_4, v_5, and v_6 as the iterations proceed. These estimates do not correspond to upper bounds on the lengths of shortest paths, as is the case when there are no negative cycles. But, they do have a graphical interpretation. They correspond to increasingly shorter walks from v_1 to vertices on the negative cycle. Not only are the estimated distances for the cycle vertices affected, the distance estimates for any vertices reachable from such a negative cycle are also contaminated. Nonetheless, even under these circumstances the algorithm correctly calculates the shortest distances to vertices reachable from v_1 which are not reachable from any of the negative cycles.

Figure 3-16 here

3-4 EUCLIDEAN SHORTEST PATHS: SEDGEWICK-VITTER HEURISTIC

A *euclidean graph* G(V,E) is a weighted graph embedded in euclidean n-space whose vertices correspond to points in n-space and in which the weight of an edge (u,v) is equal to the euclidean distance between the points u and v. Observe that in the case that G is drawn in the plane, G need not be planar. We will describe a heuristic modification of Dijkstra's algorithm for euclidean graphs that finds shortest paths in better expected (average) time than Dijkstra's algorithm. The worst case performance remains the same.

Let us recall the structure of Dijkstra's algorithm for finding the shortest distance through a directed graph G(V,E) from a given vertex x to a target vertex y. The method extends a search tree T containing a shortest path subtree S until S reaches y. During the search process, the vertices are subdivided into three classes depending on whether they are unreached (not yet in T), reached (in T, but not yet in S), or lie in the shortest path tree S.

The Sedgewick-Vitter algorithm adaption of Dijkstra's algorithm is as follows. We maintain, for each vertex w in Reached, a lower bound, denoted by Heur_Dist(w), on the distance through G from x to y on a shortest path through w. Heur_Dist(w) is defined as the length of the path from x to w through the search tree plus the euclidean distance from w to y. While Dijkstra's algorithm uses the distance through the search tree Dist(u) to determine which vertex u to select as the next vertex to enter S; the Sedgewick-Vitter algorithm selects as the next vertex to enter S that vertex u that minimizes Heur_Dist(u). When the search reaches y, the algorithm terminates with the shortest path to y.

Sedgewick and Vitter have analyzed the algorithm for various classes of random euclidean graphs and shown its average performance is O(|v|), provided the set of reached vertices is implemented as a so-called Fibonacci heap (see Fredman and Tarjan [1987]). In contrast, using a similar implementation, Dijkstra's algorithm attains a worst case performance of O(|E| + |v| log |v|). We refer the reader to Section 3-5 for a detailed treatment of Fibonacci heaps.

At pass 1, the algorithm correctly identifies shortest paths emanating from v containing one edge. Assume by induction that by pass k - 1 the algorithm has determined the shortest paths from v containing k - 1 edges. (Observe that even though the correct k - 1 edge shortest paths have been determined, we do not actually recognize them until the algorithm terminates, which may be many passes later). Let u be a vertex such that the shortest path from v to u, denoted by P(u), contains k edges. Let w be the predecessor of u on P(u). By subpath optimality, the part of P(u) from v to w is a shortest path from v to w, containing k - 1 edges. By induction, this subpath is correctly found before the k^{th} iteration. Therefore, by the end of the k^{th} iteration, Dist(u) has received its correct value, when the edge (w,u) is examined. Since this distance estimate cannot be improved upon, it should not be changed in subsequent passes of the algorithm. Since the only phenomenon that could cause the algorithm to blindly make such a change (and so violate the shortest path interpretation of the estimate) would be due to the effect of a negative cycle, and since these are prohibited, the estimate remains at this optimal value. Consequently, the algorithm correctly calculates the distance to every vertex reachable from v. This completes the proof of the theorem.

THEOREM (PERFORMANCE OF FORD'S ALGORITHM) Let G(V,E) be a weighted digraph containing no cycles of negative weight. Let v be in V(G). Then, Ford's algorithm finds the shortest path from v to every vertex reachable from v in time O(|v||E|).

The proof is trivial. The time performance of the algorithm is O(|v||E|) because, at most, |v| passes are required and each pass examines at most |E| edges.

Ford's algorithm differs from both Dijkstra's and Floyd's algorithms in one interesting respect. If Dijkstra's algorithm is terminated at some stage k before the target vertex is reached, the k nearest vertices found by the algorithm to that point are identifiable from the partial results provided by the algorithm. Similarly, if Floyd's algorithm is terminated at the k^{th} stage, then shortest paths using only the first k vertices internally are identifiable. In contrast, if Ford's algorithm is terminated at the k^{th} stage, although one may know that all cardinality k shortest paths have been found by that point, one can nonetheless not tell which of the distance estimates are final and which are only tentative.

EFFECT OF NEGATIVE CYCLES

The algorithm may or may not operate correctly when the digraph contains negative cycles. It depends on the characteristics of the cycles.

