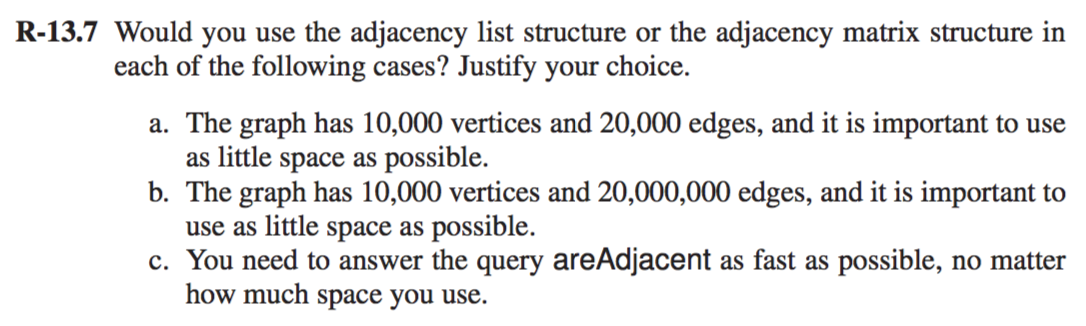
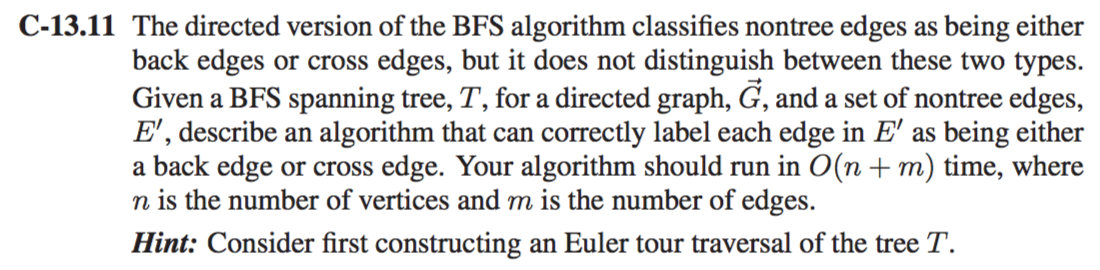
Homework 6

Atiq Patel CWID: 10432883



**Solution:**

1. An Adjacency list will be preferred option graph with 10,000 vertices with 20,000 edges when it comes to saving space. This is for the reason that for every vertex x, the space that is used by adjacency list for v is directly proportional to the degree of v O(deg(v)) which is mentioned in Theorem 13.6. O (n+m)
2. In this case, there are 10,000 vertices with 20,000,000 edges. There are a lot more edges inside the case so in case adjacency matrix would be preferred as for every vertex v there are almost as much as quadratic amount of edges.
3. The areAdjacent operation can be done with O (1) time if we make the use of an Adjacency matrix instead of a list. It would however take more space than the list but it will give you the fastest time **O (1)** time.



**Solution:**

An Euler Tour in a graph is a tour that traverses each and every node present in the graph. These types of Tour for a graph are called as Eulerian graph. The following the pseudocode for the traversal.

**Algorithm EulerFind(T,n):**

**Input:** A graph T

**Output:** The T with all edges E being either a back edge or a front one.

Start by performing the action for visiting node n on the left side

**If (**n== internal node) **then do**

EulerFind(T,T.leftchild(n)) //Tour recursively the left subtree of v

//Perform the action for visiting node n from below

**If** (n==internal node) **then do**

EulerFind(T,T.rightchild(n))

//Perform the action for visiting node n on the right side

The run time for this algorithm is O (1) time. The Euler Tour traversal would visiting nodes which takes O (1) time and the time taken is constant. Since there are n nodes present in the tree T. The total run time would be **O (n)** times.

