Data Structures and Algorithms in Java™

Sixth Edition

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Michael H. Goldwasser
Department of Mathematics and Computer Science
Saint Louis University

Instructor's Solutions Manual

WILEY

Chapter

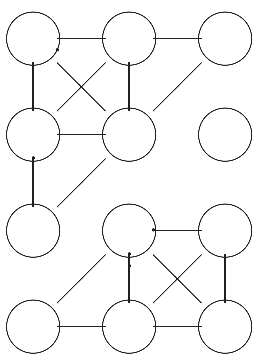
14 Graph Algorithms

Hints and Solutions

Reinforcement

R-14.1) Hint Use the drawing style of the book for the figure.

R-14.1) Solution



R-14.2) Hint Recall that an n-vertex component of a simple undirected graph can have no more than $n(n-1)/2$ edges.

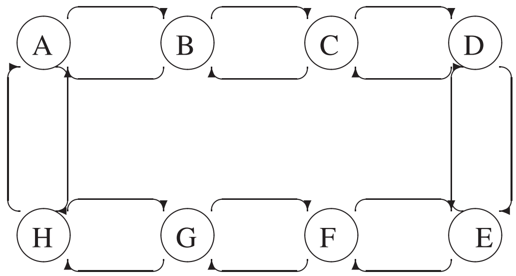
R-14.2) Solution If G has 12 vertices and 3 connected components, then the maximum number of edges it can have is 45. This would be a fully connected component with 10 vertices (thus having 45 edges) and two connected components each with a single vertex.

R-14.3) Hint Fix a canonical ordering of the nodes.

R-14.4) Hint Fix a canonical ordering of the nodes.

R-14.5) Hint When drawing the Euler tour notice that every in has an out.

R-14.5) Solution



The Euler tour is $A \to B \to C \to D \to E \to F \to G \to H \to A \to H \to G \to F \to E \to D \to C \to B \to A$

R-14.6) Hint Recall the properties of inserting and removing elements in a doubly linked list.

R-14.6) Solution Inserting a vertex runs in $O(1)$ time since it is simply inserting an element into a doubly linked list. To remove a vertex, on the other hand, we must inspect all edges. Clearly this will take $O(m)$ time.

R-14.7) Hint Don't forget to update the edge list E.

R-14.8) Hint Don't forget to update the edge list E.

R-14.9) Hint How can the edges method be implemented?

R-14.10) Hint How can the edges method be implemented?

R-14.11) Hint The point of this exercise is to explore the trade-offs of the various representations. If you are having trouble, please review these trade-offs.

R-14.11) Solution

a. The adjacency list structure is preferable. Indeed, the adjacency matrix structure wastes a lot of space. It allocates entries for 100,000,000 edges while the graph has only 20,000 edges.
b. In general, both structures work well in this case. Regarding the space requirement, there is no clear winner. Note that the exact space usage of the two structures depends on the implementation details. The adjacency matrix structure is much better for operation getEdge, while the adjacency list structure is much better for operations insertVertex and removeVertex.
c. The adjacency matrix structure is preferable. Indeed, it supports operation getEdge in $O(1)$ time, irrespectively of the number of vertices or edges.

R-14.12) Hint Use a pencil and paper.

R-14.13) Hint Review the pseudocode while observing the different running times needed to implement each step when the graph is represented with an adjacency matrix.

R-14.14) Hint Work out an example with a complete graph of six vertices.

R-14.14) Solution The depth-first search tree of a complete graph is a path.

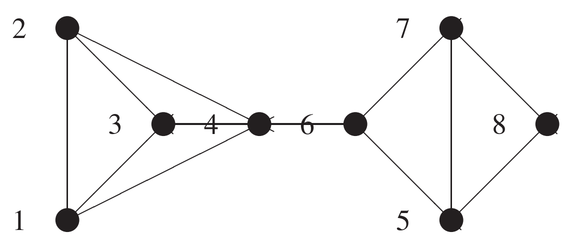
R-14.15) Hint Work out an example with a complete graph of six vertices.

R-14.15) Solution The breadth-first search tree of a complete graph is a star, that is, a rooted tree having all external-node children.

