CS 3510 Honors Algorithms
Solutions : Homework 3

Problem 1. [**Shortest Paths**]

(a) Let s be the vertex from which the proposed shortest path tree, say T, is constructed. First, we can traverse the tree in linear time and find the distances $d_T(s, v)$ in T from s to any vertex $v \in V$ (**Check** that this can be done in linear time). Then we can verify, for each directed edge $(u, v) \in E$, whether $d_T(s, u) + \text{weight}(u, v) \geq d_T(s, v)$.

If this is not true, this means that T is not the shortest path tree, because there exists a shorter path to v from s, which is not given by T. Indeed, it can also be proved that if T is not the shortest path tree from s, then we can find an edge, where the above inequality does not hold.

We can easily check that this verification requires $O(|E|)$ running time, once we have all the d_T values calculated.

(b) Consider the graph where there are two paths from s to t. One containing two edges, of weight 2 and -1 each and then a direct edge from s to t of weight 2. Clearly the two edge path is shorter.

Assume that, by the suggested scheme, we are forced to add 5 to all edge weights to make it nonnegative. The two edge path has weights 7 and 4, summing to 11 now and the direct edge is of weight 7. If we run Dijkstra's algorithm, it will return the single edge as the shortest path, which was not the case in the original graph.

(c) Since the edges might be negative, we need to do a more exhaustive procedure here. But we can take advantage of the fact that the shortest path has atmost k edges.

As part of the initialization, we start from u, assigning $d(u) = 0$ and $d(w) = \infty, \forall w \neq u$. Run through all the directed edges (x, y), updating the distances as follows, $d(y) = \min(d(y), d(x) + \text{weight}(x, y))$. Note that this takes $O(|E|)$ steps, and returns the shortest path distances, for which the shortest path consists of atmost one edge.

Repeat this k times, we would get the shortest path distances for the vertices, for which the shortest path consists of atmost k edges. The running time now becomes $O(k|E|)$. (Note that this the **Bellman Ford** algorithm restricted to just k iterations.)

Problem 2. [**One Negative Edge**]
Here we can modify Dijkstra's algorithm to get the shortest path. Suppose we want to find the shortest path from s to t. Let $e = (y, z)$ be the negative weighted edge.

We can start the Dijkstra's algorithm from s and proceed as usual till we reach y, say at a distance $d(s, y)$. Assuming that we have no negative cycles (**Think** on why this is a reasonable assumption to make), run a Dijkstra's algorithm starting from y and find the distance $d(y, t)$. (**Exercise :** Show that when all the negative edges are outgoing edges from y, Dijkstra's will still work.)

Another claim that can be made is that if the shortest path from s to t contains the negative edge e, then the path is given by combining the shortest path from s to y and that from y to t (**Exercise :** Prove this claim). So if the shortest path from s to t contains e, then $d(s, t) = d(s, y) + d(y, t)$. Observe that we are doing just two Dijkstra's here, so the running time complexity is the same as that of the Dijkstra's algorithm.

If the shortest path does not contain the negative edge, then we can run Dijkstra's on the graph with the negative edge removed and this will still return the same shortest path from s to t. (**Why?**) So in either case that the shortest path contains the negative edge or no, we can find out the $d(s, t)$ in time which is the same order as that of Dijkstra's algorithm. We just do each of the above two procedures, and take the smaller of the two values.

Problem 3. [**Bounded Edge Weights**]
We can use the following modified version of BFS. Instead of queueing the vertices, we queue it in such a way that we can take care of the weight of the edges as well. We queue each vertex with a latency equivalent to the weight of the last edge. Whenever we pop a vertex with the latency greater than 1, we decrement latency by 1, and put the vertex back at the end of the queue. Else we do the same operations on it similar to the usual BFS.

```
ModifiedBFS(G,s)
dist(s):=0;
dist(v):=inf for all other v;
```

```
Q=[(s,1)]
while Q is not empty {
     (v,lat) = eject(Q);
     if lat>1, then inject(Q,(v,lat-1));
     else {
          for all edges (v,w) in E
               if dist(w) < dist(v) + weight(v,w) {
                  dist(w) = dist(v) + weight(v,w);
                  parent(w) =  v;
                  if w already in Q, then w.lat = weight(v,w);
                       else inject(Q,(w,weight(v,w));
               }
     }
}
```

We can maintain the latency valyes and the distances in another data structure and use a simple queue structure for Q. Note that a given vertex might be injected to the queue atmost c times. Every edge is traversed atmost once. So the running time is $O(c|V| + |E|)$.