Figure 3-15 shows a digraph containing a negative cycle which is unreachable from v_1. If we apply Ford's algorithm to this graph, with v_1 as the starting vertex, it will correctly calculate all shortest paths starting at v_1. Since none of the vertices on the negative cycle are reachable from v_1, none of the vertices on the negative cycle have their Dist values changed from their initial states. Thus, the algorithm terminates correctly.

Figure 3-15 here

```
Set Dist(u) to M,  for u <> v, for M large

repeat

  Set Ford to True
  Set Pass to Pass + 1

  for every edge (u,w) in G do

    if   Dist(u) + Length(u,w) < Dist(w)

    then Set Dist(w) to Dist(u) + Length(u,w)
         Set Pred(w) to u
         Set Ford    to False

until  Ford  or*  Pass ≥ |v|

End_Function_Ford
```

An example is shown in Figure 3-14 and worked out below. If an edge (u,w) improves the estimated distance to w, we star the revised entries.

		Pass 1		Pass 2		Pass 3	
	Edge(u,w)	Dist(w)	Pred(w)	Dist(w)	Pred(w)	Dist(w)	Pred(w)
(1)	(1,2)	2*	1*	2	1	1	3
(2)	(1,4)	2*	1*	2	1	2	1
(3)	(1,5)	2*	1*	2	1	1	3
(4)	(2,3)	4*	2*	3	2	3	4
(5)	(3,1)	0	0	0	0	0	0
(6)	(3,2)	2	1	1*	3*	1	3
(7)	(3,5)	2	1	1*	3*	1	3
(8)	(4,3)	3*	4*	3	4	3	4

Figure 3-14 here

THEOREM (CORRECTNESS OF FORD'S ALGORITHM) Let G(V,E) be a weighted digraph with real edge weights and containing no cycles of negative weight. Let v be in V(G). Then, Ford's algorithm finds the shortest distances from v to every other vertex in G reachable from v.

The proof is by an induction on the number of edges in a shortest path. The proof has elements in common with the proofs of both Dijkstra's and Floyd's algorithm. We will rely on properties of shortest paths in digraph's containing no negative cycles described in the proof for Floyd's algorithm. They were path optimality, the shortest walk from v to any other vertex u is a path; and subpath optimality, the leading subpaths of a shortest path are shortest paths. Subpath optimality implies that if P is a shortest path from v to u and w is a predecessor of u on that shortest path, the subpath of P from v to w is a shortest path from v to w. Path optimality implies that the shortest walk from, say, v to u can always be chosen to be a shortest path from v to u.

By the end of its k^{th} iteration, the algorithm finds all the shortest paths emanating from v that have at most k edges.

We can also visualize Ford's algorithm as constructing a search tree, like Dijkstra's algorithm. However, the search tree is more volatile than the search tree in Dijkstra's algorithm. As usual, we maintain a predecessor pointer, for each vertex u, that points to the predecessor of u on the current best shortest path P(u) from v to u. The algorithm scans the edge list repeatedly, and, whenever adding an edge (w,u) to an estimated shortest path P(w) leads to a shorter path to u, it makes w the predecessor of u. The algorithm is stopped at any iteration that yields no improvement in any of the estimated distances, or after |v| - 1 iterations, whichever comes first.

ORD'S ALGORITHM

We use an edge list representation for G(V,E).

```
type  Graph = record
                 |v|: Integer Constant
                 |E|: Integer Constant
                 Edges(|E|,2): 1..|v|
                 Length(|E|): Real
                 Dist(|v|): Real
                 Pred(|v|): 0..|v|
              end
```

Edges stores the edge list for G, the pairs (Edges(i,1), Edges(i,2)) being the i^{th} edge of G, i = 1..|E|. Length(i) gives the length of the i^{th} edge, i = 1..|E|. During execution, Dist(i) equals the length of the estimated shortest path to vertex i, and Pred(i) gives the predecessor of i on the path.

We define the procedure Ford (G,v) to return in Dist(u) the shortest distance from v to u. The shortest paths can be extracted by following the predecessor pointers from a given vertex back to v. If a vertex is unreachable from v, Dist is left equal to the large default value M. The procedure fails if changes occur in the distance estimates during the $|v|^{th}$ pass, which can happen only if G has a negative cycle.

```
Function Ford (G, v)

(* Finds the shortest paths from v to every vertex reachable from
   v, and fails if it reaches a negative cycle *)

var  G: Graph
     Pass: 0..|v| - 1
     u,v,w: 1..|v|
     M: Integer Constant
     Ford: Boolean function

Set Pass    to 0
Set Dist(v) to 0
```

be reduced to an equivalent shortest path.