R-14.16) Hint Follow the algorithm descriptions and drawing styles given in the book.

R-14.16) Solution

a. A drawing of graph G is given below:



b. A DFS traversal starting at vertex 1 visits the vertices in the following order: 1, 2, 3, 4, 6, 5, 7, 8.
c. A BFS traversal starting at vertex 1 visits the vertices in the following order: 1, 2, 3, 4, 6, 5, 7, 8.

R-14.17) Hint Model the courses and their prerequisites as a directed graph.

R-14.17) Solution The topological sorting algorithm can help us solve this problem.

- Build a directed graph to represent the course prerequisite requirements. The nine courses are vertices in the directed graph. If a course A is a prerequisite for another course B, the directed graph has a directed edge from A to B.
- Apply the topological sorting algorithm on this directed graph. The result is one possible sequence of courses for Bob.

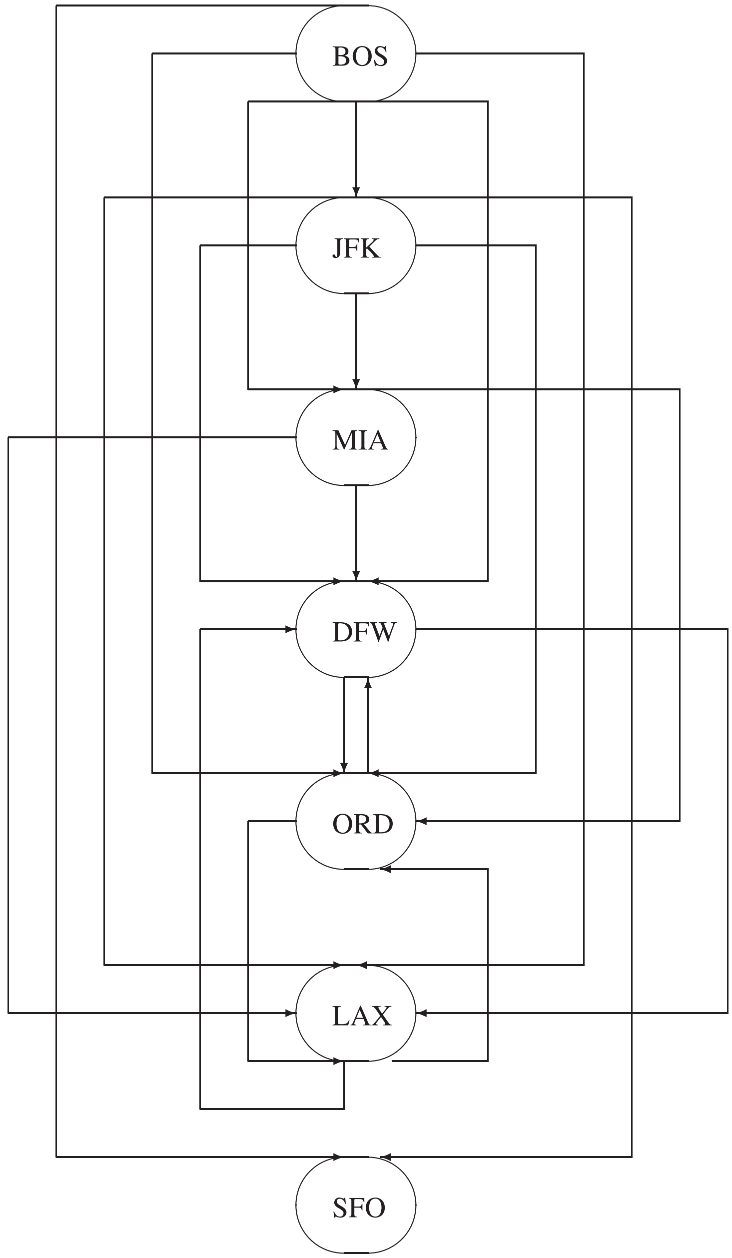
For the given set of prerequisites, the solution is not unique. For example, one possible solution is LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, LA169. Another solution is LA15, LA16, LA127, LA31, LA32, LA169, LA22, LA126, LA141.

