CS 600 Homework 6 Solutions

R-13.7 Graph Representation

For a graph, G, with n vertices and m edges, the adjacency list representation uses $O(n+m)$ space and the adjacency matrix representation uses $O(n^2)$ space.

a) Graph has 10,000 vertices and 20,000 edges and it is important to use as little space as possible. In this case $n^2 > n+m$ where n is 10,000 and m is 20,000. Therefore, it is better to represent this graph using an adjacency list.

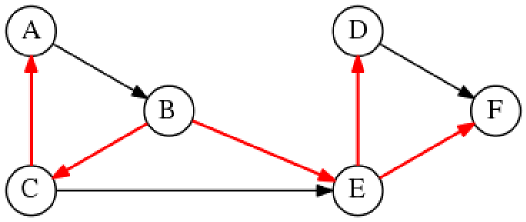
b) Graph has 10,000 vertices and 20,000,000 edges and it is important to use as little space as possible. In this case we have $n^2 > n+m$ where n^2 is 100,000,000 and $n+m$ is 20,010,000. Therefore, it is better to represent this graph using an adjacency list.

c) It is better to use an adjacency matrix to represent the graph since finding whether two vertices, v and w, are adjacent takes $O(1)$ time.

C-13.11 BFS Spanning Tree

We have a BFS Spanning tree, T, for a directed graph, $\vec{G}$, and a set of nontree edges, E'. We need to correctly label each edge in E' as being either a back edge or a cross edge, in $O(n+m)$ time, where n is the number of vertices and m is the number of edges.

Consider the example shown below. The BFS spanning tree, T, rooted at node B is highlighted red.



Note that the edge $A \rightarrow B$ is a back edge and the edge $C \rightarrow E$, is a cross edge. The Euler tour traversal, Eu, of T is given by,

$$Eu = (B, C, A, A, A, C, C, B, E, D, D, D, E, F, F, F, E, B)$$

In this situation, each node is visited 3 times, once from the 'left', once from the 'bottom', and once from the 'right'. Since the occurrences of vertex A are surrounded by vertex C in Eu, we can say that vertex C is the ancestor of vertex A. So, if the edge $A \rightarrow C$ was in E', then it would be a back edge. Similarly, we can conclude by looking at Eu that vertex B must be an ancestor of A, therefore if the edge $A \rightarrow B$ was in

E' it would be a back edge. Likewise, E is an ancestor of F, and so if the edge $F \rightarrow E$ was in E', it would be a back edge.

From this we can arrive at an algorithm to find the set of back edges, E_b from the set E' of all the non-tree edges. We perform an Euler tour traversal, Eu, of the BFS spanning tree, T, and record for each vertex v_i in the graph, the index of it's first and last visits in Eu (we could have a recordIndex method that gets called every time we visit a vertex from the left or from the right). This operation takes $O(n)$ time.

Next we scan through the m number of edges in the graph. Let us suppose that the current edge we are looking at is from v_i to v_j. If the left and right visits of v_j surround the left and right visits of v_i, we can conclude that v_j is an ancestor of v_i. Therefore the edge, $v_i \rightarrow v_j$ is a back edge. If the edge we are looking at is in E' but is not a back edge, then it is a cross edge. Performing this operation takes $O(m)$ time.

In total, we have an algorithm that can distinguish the back edges and cross edges in the graph in $O(n + m)$ time.

A-13.6 RT&T video phone

The RT&T company has a network of n stations connected by m communication links. In order for the prototype video-phone system to have an acceptable image quality, the number of links used to transmit video signals between the two parties in the video phone call cannot exceed 4. We need to design an algorithm that computes, for each station, the set of stations it can reach using no more than 4 links.

We can use a modified version of the Floyd-Warshall algorithm to compute the required set for each station. The transitive closure of a digraph, $\vec{G}$ is $\vec{G}^* = \vec{G}_n$, where $\vec{G}_n$ is computed by the Floyd-Warshall algorithm. Since we are concerned only with stations that are up to 4 links away, computing $\vec{G}_4$ will suffice.

Note that the Floyd-Warshall algorithm presented in the text operates on digraphs. However, every undirected graph can be made into a digraph by replacing each undirected edge in the graph with two directed edges. For the purpose of this exercise we can assume that the graph we are dealing with is viewed as a directed graph (however, it is more likely that the graph is undirected, since the communication link relation is symmetric; this doesn't affect the solution)

Here is the pseudocode for the modified variant of Floyd-Warshall.

Algorithm POSSIBLE-VIDEO-CALLS

```
Input: A graph $\vec{G}$
Output: A graph connecting stations that are 4 links away

$v_1, v_2, \ldots, v_n$ is an arbitrary numbering of vertices in $\vec{G}$
$\vec{G}_0 \leftarrow \vec{G}$
for $k \leftarrow 1$ to $4$ do
    $\vec{G}_k \leftarrow \vec{G}_{k-1}$
    for $i \leftarrow 1$ to $n$ & $i \neq k$ do
        for $j \leftarrow 1$ to $n$ & $j \neq i, k$ do
            if $(v_i, v_k) \wedge (v_k, v_j) \in \vec{G}_{k-1}$ then
                if $(v_i, v_j) \notin \vec{G}_k$ then
                    add $(v_i, v_j)$ and $(v_j, v_i)$ to $\vec{G}_k$
return $\vec{G}_4$
```

Note that we add both (v_i, v_j) and (v_j, v_i) to $\vec{G}_i$, essentially rendering the graph undirected.