Figure 3-12 here

To prove subpath optimality we argue as follows. Refer to Figure 3-13 for an illustration of the following notation. Let P(i,j) be a shortest path from i to j. Suppose k lies on P(i,j). Then P(i,k) must be a shortest path from i to k. Otherwise, there exists a still shorter path P'(i,k) from i to k. The union P'(i,k) U P(k,j) is a walk from i to j. (It might not be a path, as illustrated in Figure 3-13.) By path optimality such a walk can be reduced to an equal valued path from i to j. But, since P'(i,k) is shorter than P(i,k), this path would be shorter than the presumed shortest path P(i,j) from i to j. It follows by contradiction that

P(i,k) is a shortest path from i to k. This completes the proof of the theorem.

Figure 3-13 here

THEOREM (CORRECTNESS OF FLOYD'S ALGORITHM) Let G(V,E) be a weighted digraph containing no negative cycle. Then, Floyd's algorithm correctly finds all inter-vertex distances.

The proof follows easily from path and subpath optimality. Thus, the relation

$$SD(i,j,k+1) = \min\{SD(i,j,k),\ SD(i,k+1,k) + SD(k+1,j,k)\}$$

merely asserts that a stage k + 1 shortest path either contains k + 1 or does not. If it does not, the optimal stage k + 1 solution is the same as the optimal stage k solution. If it does, then by subpath optimality the two subpaths used at stage k are shortest paths whose values were correctly calculated by stage k. The optimal stage k + 1 path can be a combination of two such paths or a stage k path. By path optimality, we know the combination of the two subpaths is a path. The algorithm determines the correct decision by selecting the best of the two alternatives.

FORD'S ALGORITHM:

VERTEX TO ALL VERTICES

Ford's algorithm finds the shortest paths from a given vertex v to every vertex reachable from v in a weighted digraph G(V,E) with real edge weights and containing no negative cycles. When applied to a digraph containing negative cycles, the algorithm will fail to terminate within |v| - 1 iterations, confirming the presence of a negative cycle.

Ford's algorithm solves the original problem by effectively embedding it in a sequence of subproblems, like Floyd's algorithm, although in this case the subproblems are only implicit. The embedding parameter is the number of edges on the path, and the algorithm finds successively longer shortest paths, that is, shortest paths emanating from v with increasing numbers of edges.

-1*	-1*	0	1	4*	4*	3	4
-2	-2	0	0	1	2	1	4

(e) SD after stage 4. (e') SP after stage 4.

Figure 3-8. Trace of SD and SP for G.

VALIDITY OF FLOYD'S ALGORITHM

he correctness of the algorithm depends on the following structural characteristics of shortest paths in digraphs with no negative cycles.

(1) *Path Optimality*: The shortest walk between a pair of vertices is always reducible to an equal valued shortest path.

(2) *Subpath Optimality*: The subpaths of a shortest path are also shortest paths.

Path Optimality can fail if the graph has negative cycles. Figure 3-9 shows a graph containing a negative cycle where the shortest walk v_1-v_3-v_2-v_3-v_4 has length 1, while the two shortest paths v_1-v_3-v_4 and v_1-v_2-v_3-v_4 have length 2. For this graph, the shortest walk is shorter than the corresponding shortest path. Figure 3-10 shows a graph with a shortest walk v_1-v_3-v_2-v_3-v_4 which has the same length as the shortest path v_1-v_3-v_4 obtained by eliminating the zero weight cycle v_3-v_2-v_3 from the walk.

Figures 3-9 amd 3-10 here

Subpath Optimality can also fail if the graph has a negative cycle. Figure 3-11 shows such a graph; v_1-v_2-v_3 is a shortest path (walk) from v_1 to v_3, but v_1-v_2 is not a shortest path from v_1 to v_2 (v_1-v_3-v_2 is). However, if the graph contains no negative cycles, we can prove the following theorem.

Figure 3-11 here

THEOREM (PATH AND SUBPATH OPTIMALITY) Let G(V,E) be a weighted digraph containing no negative cycle. Then, G satisfies both path and subpath optimality.

To prove path optimality holds, we argue as follows. Observe that a walk can always be reduced to a path by iteratively removing cycles from the walk. The process is concretely illustrated in Figure 3-12. Figures 3-12a and 12b show a walk from v_1 to v_5 with the multiply traversed edge v_2-v_3. The walk is v_1-v_2-v_3-v_4-v_2-v_3-v_5. Figure 3-12c shows the walk with the cycle v_2-v_3-v_4-v_2 eliminated. Each multiple occurrence of an edge on the cycle is removed from the walk. The resulting reduced walk is a path. The same process works in general. Consequently, if every cycle has nonnegative weight, every walk can be reduced to a path of at most equal weight. Therefore, a shortest walk can

0	2	2	M
M	0	M	2
M	M	0	1
-2	-2	M	0

(a) Initial SD Matrix.