R-14.18) Hint Use the transitive closure algorithm given in the book.

R-14.18) Solution BOS, JFK, MIA, ORD, DFW, SFO, LAX

R-14.19) Hint Mimic the drawing style used in the book.

R-14.19) Solution



R-14.20) Hint If you wish, you may run our implementation to determine the order.

R-14.20) Solution The edges are added as follows:
BOS $\rightarrow$ SFO via JFK,

BOS → DFW via JFK,
MIA → ORD via LAX,
JFK → LAX via MIA,
JFK → ORD via MIA,
BOS → LAX via MIA,
BOS → ORD via MIA,
LAX → DFW via ORD,
LAX → SFO via DFW,
MIA → SFO via DFW,
ORD → LAX via DFW,
and ORD → SFO via DFW

R-14.21) Hint Are there any pairs of vertices that are not related?

R-14.21) Solution There are $n(n-1)/2$ in the transitive closure of a graph that consists of a simple directed path of n vertices (including the original edges).

R-14.22) Hint Design a counting scheme that charges each level in T with the transitive closure edges that go from that level to lower levels.

R-14.22) Solution Let us use a counting scheme that charges each level in T with the transitive closure edges that go from that level to lower levels. Note that every node in T will have transitive closure edges to all of its descendants. Thus, a node v will be the origin of as many transitive closure edges as the size of the subtree rooted at v (minus one, for v itself, but let us ignore this fact since we just have to prove an upper bound). In other words, each level of T contributes at most n edges to the transitive closure. Thus, there are $O(n \log n)$ edges in the transitive closure.

R-14.23) Hint Using the drawing style given in the book.

R-14.24) Hint The essential property of course is to observe the edge directions. So work out the pseudocode with this in mind.

R-14.25) Hint Use the drawing style given in the book, and show the order in which the edges are added to the MST.

R-14.26) Hint Use the drawing style given in the book, and show the order in which the edges are added to the MST.

R-14.27) Hint Use an MST algorithm.

R-14.28) Hint The black lines represent unexplored edges. What about the others?

R-14.29) Hint Edges in the DFS tree are represented by solid lines. What about the others?

R-14.29) Solution Edges in the DFS tree are represented by solid lines. Back edges are shown with dashed lines. Forward and cross edges are shown with dashed black lines.

R-14.30) Hint The black lines represent unexplored edges. What about the others?

R-14.30) Solution The black lines represent unexplored edges. The blue lines represent edges leading to the discovery of a new vertex. Dashed lines represent cross edges. Thick lines represent the discovery edges at a specific depth.

R-14.31) Hint Solid black lines are edges in the original input graph. What about the others?

R-14.32) Hint Traversed edges are shown with dashed blue lines. What about the others?

R-14.32) Solution Traversed edges are shown with dashed blue lines. Solid black lines are untraversed edges. Thick lines show the edges traversed in the current iteration.

R-14.33) Hint Solid black edges have not been relaxed yet. What about the others?

R-14.34) Hint Edges being considered in the current iteration for the MST are shown in thick blue lines. What about the others?

R-14.34) Solution Edges being considered in the current iteration for the MST are shown in thick blue lines. Dashed lines represent rejected edges. The cloud represents all the vertices currently in the MST.

R-14.35) Hint Edges being considered in the current iteration for the MST are shown in thick blue lines. What about the others?

R-14.35) Solution Edges being considered in the current iteration for the MST are shown in thick blue lines. Dashed lines represent rejected edges. The cloud represents all the vertices currently in the MST.

R-14.36) Hint Note how much time is spent in each step of George's algorithm.

Creativity

C-14.37) Hint Make sure to remove the edge from the adjacency map for each incident vertex.

C-14.38) Hint Use a data structure described in this book.

C-14.39) Hint Suppose that such a nontree edge is a cross edge, and argue based upon the order the DFS visits the end vertices of this edge.

C-14.40) Hint The DFS method must be changed so that the repetition stops when discovering v.

C-14.41) Hint There has to be a good traversal that does this.