The main loop in our case is executed 4 times (indexed by k), and the inner loops run in a total of $O(n^2)$ time assuming that the methods to check whether two stations are adjacent and to add an edge to the graph run in $O(1)$. The total running time of this method is $O(n^2)$. This requires that the graph be implemented using an adjacency matrix.

C-14.7 Single-source shortest path - known weights

We are given a connected weighted undirected graph, G, with n vertices and m edges. The weight of each edge in G is an integer in the interval $[1, c]$ for some fixed constant $c > 0$. We need to show a way of solving the single-source shortest-paths problem, for any vertex v, in G, in $O(n + m)$ time.

Dijkstra's algorithm runs in $O((n+m) \log n)$ time. The $\log n$ factor is due to the heap operations, removeMin and updateKey. If we are able to perform operations that have the same effect in $O(1)$ time, then we have an algorithm that takes time $O(n + m)$ to solve the single-source shortest-paths problem.

We note that the distance from a vertex v to any other vertex in G can be at most $O(cn) = O(n)$. The upper bound for the shortest distance between v and any other node in the graph is cn. We initialize an array, A, of size $cn + 1$. $A[i]$ contains pointers to all nodes in the graph that are of distance i from the starting node v. We perform Dijkstra's algorithm using A instead of a heap with the following changes.

To perform updateKey(u, d), where u is the vertex, and d is the new distance value, we simply remove the u from $A[j]$ and place it in $A[d]$ (where $A[j]$ is the original location of u). Note that this operation takes $O(1)$ time (we have to manipulate a few pointers).

To perform removeMin, we iterate through A until we find an index that is not empty and remove the first vertex at that index. We store the value of this index so that we can later start looking through the array from the index itself. This implies that we traverse the array A only once. Note that in Dijkstra's algorithm, once a vertex is popped using removeMin, the distance value of that vertex doesn't change anymore. Therefore, the method of traversing this array only once is consistent.

This modified version of Dijkstra's algorithm achieves the required time complexity of $O(n + m)$. The important point here is that since there is an upper bound on the maximum distance between the vertices, we can use an array instead of a priority queue.

There is a caveat. Suppose if $c \in \Omega(n^2)$, then this method is worse than the regular implementation of Dijkstra's algorithm. In such a situtaion, one possibility would be to use a hash table (the input distribution of distances is known, so we can assume a hash function that is sufficiently expressive) to acheive an amortized bound of $O(n + m)$.

A-14.2 CONTROL agency problem

The CONTROL agency has built a communication network that links n stations spread across m communication channels. A total of k channels are compromised. Further, CONTROL knows the k channels that have been compromised. A message, M, needs to be sent from the headquarter station, s, to a field station, t. The message should traverse a path that minimizes the number of compromised edges.

We note that the only criteria is that the path should have the least number of compromised edges. The actual 'length' of the path does not matter.

We can model this as a shortest-path problem on a weighted graph, by assigning a weight of 1 to the compromised nodes and a weight of 0 to the other nodes. The goal then is to simply find the shortest path from s to t in this weighted graph. Since there are no negative-weights, we can use Dijkstra's algorithm. Note that we must also assume that the graph is simple.

We can achieve a complexity of $O((n+m)\log n)$ or $O(n^2)$ depending on how we choose to implement the priority queue required by the algorithm. If the graph is dense and the number of edges is in the order of $O(n^2)$, then it is better to use an unsorted doubly linked list to implement the queue. This will achieve a complexity of $O(n^2)$ as opposed to $O(n^2 \log n)$. If the graph isn't dense, then we can use a heap to implement the queue and achieve a complexity of $O((n+m)\log n)$.

A-14.5 Kingdom of Bigfunnia problem

We are given an opportunity to walk through a maze and pick up bags, each containing varying amounts of gold, that are placed along the corridors of the maze. While in the maze, we cannot retrace our steps and can only walk in the directions of arrows painted on the walls. We are given a map of the maze.

The graph that the maze forms in directed and acyclic. Further, it is a simple graph since there are no parallel edges or self-loops. We can apply the single source shortest path algorithm for directed acyclic graphs in this scenario (this algorithm is called DAGShortestPaths in the textbook). We first compute a topological ordering and then use it to figure out the shortest path. However, in this particular case, we want to find the path with the most amount of gold, and not the shortest path. Here is the pseudocode for the modified algorithm that can solve this task.

```
Algorithm MostGold
  Input: A graph $\vec{G}$ and a starting vertex $s$
  Output: Amount of gold picked up at end vertex $t$

  Compute a topological ordering $(v_1, v_2, \ldots, v_n)$ for $\vec{G}$
  $D[s] \leftarrow 0$
  for each vertex $u \neq s$ of $\vec{G}$ do
      $D[u] \leftarrow -\infty$
  for $i \leftarrow 1$ to $n-1$ do
      for each edge $(v_i, u)$ outgoing from $v_i$ do
          if $D[v_i] + \text{Gold}((v_i, u)) > D[u]$ then
              $D[u] \leftarrow D[v_i] + \text{Gold}((v_i, u))$
  return $D[t]$
```

In the above algorithm, the method Gold returns the amount of gold at a certain corridor. Note that the locations where the bags of gold are placed are the vertices of our graph, and corridors are the edges. We initially, set $D[u]$ to $-\infty$ for all vertices $u \neq s$. We modify the value $D[u]$ based on whether the edge we are considering in the current iteration yeilds a path from v_i to u, with more gold.

We end up getting the most number of gold we can gain for each node in the graph. This algorithm runs in $O(n+m)$ time where n is the number of bags of gold, and m is the number of corridors in the maze. This assumes that we can perform a topological ordering of the graph in $O(n+m)$ time (can be done using DFS).