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

(a')Initial SP Matrix.

0	2	2	M
M	0	M	2
M	M	0	1
-2	-2	0*	0

(b) SD after stage 1.

1	2	3	4
1	2	3	4
1	2	3	4
1	2	1*	4

(b') SP after stage 1.

0	2	2	4*
M	0	M	2
M	M	0	1
-2	-2	0	0

(c) SD after stage 2.

1	2	3	2*
1	2	3	4
1	2	3	4
1	2	1	4

(c') SP after stage 2.

0	2	2	3*
M	0	M	2
M	M	0	1
-2	-2	0	0

(d) SD after stage 3.

1	3*	3	3
1	2	3	2
1	2	3	4
1	2	1	4

(d') SP after stage 3.

0	1*	2	3
0*	0	2*	2
4*	2	4*	2

```
Procedure Extract_Shortest_Path (SP, |v|, i, j, P)

(* Returns the shortest path from i to j in P *)

var  |v|: Integer Constant
     SP(1..|v|, 1..|v|,0..|v|): 1..|v|
     P(|v|): 0..|v|
     i, j, Next: 1..|v|
     Nextcount: Integer function

Set P(Nextcount) to i
Set k to i

while k <> j do  Set k to SP(k, j, |v|)
                 Set P(Nextcount) to k

End_Procedure_Extract_Shortest_Path
```

PERFORMANCE OF FLOYD'S ALGORITHM

The time performance of the algorithm is $O(|v|^3)$. For clarity, we have separated the data from different stages so that the space requirements are apparently $O(|v|^3)$. However, the values at stage k obviously depend only on the values from stage k - 1; so we actually require only $O(|v|^2)$ memory (such as two |v|x|v| arrays whose roles alternate.) Indeed, the computations can even be done in place. Observe that SD(i,j,k) depends on SD(i,j,k-1), SD(i,k,k-1), and SD(k,j,k-1). The last two terms remain constant at stage k, because vertex k is not an internal vertex on paths from i to k or k to j. Furthermore, no other stage k term except SD(i,j,k) depends on SD(i,j,k-1). Consequently, the storage for SD(i,j,k) and SD(i,j,k-1) can be overlaid. Thus, the computations can be done in place, and the algorithm needs to make no lexical distinction between the terms from the various stages. We can rewrite floyd's algorithm in an in-place form by merely replacing every occurrence of SD(i,j,k) and SP(i,j,k) in Floyd or Floyd_paths by an occurrence of SD(i,j) and SP(i,j).

An example of the algorithm is shown in Figures 3-7 and 3-8. Recall that neither row nor column k changes at stage k, and the diagonals entries never change unless the digraph has a negative cycle. When the graph contains negative cycles, an entry SD(i,i) will eventually become negative if i lies on such a cycle. If this occurs it indicates the graph fails to satisfy the conditions of applicability of Floyd's algorithm. The graph in Figure 3-7 has no negative cycles. The initial SD and SP matrices are shown in Figure 3-8a. The first iteration allows v_1 as an intermediate vertex, with the results shown in Figure 3-8b. We have starred SD values that change at a given stage.

Figure 3-7 here

SP(i,j,k) as the successor of vertex i on the stage k shortest path from vertex i to vertex j.

See Figure 3-5. For example, in Figure 3-6, SP(1,2,4) is 3, because the stage 4 shortest path from v_1 to v_2 is v_1-v_3-v_4-v_2. SP (*,*,|v|) stores in implicit form the $|v|^2$ shortest paths between the $|v|^2$ pairs of vertices of G. This economical representation of $|v|^2$ paths is interesting in itself. Since there are $|v|^2$ paths, each containing as many as O(|v|) vertices, it seems that we need $O(|v|^3)$ storage to represent the paths. SP represents the paths in $O(|v|^2)$ storage, though at the expense of O(|v|) processing time to extract the vertices of a particular path.

Figure 3-5 and 3-6 here

We can calculate SP by a simple modification of Floyd's algorithm. If P(i,j,k) contains k, the part of P(i,j,k) from i to k is just P(i,k,k - 1). In that case, the successor of i on P(i,j,k) (namely SP(i,j,k)) is the same as the successor of i on P(i,k,k - 1) (namely SP(i,k,k - 1)). If P(i,j,k) does not contain k, then S(i,j,k - 1) does not change. The modified procedure follows.

```
Procedure Floyd_Paths (G,SD,SP)

(* Returns the shortest distance i to j distance in  SD(i,j,|v|)
   and the shortest paths between all pairs of vertices in G in
   SP(*,*,|v|) *)

var  G: Graph
     i,j,k: 1..|v|
     SD(1..|v|, 1..|v|, 0..|v|): Real
     SP(1..|v|, 1..|v|, 0..|v|): 1..|v|

for i, j = 1..|v|  do:  Set SD(i,j,0) to Dist(i,j)
                        Set SP(i,j,0) to j

for k = 1 .. |v|  do
 for i = 1 .. |v|  do
  for j = 1 .. |v|  do

    if    SD(i,j,k - 1) < SD(i,k,k - 1) + SD(k,j,k - 1)

    then  Set SD(i,j,k) to SD(i,j,k - 1)
          Set SP(i,j,k) to SP(i,j,k - 1)

    else  Set SD(i,j,k) to SD(i,k,k - 1) + SD(k,j,k - 1)
          Set SP(i,j,k) to SP(i,k,k - 1)

End_Procedure_Floyd_Paths
```