C-14.42) Hint Maintain a set of all vertices upon which a recursive call to DFS is currently active.

C-14.43) Hint Modify the DFSComplete method in order to tag each vertex with a component number when it is first discovered.

C-14.44) Hint The solution involves a traversal discussed in this chapter.

C-14.44) Solution Do a DFS of the maze, treating the branch points as vertices and the corridors as edges. If there are no back edges and every part of the maze is reachable from the start, including the finish, the maze is constructed correctly.

C-14.45) Hint Number the vertices 0 to $n-1$.

C-14.45) Solution Number the vertices 0 to $n-1$. Now add an edge from vertex i to vertex $(i+1)$ mod n, if that edge does not already exist. This connects all the vertices in a cycle, which is itself biconnected.

C-14.46) Hint Suppose there is a forward edge or back edge and show why it would not be a nontree edge.

C-14.47) Hint Suppose there is such an edge and show why it would not be a nontree edge.

C-14.48) Hint Justify this by induction on the length of a shortest path from the start vertex.

C-14.49) Hint Take each part in turn and use induction or contradiction as justification techniques.

C-14.50) Hint Starting with the source vertex in the queue, repeatedly remove the vertex from the front of the queue and insert any of its unvisited neighbors to the back of the queue.

C-14.51) Hint Do a BFS search in G.

C-14.51) Solution Construct a BFS search tree T for G. Place the vertices on even levels in X and the ones in odd levels in Y. Now double-check that there are no edges between a pair of vertices in X or a pair of vertices in Y. This algorithm runs in $O(n+m)$ time.

C-14.52) Hint Start out greedy and patch in the places this approach misses.

C-14.53) Hint Perform "baby" searches from each station.

C-14.53) Solution For each vertex v, perform a modified BFS traversal starting a v that stops as soon as level 4 is computed.

Alternatively, for each vertex v, call method DFS$(v,4)$ given below, which is a modified DFS traversal starting at v:

DFS(v,i)

  if $i > 0$ and v is not marked **then**
    Mark v.
    Print v.
    for all vertices w adjacent to v **do**
      DFS$(w, i-1)$.

C-14.54) Hint Try to compute the *diameter* of the tree T by a pruning procedure, starting at the leaves (external nodes).

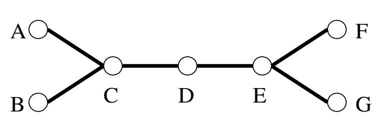
C-14.54) Solution We can compute the *diameter* of the tree T by a pruning procedure, starting at the leaves (external nodes).

a. Remove all leaves of T. Let the remaining tree be T_1.
b. Remove all leaves of T_1. Let the remaining tree be T_2.
c. Repeat the "remove" operation as follows: Remove all leaves of T_i. Let remaining tree be T_{i+1}.
d. When the remaining tree has only one node or two nodes, stop! Suppose now the remaining tree is T_k.
e. If T_k has only one node, that is the center of T. The *diameter* of T is $2k$.
f. If T_k has two nodes, either can be the center of T. The *diameter* of T is $2k+1$.

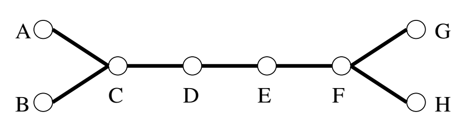
C-14.55) Hint Perform a traversal of T to compute distances to leaves.

C-14.55) Solution

a. (a) Remove all leaves of T. Let the remaining tree be T_1.
   (b) Remove all leaves of T_1. Let the remaining tree be T_2.
   (c) Repeat the "remove" operation as follows: Remove all leaves of T_i. Let remaining tree be T_{i+1}.
   (d) Once the remaining tree has only one node or two nodes, stop! Suppose now the remaining tree is T_k.
   (e) If T_k has only one node, that is the center of T. The *eccentricity* of the center node is k.
   (f) If T_k has two nodes, either can be the center of T. The *eccentricity* of the center node is $k+1$.
b. No! Not always unique. It's possible that the remaining tree has two nodes. We don't like to remove the leaves of a two-node tree (there will be nothing left!). You can try the following two trees. The center of the first tree is D with *eccentricity* 2. The center of the second tree is either D or E with *eccentricity* 3.