**Algorithm BST(T,node):**

**Input: :** A graph T

**Output:** The graph T with all edges E being either a back edge or a front one.

Root🡨node

Ai🡸root

EL🡸0

**While**( Ai!= empty) **do**

El🡸EL+1

**Foreach** vertex v in EL **do**

**If** (T.hasnext(Leftnode)) **do**

T.leftnode🡨root

**if** (Edge(e)!=visited) **do**

**if** (Vertex(w)!=visited) **do**

//Mark Edge(e) as visited

Edge(e)🡸discovery edge

//Mark w as visited and insert into empty list EL

EL🡸explored(w) //put it into Ai

**Else do**

Edge(e)🡸 Cross Edge

**Else do**

**//**Continue

**If** (T.hasnext(Rightnode)) **then**

T.rightnode🡨root

**if** (Edge(e)!=visited) **do**

**if** (Vertex(w)!=visited) **do**

//Mark Edge(e) as visited

Edge(e)🡸discovery edge

//Mark w as visited and insert into empty list EL

EL🡸explored(w) //put it into Ai+1

**Else do**

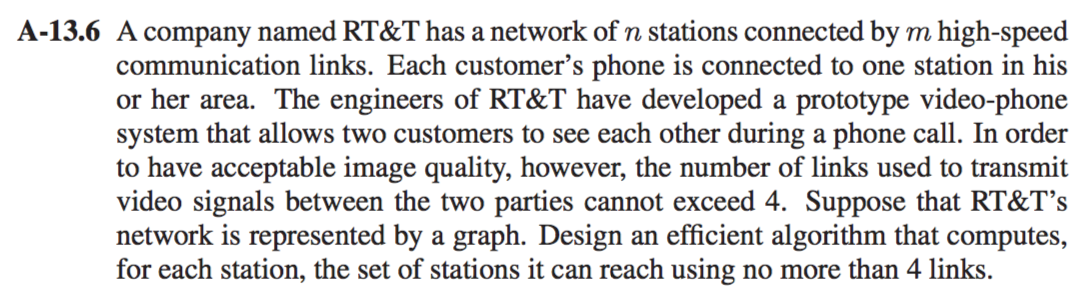
Edge(e)🡸 Cross Edge

**Else do**

**//**Continue

i🡸i+1

Each and every vertices of the graph tree T are visited only once so suppose there n number of vertices and m number of edges then the run time for this algorithm would be **O(n+m).**



**Solution:**

This can be done using Depth First Search with slight modifications wherein the each station would only be able to reach other stations with not more than 4 stations. Here the stations are considered as links

**Algorithm MDFS(G,V,D,S):**

**Input:** A graph G with a set of vertices V, Its Depth D and array A

**Output:** A hashmap Final which is in key value pair which gives out each the total links of each vertex which is less than 4.

Label *v* as explored

for each edge, *e*, that is incident to *v* in *G* do

if *e* is unexplored then

Let *w* be the end vertex of *e* opposite from *v*

A🡨//put opposite edge into A.

if *w* is unexplored then

A🡨//INSERT w at the end.

A🡨MDFS(*G,w,d-1,A*)

Return A

**Main()**

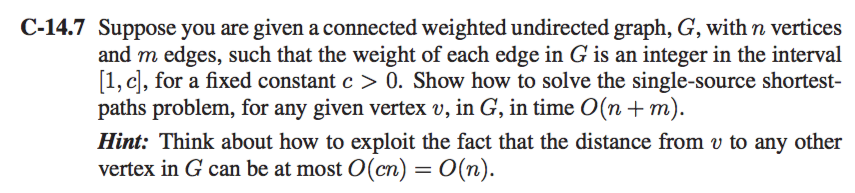
Final[]🡸0

**ForEach** Vertex(v) in G.vertex():

Final[k,v]🡸 [vertex,MDFS(G,V,4,S)] //stores the result which is the information of each vertex and its link not more than 4 links in the hashmap in key value pairs.

Display[Final]

The total operation will be for n nodes and m edges which are 4 in this cast. Thus it will perform n times DFS traversings. The run time for this algorithm would be **O (n(n+m))** which also be said as O(n^2). as m=4 which we know in this case.

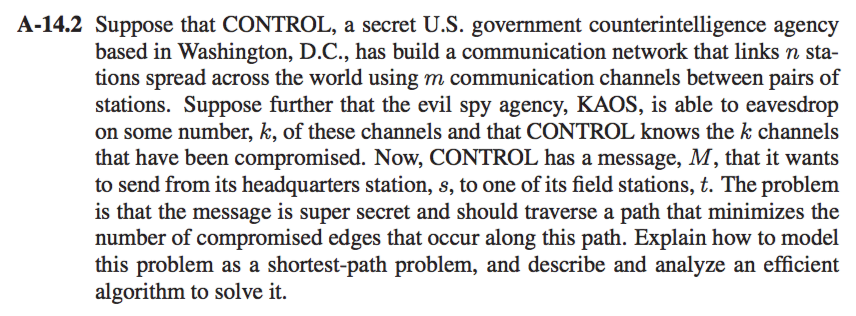


**Solution:**

There are several approaches to find a shortest path to this problem but we would take the Dijkstra’s Algorithm to find the shortest path because it gives us the time complexity of O (n+m).