We can use the procedure Extract_Shortest_Path to extract the shortest paths from SP. The procedure uses an auto-incrementing integer function Nextcount which is initially 1. P is initially identically 0.

Let P(i, j, k) denote a shortest path from i to j whose internal vertices have indices on 0..k, and let SD(i,j,k) denote the length of P(i,j,k). Then

  SD(i,j,k + 1) = min{SD(i,j,k), SD(i,k + 1,k) + SD(k + 1,j,k)}.

The validity of this procedure is less obvious than it seems, and we will establish it later. Refer to Figure 3-4 for an illustration. The rule merely says to always choose the better of the two possible ways of getting from i to j. The stage k path, P(i,j,k), of length SD(i,j,k) is the better path if k + 1 does not lie on a shortest stage k + 1 path from i to j; while the path through k + 1 and of length SD(i, k + 1, k) + SD(k + 1, j, k) is the better path if k + 1 does lie on a stage k + 1 shortest path from i to j.

Figure 3-4 here

FLOYD'S ALGORITHM

The statement of the algorithm is simple. We represent the graph by its distance matrix Dist, where Dist(i,j) gives the length of the (i,j) edge, where diagonal entries are set to zero, and if there is no edge between (for i <> j), Dist(i,j) is set to some large positive integer M. The type graph G is

```
type  Graph = record
                |v|: Integer Constant
                Dist(|v|,|v|)
              end
```

We store the stage k shortest distances in a |v|x|v|x(|v| + 1) array SD(i,j,k). Later, we will observe that the algorithm can be done in place. The ordering of the loop indices in the algorithm is important. The outermost of the nested **for** loops must be indexed by the stage k. This ensures all the stage k - 1 values are available before the stage k computation needs them.

```
Procedure Floyd (G,SD)

(* Returns shortest distance between i and j in SD(i,j,|v|) *)

var  G: Graph
     i,j,k: 1..|v|
     SD(1..|v|, 1..|v|, 0..|v|): Real

for i, j = 1..|v| do  Set SD(i,j,0) to Dist(i,j)

for k = 1 .. |v|  do
 for i = 1 .. |v|  do
   for j = 1 .. |v|  do
    SD(i,j,k) = min {SD(i,j,k - 1),SD(i,k,k - 1) + SD(k,j,k - 1)}

End_Procedure_Floyd
```

The algorithm as given does not calculate the shortest paths, but a simple refinement lets us both calculate them and store them very compactly. Define

```
    if   w unreached

    then Set Dist(w)  to Dist(v) + Length(v,w)
         Set Pred(w)  to v
         Insert (Reached, w)

    else if Member(Reached,w)
                 and*  Dist(w) > Dist(v) + Length(v,w)

         then Set Dist(w)  to Dist(v) + Length(v,w)
              Set Pred(w)  to v
              Change (Reached,w)

Set Dijkstra = ( v = y )

End_Function_Dijkstra
```

FLOYD'S ALGORITHM:

VERTEX TO ALL VERTEX PAIRS

Dijkstra's algorithm requires the edge weights of a digraph G(V,E) to be positive. Floyd's algorithm allows negative weights as well, but requires the digraph to be free of any cycle whose total edge weight is negative (a so-called *negative cycle*). The algorithm finds shortest paths between every pair of vertices in G and detects negative cycles if they occur. It also provides a compact matrix representation for the $|v|^2$ shortest paths found.

FLOYD'S METHOD

The idea is to embed the original shortest paths problem in a parametrized sequence of subproblems, and then use induction (or dynamic programming) to solve the subproblems. The subproblems are defined as follows. Assume the vertices of V(G) are indexed from 1..|v|, and define an *internal vertex* of a path as any vertex on the path which is not an endpoint of the path. The k^{th} subproblem (for k = 0, ..., |v|) is find the shortest paths between every pair of vertices in G(V,E) where the internal vertices of the paths are chosen from vertices on 0..k. For k = 0, there are no internal vertices on the paths, and so the shortest paths are just the given vertex to vertex edges of the input graph, where we follow the convention that a nonexistent edge is treated as if it were an edge of some large (infinite) weight. When k = |v|, the constraint is vacuous and the subproblem corresponds to the original problem.

