**Chapter 13**

**R-13.7** Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

a. The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.

b. The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

c. You need to answer the query are Adjacent as fast as possible, no matter how much space you use.

*Answer:*a. The adjacency list is used to save the space as the requirement is to use as little space as possible. For vertex, the space used by the adjacency list is proportional to the degree of the vertex hence O(deg(v)). As per theorem 13.6 the space requirement for the adjacency list structure for a graph with n vertices and m edges = O (n + m) where the number of vertices is 10,000 and number of edges is 20,000 which will in turn utilize least space as opposed to the adjacency matrix which would have taken O (n2) space.

b. In this case adjacent list and matrix both can be used as both have the space requirement and the adjacency list is used for insertion and removal of vertex and on the other hand, the adjacency matrix is used to determine which of the two vertexes are adjacent which takes O (1) time. But adjacency matrix is preferred because the graph has close to a quadratic number of edges.

c. The adjacency matrix is preferred as it can be used to determine which of the two vertices are adjacent in O (1) time. This performance can be gained by accessing both vertices to determine their respective indices I and j and then testing whether the cell A[I,j] is null or not.

**C-13.11** The directed version of the BFS algorithm classifies nontree edges as being either back edges or cross edges, but it does not distinguish between these two types. Given a BFS spanning tree, *T*, for a directed graph, *\_G*, and a set of nontree edges, *E\_*, describe an algorithm that can correctly label each edge in *E\_* as being either a back edge or cross edge. Your algorithm should run in *O* (*n* + *m*) time, where *n* is the number of vertices and *m* is the number of edges.

*Hint:* Consider first constructing a Euler tour traversal of the tree *T*.:

*Answer:*

Euler tour traverses each edge of the graph exactly once. Any graph that has Euler tour are called Eulerian. The algorithm of Euler tour of the tree T is as follow:

Euler tour(T, V):

Perform the action for visiting node v on the left

If the v is an internal node then recursively tour the left subtree of v by calling EurlerTOur (T, T.leftChild(v))

Perform the action for visiting node v from below

If the v is an internal node then recursively tour the right subtree of v by calling, EulerTour(T, T. rightChild(v) )

Perform the action for visiting node v on the right.

The running time of the Euler tour traversal can be analyzed easily, assuming visiting a node takes O (1) time. A constant amount of time at each node of the tree is sped during the traversal, so the total running time is O (n).

We modify the BFS algorithm to correctly label each edge as back edge or cross edge (Process of finding back edge and cross edge already given in the algorithm) as follows:

Algorithm BFS (T, node):

Input: A graph G and node

Output: A labeling of the edges in the connected component of s as discovery edges and cross edges

node → root

put root to Li

while Li is not empty do

create an empty list, Li+1

for each vertex, v, in Li do

if(T.hasLeft(node)) then

root → T. left ()

if edge e is unexplored then

if vertex w is unexplored then

e → discovery edge

w → explored and insert w into Li+1

else

e → cross edge

else

continue // if the edge is already explored

if(T.hasRight(node)) then

root → T. right ()

if edge e is unexplored then

if vertex w is unexplored then

e → discovery edge

w → explored and insert w into Li+1

else

e → cross edge

else

continue // if the edge is already explored

i ← i + 1

As Euler traversal method is used each vertex in the tree is visited only once, and so the algorithm will take same time taken by BFS algorithm O (n+m) where n is the number of vertices and m is the number of edges.

**A-13.6** A company named RT&T has a network of *n* stations connected by *m*high-speed communication links. Each customer’s phone is connected to one station in his or her area. The engineers of RT&T have developed a prototype video-phone system that allows two customers to see each other during a phone call. In order to have acceptable image quality, however, the number of links used to transmit video signals between the two parties cannot exceed 4. Suppose that RT&T’s network is represented by a graph. Design an efficient algorithm that computes, for each station, the set of stations it can reach using no more than 4 links.

*Answer:*

DFS search is applied for each vertex to find the station which can be reached by at most 4 edges from the vertex. Here it is given that there is n number of vertices and m edges (Stations and links).

Hence

Algorithm ModifiedDFS (G, V, depth, s)

Input The graph, vertex and its depth and the sequence as an input.

Output Sequence of vertices reachable from v in at most depth

Count = 0

Mark v as visited

If (depth = 0) then

Return s

For each edge in G.incident Edges() do

Tov ← G.opposite (edge, v)

If (! IsVisited(Tov) then

S.insetLast(Tov)

S←ModifiedDepthForSearch(G, Tov, depth -1 , s)

Return s

SO, to compute set of stations where each station can reach other with no more than 4 links we can implement the algorithm that is below:

ComputeSets(G)

Where the graph is the input and the output is a dictionary containing keys on the vertex with values being sequences of nodes 4- reachable from that vertex.

For each vertex in G.vertices()

D.insert (Vertex, Modified\_DFS(G, Vertex, 4, New\_Sequence))

The graph where each vertex is at most 4 edges far from any other, is given and here DFS traversals are applied.