Lets consider a priority queue PQ. A table T of size cn wherein T(i) shows a set that contains all the vertices v of T(v) equivalent to i. This would consume O (cn) time but as the question suggested we would consider it as O (n) as c is a constant.

After each and every interval we have to keep a record of it. Thus it would be stored in T(i) only if it contains the smallest index i.



**Solution:** Let us assume the weight 0 for uncompromised edges and 1 being the weight for compromised edges. Thus the shortest path from v to u will be minimizing the number of compromised edges we can apply Dijsktra’s All pair shortest path algorithm.

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

**Input:** A simple graph undirected graph G with weights and vertex v of G.

**Output:** An output graph Final[D] for each vertex D of G sucht that Final[D] is the distance from v to D inside G.

Final[v]🡸0

**ForEach**(Vertex(D)!=Vertex(V)) inside G **do**

Final[u]🡸+infinity

Consider a priority queue PQ which has all the vertices of G

**While**(PQ!= empty) **do**

u🡨pq.removeMinimum()

**foreach** vertex(Z) adjacent to v such that there exists Z which is inside PQ **do**

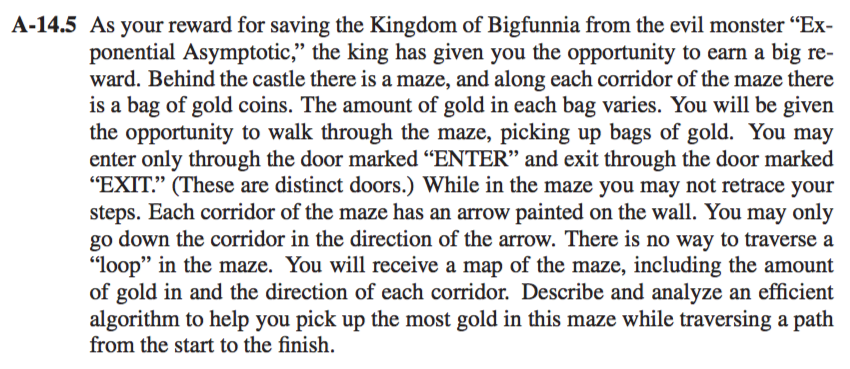
//Edge Relaxation on Edge(u,z)

**If** Final[v] + w((v,z)) < Final[z] **do**

Final[D]🡸 Final[u]+w((v,z))  
 Final[D]🡸 Transfer key(Vertex(z)) in Q

Display Final[D] // for every vertex u

The run time for this algorithm is **O ((n+m) log n)** to find the shortest path with uncompromised edges.



**Solution:**

This is a Directed Acyclic Graph with maximization problem.

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

**Input:** A DAG G with weighted with n vertices and m edges and a unique vertex Ux in G

**Ouput:** An optimized and maximized path for finding the maximum gold. Final[D] where is a is the vertex of D in G such that Final[D] is the distance from v to D.

**Foreach** Vertex(v) to Vertex(d) in G

Topological sort (v1,v2,,v3,….vn)

Final[Ux]🡸0

**Foreach** vertex a != Ux of G **do**

Final[D] 🡸 +inf

**For** i🡸0 to n-1 **do**

**ForEach** Edge(Vi,D) from vi **do**

**If** Final[Vi] + w((Vi,d)) < Final[D] **do**

Final[D]🡸Final[Vi]+ w((Vi,u))

Display Final[D] //Output

The algorithm given is used to find Shortest path between 2 vertices. Here the shortest path is actually the weight of the algorithm.

Here we are maximizing gold So the total the answer would be the summation would be initial weight plus the new weight w(v,d) where weight vi to d should be greater than the weight u.

**ForEach** Edge(Vi,D) from vi **do**

**If** Final[Vi] + w((Vi,d)) < Final[D] **do**

Final[D]🡸Final[Vi]+ w((Vi,u))

The run time for this algorithm would be **O (n+m).**