At stage k, we have the shortest paths and distances between every pair of vertices, where the internal vertices have indices on 0..k. We progress from the solutions at stage k to the solutions at stage k + 1 by allowing k + 1 as an intermediate vertex at stage k + 1 only if a stage k + 1 path through k + 1 improves the current shortest estimated distance. The decision is implemented by merely considering the alternative (use k + 1 or do not use k + 1) for every pair of vertices in G.

or Change operations per selected vertex v. Each of these takes O(log |v|) steps, and so (2) requires O(deg(v)log(|v|)) steps or, summed over all iterations of (2), O(|E| log |v|) steps. Therefore, the total work is O(|v| log(|v|) + |E| log |v|), which is just O(|E| log |v|). This is an improvement on the earlier estimate ($O(|v|^2)$) when |E| is O(|v|), but inferior if |E| is $O(|v|^2)$.

Using the standard array representation for a heap, augmented to include information needed for our algorithm, we can represent a heap as follows:

```
type Vertex Heap = record
                     H(|v|):   0..|v|
                     Last:     0..|v|
                     Ptr(|v|): 0..|v|
                   end
```

The indices of the vertices in the heap are stored in H. We assume the heap is ordered on the Dist values of its vertices. In order to minimize redundancy, we keep these values only in the linear array representation of the graph G, where they are directly accessible from the vertex indices. Last gives the position in H of the last element of H. Ptr is the auxiliary array that provides direct access into H on the basis of the vertex index alone. It also lets us test for heap membership in O(1) time by testing whether Ptr(w) is 0 or not.

Using this data type, we can restate Dijkstra's algorithm as follows. We will use the basic heap functions. Create (Reached) initializes the heap Reached. Insert (Reached, w) adds the vertex w to the heap Reached. Delete (Reached, v) deletes the smallest element of the heap, returning its index in v, and fails if the heap is empty. Member (Reached,w) succeeds if w is in Reached, and fails otherwise. Change(Reached, w) is used if Dist(w) is altered, and appropriately changes the position of w in the heap.

```
Function Dijkstra (G,x,y)

(* Returns the shortest distance from x to y in Dist(y), and
   the shortest path in Pred fields starting at y, or fails *)

var  G:  Graph
     v,w,x,y:  1..|v|
     M: Integer constant
     Reached:  Vertex Heap
     Delete, Member, Dijkstra: Boolean function

Create (Reached), Insert (Reached,x)
Set Pred(w)   to 0, for each vertex w in G
Set Dist(x)   to 0
Set Dist(w)   to M (large), for each vertex w <> x in G

while  Delete (Reached, v)  and*  v <> y   do

  for each neighbor w of v do
```

ANALYSIS OF DIJKSTRA'S ALGORITHM AND ENHANCEMENT

The computationally critical steps in the algorithm are

(1) Finding the next nearest vertex v in Reached, and
(2) Updating the fields of the vertices adjacent to v.

Each step is iterated as many as |v| times, since the target vertex may be as far as |v| vertices away. The simplest way to implement step (1) is to linearly scan Reached to select the minimum. For step (2), as many as O(|v|) neighbors of the selected vertex must be updated, taking O(|v|) time. Accounting for iteration, the whole procedure then takes $O(|v|^2)$ steps.

We can try to refine this analysis by observing that there are only deg(v) neighbors at each selected minimum vertex v; so the total number of actions under (2) is at most O(|E|). However, the total number of steps in (1) is still $O(|v|^2)$; so this yields no overall improvement in the performance estimate. Nonetheless, this analysis suggests (1) may be a bottleneck; so we may be able to improve performance by doing (1) more efficiently.

We can speed up (1) by representing Reached as a heap of vertices ordered on the value of their Dist field. (We only need to keep the index and value of Dist for each vertex in the heap.) Recall that a heap is a data structure that facilitates both sorting and minimum selection. We will assume familiarity with the definition of a heap. The basic *heap operations* are

(H1) Create (a heap),
(H2) Find_min (find the minimum element in the heap),
(H3) Delete (the least element in the heap and restore the heap),
(H4) Insert (a new element into a heap),
(H5) Member (test if an element is in a heap), and
(H6) Change (the value and if necessary the position of an existing heap element).

We will analyze the performance of heaps, particularly the Create and Delete operations, in detail in Section 4-3. Here, let us recall that we can delete the smallest element of a |v| element heap in O(log(|v|) time or add a new element to a heap in O(log |v|) time.