**Chapter 14**

**C-14.7** Suppose you are given a connected weighted undirected graph, *G*, with *n* vertices and *m* edges, such that the weight of each edge in *G* is an integer in the interval [1*, c*], for a fixed constant *c >* 0. Show how to solve the single-source shortest paths problem, for any given vertex *v*, in *G*, in time *O* (*n* + *m*).

*Hint:* Think about how to exploit the fact that the distance from *v* to any other vertex in *G* can be at most *O*(*cn*) = *O*(*n*).

*Answer:*

To find the single source shortest path which has different time complexity various implementations are used. For example, to get complexity O (n +m) we use Dijkstra’s algorithm. A priority queue is implemented, and a lookup table of size O (cn) = O (n) is used where D[i] is representing a set where all the vertices with D[v] label equal to i. The distances from the beginning are drastically increasing iterations is continues through the Dijkstra’s algorithm, the non-empty cell D[i] is tracked in D with the smallest index I in amortized O (1) time. This process allowed to perform removeMin operations in amortized constant time by removing from the non-empty D [i] cell with smallest I and update the key value for any vertex in O (1) time by changing the location from one cell in D to another.

Hence, total running time for Dijkstra’s algorithm becomes O (n + m).

**A-14.2** Suppose that CONTROL, a secret U.S. government counterintelligence agency based in Washington, D.C., has build a communication network that links *n* stations spread across the world using *m* communication channels between pairs of stations. Suppose further that the evil spy agency, KAOS, is able to eavesdrop on some number, *k*, of these channels and that CONTROL knows the *k* channels that have been compromised. Now, CONTROL has a message, *M*, that it wants to send from its headquarters station, *s*, to one of its field stations, *t*. The problem is that the message is super secret and should traverse a path that minimizes the number of compromised edges that occur along this path. Explain how to model this problem as a shortest-path problem, and describe and analyze an efficient algorithm to solve it.

*Answer:*

The weight of 0 to each uncompromised edge and weight of 1 to each compromised edge can be assigned. Hence the shortest path from s to t will be minimizing the number of compromised edges along this path and Dijkstra’s algorithm is applied.

DijkstraShortestPaths(G,V):

A simple undirected weighted graph G with nonnegative edge weights and a distinguished vertex v of G is the input and output is a label, D[u] for each vertex u of G such that D [u] is the distance from v to u in G

D[v]←0

For each vertex, u ≠ v of g fo

D[u] ←+∞

Let priority queue q contain all the vertices of G using the D labels as keys.

While Q is not empty do

u←Q.removeMin()

for each vertex a adjacent to u such that z is in q do

if D [u] + w(u,a) < D[a] then

D[a] ← D [u] +w (u,a)

Change key for vertex z in q to D[a] return the label D {u} of each vertex u

The above algorithm would take O ((n + m) log n) time to find the shortest path using the Dijkstra’s algorithm.

**A-14.5** As your reward for saving the Kingdom of Bigfunnia from the evil monster “Exponential Asymptotic,” the king has given you the opportunity to earn a big reward. Behind the castle, there is a maze, and along each corridor of the maze, there is a bag of gold coins. The amount of gold in each bag varies. You will be given the opportunity to walk through the maze, picking up bags of gold. You may enter only through the door marked “ENTER” and exit through the door marked

“EXIT.” (These are distinct doors.) While in the maze you may not retrace your steps. Each corridor of the maze has an arrow painted on the wall. You may only go down the corridor in the direction of the arrow. There is no way to traverse a “loop” in the maze. You will receive a map of the maze, including the amount of gold in and the direction of each corridor. Describe and analyze an efficient

algorithm to help you pick up the most gold in this maze while traversing a path from the start to the finish.

*Answer:*

Here directed Acyclic Graph is used based on the map. The weight of each edge is added based on the amount of gold. Dijkstra’s shortest path algorithm can be applied, but instead of searching for the shortest oath we find the most weighted path in this case and we find which is the one with the most gold.

DAGShortestPath(G, s): would be the algorithm where a weighted directed acyclic graph DAG G with n vertices, m edges and a distinguished vertex s in G with output a label D[u], for each vertex U of G, such that D[u] is the distance from v to u in G.

Compute a sequence in this order (v1, v2, v3, v4, ……., vn) for G

D[s] ← 0

For each vertex u ≠s of G do

D[u] ← +∞

For I ←1 to n-1 do

For each edge (vi, u) outgoing from vi do

If D[vi] +w ((vi, u) < D[u] then

D[u] ← D [vi] + w((vi,u))

Output the distance labels D as the distances from s.

This algorithm is based to find the path having shortest distance between 2 vertices.

Here, the weight is the distance between 2 vertices and finding the shortest distance.

We have height as the pack of gold which we need to maximize. So. Only change would be in the edge relaxation part where the summation of initial Weight and new weight from vi to u should be greater than the weight of u, w (v, u) = pack of gold.

Here for each edge vi, u) outgoing from vi do

If D[vi] +w (vi, u) < D[u] then

D [u[ = D[vi] + w( vi, u)