Tree 1



Tree 2

C-14.56) Hint What is the in-degree of the vertex labeled i in a compact graph?

C-14.57) Hint Change the function to be optimized, but keep it indexed in terms of the a fixed listing of vertices.

C-14.58) Hint Think about a shortest path with negative edges.

C-14.59) Hint Be greedy.

C-14.59) Solution

Initialize W to V (as a copy of all the vertices)
while W is not empty
    Add a vertex w from W to I.
    Remove from W all other vertices adjacent to w.

The running time of this method is $O(n+m)$.

C-14.60) Hint Give G lots of edges and make every edge count for lots updates in the heap.

C-14.61) Hint Examine closely the justification for why Dijkstra's algorithm works correctly and note the place where it breaks if negative-weight edges.

C-14.62) Hint Maintain an auxiliary set of vertices that have already been discovered.

C-14.63) Hint The greedy algorithm presented in this problem is not guaranteed to find the shortest path. Why not?

C-14.63) Solution The greedy algorithm presented in this problem is not guaranteed to find the shortest path between vertices in graph. This can be seen by counterexample. Consider the weighted graph with vertices A, B, C and D, and the following edges and corresponding weights:

- $(A,B,1)$
- $(A,C,2)$
- $(B,D,3)$
- $(C,D,1)$

Suppose we wish to find the shortest path from $start = A$ to $goal = D$ and we make use of the proposed greedy strategy. Starting at A and with $path$ initialized to A, the algorithm will next place B in $path$ (because (A,B) is the minimum-cost edge incident to A) and set $start$ equal to B. The lightest edge incident to $start = B$ which has not yet been traversed is (B,D). As a result, D will be appended to $path$ and $start$ will be assigned D. At this point the algorithm will terminate (since $start = goal = D$). In the end we have that $path = A,B,D$, which clearly is not the shortest path from A to D. (Note: in fact the algorithm is not guaranteed to find any path between two given nodes since it makes no provisions for backing up. In an appropriate situation, it will fail to halt.)

C-14.64) Hint Think about the important fact about minimum spanning trees.

C-14.65) Hint How does the number of connected components change from one iteration of Baruvka's algorithm to the next?

C-14.66) Hint Use the tree-based union/find data structure from Chapter 10 along with one of the MST algorithms given in Chapter 12 and a good sorting algorithm.

C-14.66) Solution Use bucket-sort to sort the edges in $O(m)$ time and then use the tree-based union/find data structure from Chapter 10 along with Kruskal's algorithm to find the MST in $O(m\log^* n)$ additional time (using the sorted order to determine the order in which to process the edges).

C-14.67) Hint This is not a shortest-path problem.

C-14.67) Solution We can model this problem using a graph. We associate a vertex of the graph with each switching center and an edge of the graph with each line between two switching centers. We assign the weight of each edge to be its bandwidth. Vertices that represent switching centers that are not connected by a line do not have an edge between them.

We use the same basic idea as in Dijkstra's algorithm. We keep a variable $d[v]$ associated with each vertex v that is the bandwidth on any path from a to this vertex. We initialize the d values of all vertices to 0, except for the value of the source (the vertex corresponding to a) that is initialized to infinity. We also keep a π value associated with each vertex (that contains the predecessor vertex).

The basic subroutine will be very similar to the subroutine *Relax* in Dijkstra. Assume that we have an edge (u,v). If $\min\{d[u],w(u,v)\} > d[v]$ then we should update $d[v]$ to $\min\{d[u],w(u,v)\}$ (because the path from a to u and then to v has bandwidth $\min\{d[u],w(u,v)\}$, which is more than the one we have currently).

Algorithm Max-Bandwidth(G,a,b)

a. FOR (all vertices v in V)
    $d[v] = 0$

b. $d[a] = \infty$

c. $Known = \emptyset$

d. $Unknown = V$

e. WHILE ($Unknown \neq \emptyset$)
    $u = ExtractMax(Unknown)$
    $Known = Known \cup \{u\}$
    FOR (all vertices v in $Adj[u]$)
        IF ($min(d[u],w(u,v)) > d[v]$)
            $d[v] = min(d[u],w(u,v))$

f. return (d[b])

Here, the method $ExtractMax(Unknown)$ finds the node in Unknown with the maximum value of d.