We can also Change the value of an existing heap element in O(log |v|) time, but this requires some care. To keep the Change operation efficient, we have to be able to directly access the heap element whose value is to be changed. We can do this by keeping an auxiliary array, Ptr, of pointers into the heap, where Ptr(i) gives the current position in the heap of the i^{th} vertex of G. We can then implement Change much like an ordinary heap insertion. Thus, if the new changed value of an element is smaller than the value of its heap parent, the changed element moves up in the heap; otherwise, it is sifted down the heap in a standard manner (refer to Section 4-3).

If we represent Reached using a heap rather than a set, the operations (1), each of which entails a single heap delete operation, take O(|v|log |v|) time over all iterations. Each occurrence of step (2) requires deg(v) Insert

```
Set Reached   to {x}
Set Pred(w)   to 0, for each vertex w in G
Set Dist(x)   to 0
Set Dist(w)   to M (Large), for each vertex w <> x in G

while  Getmin(v) and* v <> y   do

  for each neighbor w of v do

     if w unreached

     then Add w to Reached
          Set Dist(w)  to Dist(v) + Length(v,w)
          Set Pred(w)  to v

     else if w in Reached  and  Dist(w) > Dist(v) + Length(v,w)

          then Set Dist(w)  to Dist(v) + Length(v,w)
               Set Pred(w)  to v

Set Dijkstra = ( v = y )

End_Function_Dijkstra
```

A weighted digraph G is shown in Figure 3-2. Figure 3-3 traces the development of the search tree for G rooted at v_1. The shortest path or distance to v_4 is sought. The shortest path tree edges are labelled S. The search tree edges are labelled S or R.

Figures 3-2 and 3-3 here

THEOREM (VALIDITY OF DIJKSTRA ALGORITHM) If G(V,E) is a network with nonnegative edge weights, and x and y are in V(G), and y is reachable from x, then Dijkstra's algorithm finds a shortest path from x to y.

e can prove the theorem by induction. Thus, assume that the first k iterations of the algorithm correctly identify the k nearest vertices to x. We will show that the (k + 1)st iteration correctly identifies the (k + 1)st nearest vertex to x.

Let y be the (k + 1)st nearest vertex to x. Let P be a shortest path from x to y. Let w be the predecessor of y on P. We assume that all edges have positive weight.

The part of P from x to w is a shortest path; y lies one (positive) edge past the end of this path. Therefore, w must be closer to x than y, that is, w must be among the k nearest vertices to x. By induction, the algorithm correctly identifies a shortest path to w within the first k stages. Therefore, by the end of stage k, Dist(y) is set to Dist(w) + Length(w,y), which is the true shortest distance from x to y. By the beginning of stage k + 1 the algorithm recognizes y as the next nearest vertex to x and has already calculated its correct distance from x and a shortest path realizing that distance. This completes the proof by induction.

The field Dist(v) will contain the current estimated distance to v from x calculated by the algorithm. Pred(v) gives the index of the search tree predecessor of v. The remaining fields in Vertex have the usual interpretation.

Although it is instructive to think of the algorithm in terms of a search tree, we do not need to explicitly maintain the tree as a separate data object. Instead, it is embedded in the predecessor pointers. Observe that although each search tree vertex has several search tree successor vertices, it has only a single search tree predecessor vertex; so a single predecessor pointer suffices at each vertex. After the algorithm finds the shortest path from a vertex x to a vertex y, we can read off the shortest path, in reverse order, by following the predecessor pointers from y back to x.

Although the algorithm does not explicitly maintain the search tree, we do explicitly maintain a set, Reached, consisting of vertices which already lie in the search tree, but are not yet in the shortest path subtree. We represent Reached in the algorithm as a set, but later on we will suggest a more efficient data structure for Reached.

Each edge in the graph has a defining record of the following form

```
type   Edge = record
                 Length: Real
                 Neighboring vertex: 1..|v|
                 Successor: Edge pointer
              end
```

The field Length gives the length of the edge. Neighboring vertex identifies the other endpoint of the edge.

The function Dijkstra (G,x,y) either returns a shortest path from x to y, as embedded in the predecessor pointers starting at y, or fails because y is not reachable from x. The algorithm uses a function Getmin(v) which returns in v the vertex in Reached with the minimum value of Dist(v), and removes v from Reached, effectively placing v in the implicit shortest path subtree. Getmin fails if Reached is empty, and succeeds otherwise. The operator and* denotes conditional conjunction: the second operand is not evaluated if the first operand fails.

```
Function Dijkstra (G,x,y)

(* Returns the shortest distance from x to y in Dist(y), and
   the shortest path starting at y using the Pred fields, or
   fails. *)

var  G:  Graph
     v,w,x,y:  1..|v|
     M: Integer Constant
     Reached: Set of 1..|v|
     Dijkstra, Getmin: Boolean Function
```

procedure which explores the graph in a tree-like manner. This search induces a subdigraph of G called a *search tree*, which in turn contains a distinguished subtree called a *shortest path subtree* which has the property that the shortest distances to all its vertices are known. The search process is continued until the target vertex y is incorporated in the shortest path subtree.

