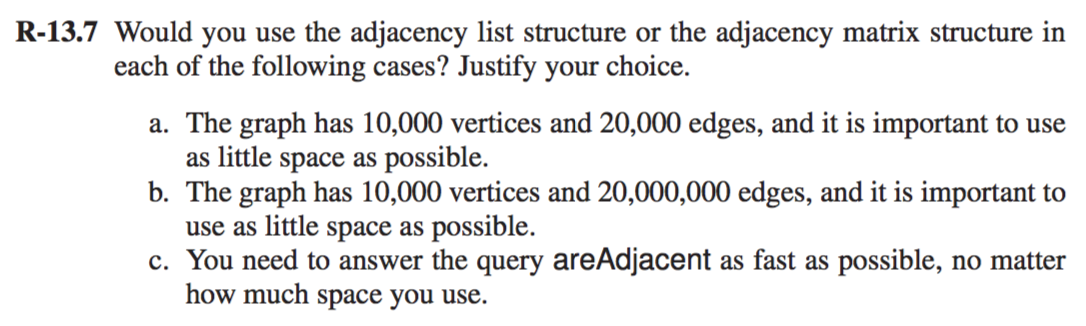
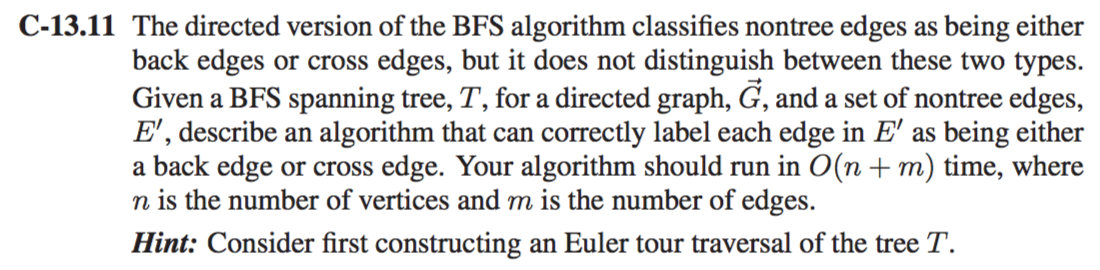
CS 600 Homework 7 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 10/23/2017



1] Consider a graph with 10,000 vertices and 20,000 edges, adjacency list will be a preferred option in order to utilize as little space as possible. This is because, for a vertex v, the space used by adjacency list for v is proportional to the degree of v. that is O(deg(v)). Thus, as defined by the theorem 13.6, the space requirement for the adjacency list structure for a graph with n vertices and m edges, is O(n + m), in this case n = 10,000 and m = 20,000, which will utilize less space as opposed to adjacency matrix which uses O(n2) space.

2] Here the graph is composed of 10,000 vertices and 20,000,000 edges. Hence, in this case adjacency matrix will be a more preferred option as the graph has close to a quadratic number of edges. Also, for the areAdjacent() operation adjacency matrix is suited better as it can determine the two adjacent vertices in O(1) time.

3] To perform areAdjacent operation as fast as possible, using adjacency matrix will be the most preferred option. This is because, using adjacency matrix, we can determine whether two vertices v and w are adjacent in O(1) time. We can achieve this performance by accessing the vertices v and w to determine their respective indices i and j and then testing whether the cell A[i,j] is null or not.



An **Euler tour** (or Eulerian tour) in a graph is a tour that traverses each edge of the graph exactly once. Graphs that have an Euler tour are called **Eulerian**. The algorithm for Euler tour traversal of the tree T is as follows:

**Algorithm** eulerTour(*T, v*):

perform the action for visiting node *v* on the left

if *v* is an internal node then

recursively tour the left subtree of *v* by calling eulerTour(*T, T.*leftChild(*v*))

perform the action for visiting node *v* from below

if *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 is easy to analyze, assuming visiting

a node takes *O*(1) time. Namely, in each traversal, we spend a constant amount

of time at each node of the tree during the traversal, so the overall running time

is *O*(*n*) for an *n* node tree.