To see the correctness of this algorithm, notice that this algorithm is structurally similar to BFS and Dijkstra's algorithm. For an edge of weight w, we push the vertex in the queue with a latency w. This is a trick that we use to convert the weight of the edges into a suitable form, so that we can still use the queue structure. Think of the latency as splitting up the edges into pieces with weight 1[1]. The difference here is that for this auxiliary edges, we don't need to do any processing. So we just decrement the latency and push it back in the queue.

Problem 4. [**Network of Roads**]

(a) Consider the network of the capital city and the villages. Take the part in the graph corresponding to this. The minimum spanning tree of the relevant part of G would be the required least cost network. We can use either Prim's or Kruskal's algorithm to find out the MST. Using

[1]Note that this is not the same as splitting up the edges to pieces of weight 1. Most of you have split up the edges and have got a running time of $O(|V| + c|E|)$. This is not the same as $O(c|V| + |E|)$. To notice the difference, think of a dense graph where $O(|E|) = O(|V|^2)$.

Prim's algorithm, starting from the capital city, the running time is $O(|V|\log|V| + |E|)$.

(b) If the network still required the old capital city to be a part of it, then there is no change. Otherwise part of the network which is connected to the capital would change.

(c) The bottleneck is given by the heaviest edge in any of the MST. To prove that, we need to observe the Kruskal's algorithm. Note that we are traversing the edges in the ascending order of their weights. Every edge is checked if it reduces the number of forests. We add the edge if and only if it reduces the number of forests. The last edge added will be the heaviest edge in the MST. If there were a network of nodes (not necessarily a tree) which had a smaller bottleneck, we could find a subset of this network, forming a tree, which still has the same bottleneck. Note that this is a spanning tree and has lesser bottleneck. Also note that all the edges in this spanning tree have lesser weights than the Kruskal's MST. This means that the set of all edges with weights less than or equal to the bottleneck are connected. But we had to proceed further when we did Kruskal's, which means that all those edges did not help in reducing the number of forests. This is a contradiction, and so we have proved that the heaviest edge in an MST is indeed the bottleneck. We can use Kruskal's to compute this. We can just keep track of the weight of the edge added last. **Verify** for yourself that running time is within the required $O((|V| + |E|)\log(|V| + |E|))$.

Problem 5. [**Timing with Hourglasses**]

(a) Consider a problem where we have two hourglasses, one of 1 minute and another one of 5 minutes. To measure a time of 4 minutes, the fastest way just takes 4 minutes and 4 turns of the 1 minute hourglass. If we have to minimize the number of turns, we could turn both the hourglasses together and measure time from the stoppage of the 1 minute hourglass to that of the 5 minute hourglass. Here minimizing the number of turns and time taken leads to different answers.

(b) We could modify the algorithm as follows. We could still use the same graph to walk through and find out a valid solution. But the conditions change, so we have to change the distances of each node. When we move to a node in the graph with a positive change we make the edge weight to be 11 o 7 or 5, depending upon which of (a, b, c) changed. For either one of (a, b, c) decreasing on an edge, let the edge weight be 0. This is the only change needed from the last solution.

Note that when we compute only the positive coefficients of $11a+7b+5c$, and neglect the negative coefficients, we get the total time taken to reach the solution. So the new graph is modified taking exactly these parameters into account. Every node is at a distance from $(0, 0, 0)$ depending on what is the total time to execute that triplet (a, b, c). Now we can start walking in the new graph, starting from $(0, 0, 0)$ and execute Dijkstra's with the new weights. We can check if $13 = 11a+7b+5c$ (note that here we need to take all coefficients into account, positive and negative) at each node that we reach. The first node which gives $t = 13$ will be the one which measures 13 minutes in the shortest time.