At each phase of the search process, a new vertex v lying in the search tree but not in the shortest path subtree is incorporated in the shortest path subtree, and the search tree is then extended from v to its neighbors. For each such neighbor w of v, we make w point to v as its search tree predecessor if either w was not previously visited, or w was already visited, but the path through the tree from x to w is longer than the path through the tree from x to v plus the edge from v to w.

Initially, the search tree fans out from x to its immediate neighbors, defining a star at x. Observe that the second nearest vertex to x is necessarily that vertex in the star which lies at the end of a shortest edge incident with x. Trivial though this observation is, it already depends critically on the positivity of the edge weights. Furthermore, we can also discern at this point the distinction between the search tree, which includes all those vertices so far reached by the search procedure (which is currently just the star), and the shortest path subtree, consisting of x and its closest neighbor.

After k stages, the shortest path subtree of the search tree contains the k nearest vertices to x. The path through this tree from x to any of its vertices is a shortest path. On the other hand, the tree path from x to vertices in the search tree, which are not yet in the shortest path subtree, are estimated shortest paths only, and are subject to revision. We will show later that the (k + 1)-st nearest vertex to x is precisely that vertex v in the stage k search tree whose estimated distance from x is a minimum among all those vertices not yet in the shortest path tree.

An example search tree is shown in Figure 3-1. The edges labelled R (Reached) lead to vertices which are in the search tree, but not yet in the shortest path tree. The edges labelled S lead to vertices which are already in the shortest path subtree. Any vertices not shown correspond to vertices as yet unreached by the search. Any edges not shown are either edges as yet unexplored, or explored edges known to not lie on shortest paths.

Figure 3-1 here

DIJKSTRA'S ALGORITHM

In our statement of Dijkstra's algorithm, we will use the standard linear array representation for a graph (refer to Chapter 1). The definition of the type Vertex is slightly different.

```
type   Vertex = record
                  Dist: Real
                  Pred: 0..|v|
                  Positional Pointer: Edge pointer
                  Successor: Edge pointer
                end
```

SHORTEST PATHS

3-1 DIJKSTRA'S ALGORITHM: VERTEX TO VERTEX

A *weighted digraph* (or *network*) is a digraph G(V,E) with real valued weights or lengths assigned to each edge. Equivalently, a weighted digraph is a triple (V,E,w) where V and E have the usual interpretation and w is a function that maps the elements of E into the reals. The *length* of a path in a weighted digraph is the sum of the lengths of the edges on the path. A *shortest path* between a pair of vertices x and y in a weighted digraph is a path from x to y of least length. The *distance* from x to y is defined as the length of a shortest path from x to y. The following model illustrates the application of shortest paths.

MODEL: VISIBILITY GRAPHS

Consider the problem of finding shortest paths through a planar region strewn with obstacles. Specifically, let x and y be a pair of points in the plane, and suppose we want to find the shortest path between x and y that avoids the interiors of a set of polygonal regions R. We can model this as a shortest path problem on a weighted graph by introducing the *visibility graph* $G_R(V,E)$ associated with the configuration. Thus, let Vert(R) denote the vertices of the polygonal obstacles of R. Then, set $V(G_R)$ = Vert(R) U {x, y}. We will say a vertex u in $V(G_R)$ is *visible* from a vertex v in $V(G_R)$ if the line segment uv cuts the interiors of none of the regions of R. In the case that u is visible from v, we include the line segment (u,v) as an edge of $E(G_R)$, and let the euclidean length of the edge (u,v) be the weight of (u,v). Then, the shortest obstacle avoiding path from x to y is just the shortest path between x and y in G_R.

DIJKSTRA'S METHOD

Dijkstra's algorithm solves the following problem. Let G(V,E) be a weighted digraph all of whose edge weights are positive, and let x and y be a pair of vertices in G. Find the shortest path from x to y in G and its length, or show there is none. The algorithm uses a search tree technique and is based on the observation that the k^{th} nearest vertex to a given vertex x is the neighbor of one of the j^{th} nearest vertices to x, for some $j < k$. It follows that if we can find the j^{th} nearest vertex to x, for every $j < k$, then we can easily find the k^{th} nearest vertex to x. For example, let Near(j) denote the j^{th} nearest vertex to x, let Dist(u) denote the distance from x to any vertex u, and let Length(u,w) denote the length of the edge from u to any neighbor w of u. Then, the k^{th} nearest vertex to x is that vertex v that minimizes

$$\text{Dist(Near(j))} + \text{Length (Near(j),v)},$$

where the minimum is taken over all $j < k$. Thus, to find the distance to y, we first inductively find the distances to all vertices closer to x than y.

The successively more distant vertices from x are found using a search