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 algo) 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

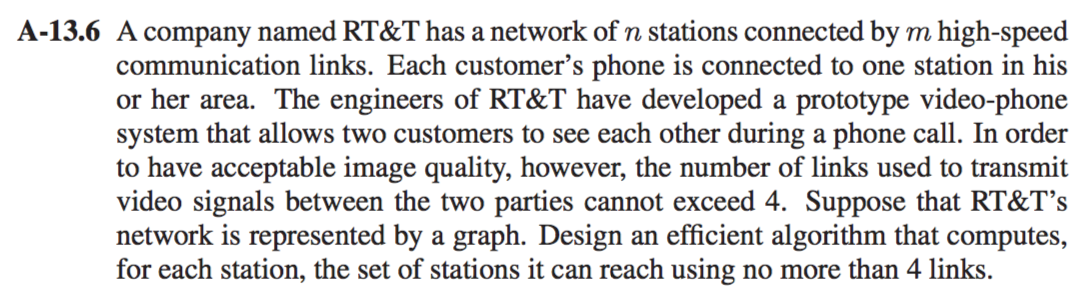
else

continue // if the edge is already explored

i ← i + 1

Since we are using Euler Traversal Method, each vertices in the tree is visited only once.

Therefore this algorithm will take same time which is taken by the BFS algorithm **O(n + m) where n is the number of vertices and m is the number of edges.**



We can perform a modified DFS search, for each vertex (station), to find the stations which can be reached by at most 4 edges (links) from the vertex. It is given that there are n vertices(stations) and m edges (links). Thus, we modify the DFS algorithm as follows:

Algorithm ModDFS (G,V,Depth,S)

**Input:** We take the graph, the vertex, its depth and the sequence as an input

**Output:** We get a sequence of vertices reachable from v in at most depth hops

cnt= 0

mark v as visited

if (depth = 0) then

return S

for each edge in G.incidentEdges() do

tov G.opposite(edge,v)

if (! isVisited(tov)) then

S.insertLast(tov)

S ModDepthForSearch(G,tov,depth − 1,S)

return S

In order to compute a set of stations where each station can reach other with no more than 4 links, we can implement the following algorithm :

**Algorithm ComputeSets(G)**

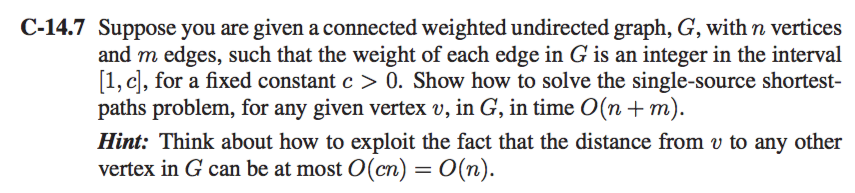
**Input:** A graph

**Output:** A dictionary containing keys on the vertex with values being sequences of nodes 4-reachable from that vertex D in a new empty dictionary.

for each vrtx in G.vertices()

D.insert (vrtx, Modified\_DFS(G, vrtx, 4, New\_Sqnce))

Given a graph where each vertex is at most 4 edges far from any other, we will perform n complete DFS traversals. Thus, the worst-case running time is O(n(n + m)).



1] We have different types of implementations for finding the single source shortest path which can has different time Complexity. To get the complexity O(n + m) suppose we use Dijkstra’s Algorithm.

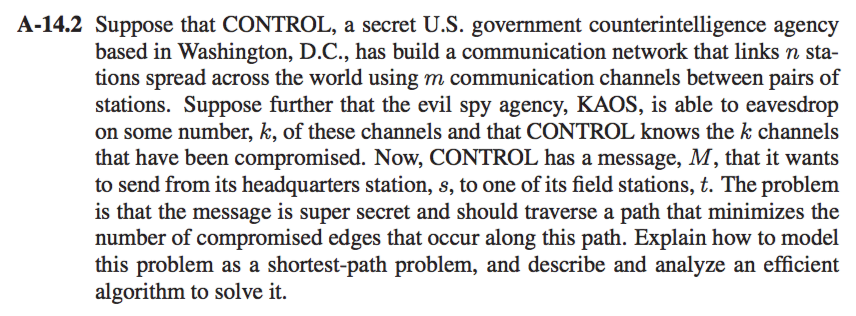
2] We solve this problem by implementing a priority queue P.