Complexity (this has not been asked for):
If we maintain $Unknown$ as a heap, then there are $n = |V|$ ExtractMax operations on it and up to $m = |E|$ changes to the values of elements in the heap. We can implement a data structure which takes $O(\log n)$ time for each of these operations. Hence, the total running time is $O((n+m)\log n)$.

C-14.68) Hint Use a greedy algorithm.

C-14.68) Solution Consider the weighted graph $G = (V,E)$, where V is the set of stations and E is the set of channels between the stations. Define the weight $w(e)$ of an edge e in E as the bandwidth of the corresponding channel.

Given below are two ways of solving the problem:

a. At every step of the greedy algorithm for constructing the minimum spanning tree, instead of picking the edge having the least weight, pick the edge having the greatest weight.
b. Negate all the edge-weights. Run the usual minimum spanning tree algorithm on this graph. The algorithm gives us the desired solution.

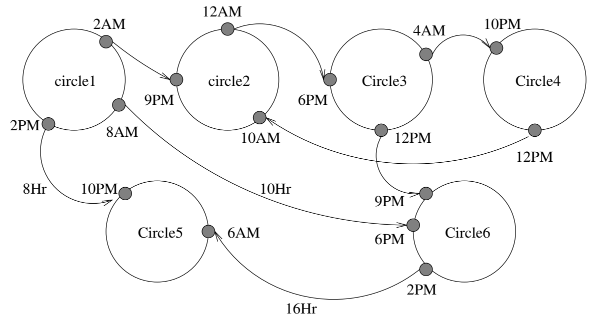
C-14.69) Hint Model this problem as an optimal path problem that goes between two vertices in a directed graph without cycles.

C-14.69) Solution Simply run the shortest-path algorithm with $path(a) < path(b)$ if there is more gold on $path(a)$ than $path(b)$.

C-14.70) Hint Define an augmented graph from the input graph and perform the appropriate computation on the augmented graph.

C-14.70) Solution There are two possible solutions:

a.
- For each airport a_i in $\mathcal{A}$, draw a circle $circle_i$ to represent the time (24 hours).
- For each flight f in $\mathcal{F}$, find the origin airport a_o, the destination airport a_d, the departure time t_d, and the arrival time t_a. Draw a vertex v_1 on $circle_o$ marked the time t_d, also draw a vertex v_2 on $circle_d$ marked the time (t_a+c(a_d)). Draw an directed edge from v_1 to v_2 with weight t_a+c(a_d)-t_d (of course we must compute the correct weight (flight time) in case like the departure time is 10:00PM and the arrival time is 1:00AM next day). The direction of edges on a circle should be clockwise.
- Now we have a new graph like the figure below:



Note that we have included the minimum connecting time in the total flight time. We can start the modified new flights without worrying about the connecting times.

Then to solve this problem is to find the shortest path from the first vertex on $circle_a$ (origin airport a) representing time t or after t to one vertex on $circle_b$ (destination airport b) in the graph. The flight sequences can be obtained from the shortest path from origin to destination.

b. The following algorithm finds the minimum travel time path from airport $a \in \mathcal{A}$ to airport $b \in \mathcal{A}$. Recall, we must depart from a at or after time t. Note, "$\oplus$" and "$\preceq$" are operations which we must implement. We define these operations as follows: If $x \oplus y = z$, then z is the point in time (with date taken into consideration) which follows x by y time units. Also, if $a \preceq b$, then a is a point in time which precedes or equals b. These operations can be implemented in constant time.