We also use a lookup table, D, of size O(cn) = O(n) (Suggested in question), where D[i] is representing a set where all the vertices with D[v] label equal to i.

3] Because the distances from the beginning are drastically increasing as we are iterating through Dijkstra’s algorithm, we keep track of the nonempty cell, D[i], in D with smallest index, i, in amortized O(1) time.

4] This allows us to perform removeMin operations in amortized constant time by removing from the non-empty D[i] cell with smallest i and also to update the key value for any vertex in O(1) time by moving the vertex from one cell in D to another.

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



We can give a weight of 0 to each uncompromised edge and a weight of 1 to each compromised edge. Hence, shortest path from s to t will be minimizing the number of compromised edges along this path and we apply Dijsktra’s Algo.

**Algorithm** DijkstraShortestPaths(*G, v*):

***Input****:* A simple undirected weighted graph *G* with nonnegative edge weights,

and a distinguished vertex *v* of *G*

***Output****:* 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* do

*D*[*u*] *←* +*∞*

Let a priority queue, *Q*, contain all the vertices of *G* using the *D* labels as keys.

while *Q* is not empty do

// pull a new vertex *u* into the cloud

*u ← Q.*removeMin()

for each vertex *z* adjacent to *u* such that *z* is in *Q* do

// perform the *relaxation* procedure on edge (*u, z*)

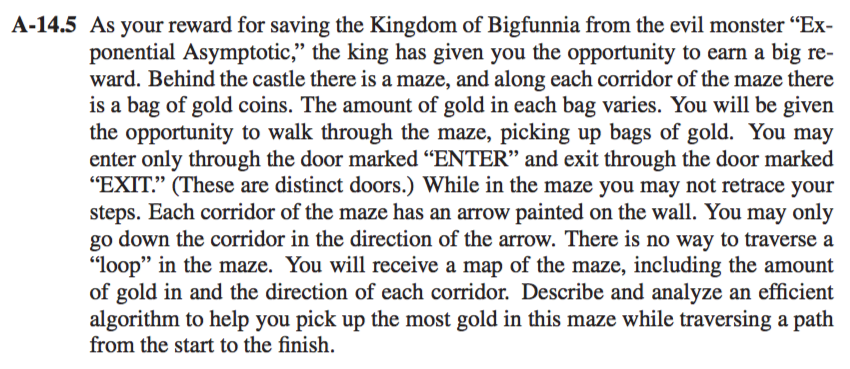
if *D*[*u*] + *w*((*u, z*)) *< D*[*z*] then

*D*[*z*] *← D*[*u*] + *w*((*u, z*))

Change the key for vertex *z* in *Q* to *D*[*z*]

return the label *D*[*u*] of each vertex *u*

The above algorithm takes O((n + m) log n) time to find the shortest path using Dijsktra’s Algorithm.



This is an example of Directed Acyclic Graph.

**Algorithm** DAGShortestPaths(*G, s*):

***Input****:* A weighted directed acyclic graph (DAG) *G* with *n* vertices and *m*

edges, and a distinguished vertex *s* in *G*

***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 topological ordering (*v*1*, v*2*, . . . , vn*) for *G*

*D*[*s*] *←* 0

**for** each vertex *u ≠ s* of *G* **do**

*D*[*u*] *←* +*∞*

**for** *i ←* 1 to *n −* 1 **do**

// Relax each outgoing edge from *vi*

**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*.

The above algorithm is based to find the path having shortest distance between 2 vertices. Here, weight is the distance between 2 vertices and finding the shortest distance.

In our Question, We have weight as the pack of gold which we need to maximize. So only change would be in the Edge Relaxation part: i.e. summation of Initial Weight and new weight from 𝑣𝑖 𝑡𝑜 𝑢 should be greater than weight of 𝑢. 𝑤(𝑣, 𝑢) = pack of gold.

**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))**

With this we will always choose the path with highest gold. This algorithm will take 𝑶(𝒏 + 𝒎) time.