initialize fringe to empty
initialize incoming_flight(x) to nil for each x in $\mathcal{A}$
earliest_arrival_time(a)=t
earliest_arrival_time(x)=∞ for x in $\mathcal{A}$ where $x \neq a$
repeat
    remove from fringe airport x such that $earliest\_arrival\_time(x)$ is soonest
    if $x \neq b$ then
        if $x = a$ then
            time_can_depart=t
        else
            time_can_depart=earliest_arrival_time(x) $\oplus$ c(x)
        for each flight f in $\mathcal{F}$ such that $a_1(f) = x$
            if $time\_can\_depart \preceq t_1(f)$ and $t_2(f) \preceq earliest\_arrival\_time(a_2(f))$ then
                earliest_arrival_time($a_2(f)$)=$t_2(f)$

incoming_flight($a_2(f)$)=f
add $a_2(f)$ to the fringe if it is not yet there
until $x = b$

initialize $city$ to b
initialize $flight\_sequence$ to nil
while (city$\neq a$) do
append $incoming\_flight(city)$ to $flight\_sequence$
assign to $city$ the departure city of $incoming\_flight(city)$

reverse $flight\_sequence$ and output

The algorithm essentially performs Dijkstra's shortest-path algorithm (the cities are the vertices and the flights are the edges). The only small alterations are that we restrict edge traversals (an edge can only be traversed at a certain time), we stop at a certain goal, and we output the path (a sequence of flights) to this goal. The only change of possible consequence to the time complexity is the flight sequence computation (the last portion of the algorithm). A minimum-time path will certainly never contain more that M flights (since there are only M total flights). Thus, we may discover the path (tracing from b back to a) in $O(M)$ time. Reversal of this path will require $O(M)$ time as well. Thus, since the time complexity of Dijkstra's algorithm is $O(M \log N)$, the time complexity of our algorithm is $O(M \log N + M + M) = O(M \log N)$. Note: Prior to execution of this algorithm, we may (if not given to us) divide $\mathcal{F}$ into subsets $F_1, F_2, \ldots, F_N$, where F_i contains the flights which depart from airport i. Then when we say "for each flight f in $\mathcal{F}$ such that $a_1(f) = x$", we will not have to waste time looking through flights which do not depart from x (we will actually compute "for each flight f in F_x"). This division of $\mathcal{F}$ will require $O(M)$ time (since there are M total flights), and thus will not "slow down" the computation of the minimum time path (which requires $O(M \log N)$ time).

C-14.71) Hint If $M^2(i, j) = 1$, then there is a path of length $<= 2$ (a path traversing at most 2 edges) from vertex i to vertex j in the graph G. Alternatively, if $M^2(i, j) = 0$, then there is no such path. Now generalize from here...

C-14.71) Solution

a. If $M^2(i, j) = 1$, then there is a path of length $<= 2$ (a path traversing at most 2 edges) from vertex i to vertex j in the graph G. Alternatively, if $M^2(i,j) = 0$, then there is no such path.

b. Similarly, if $M^4(i,j) = 1$, then there exists a path of length $<= 4$ from v_i to v_j, otherwise no such path exists. The situation with M^5 is analogous to that of M^4 and M^2. In general, M^p gives us all the vertex pairs of G which are connected by paths of length $<= p$.

C-14.72) Hint The running time is more than linear.

C-14.72) Solution Karen's algorithm sets parent pointers to the parent pointers of parents. Since the paths from a position in a tree-based partition structure go up until reaching the root, which points to itself, eventually every position will point to the root. Moreover, each time Karen makes a pass over S, she doubles the number of position each position has "linked out" of its path to the root. Thus, the running time of her algorithm is $O(h \log h)$.

Projects

P-14.73) Hint Test the basic structure by having a method that visualizes the adjacency matrix.

P-14.74) Hint Test the basic structure by having a method that visualizes the edge list.

P-14.75) Hint Test the basic structure by having a method that visualizes the adjacency list.

P-14.76) Hint Implement the static undirected version first, then add the extra methods.

P-14.77) Hint Have some way of printing out each phase of the algorithm for debugging purposes.

P-14.78) Hint Look at Dijkstra's shortest path algorithm as a model.

P-14.79) Hint Use uniform and Gaussian distributions for the edge weights.

P-14.80) Hint Use the Swing GUI components to implement the visualization.

P-14.81) Hint Do a Breadth-First Search from each node in the network.