Problem Set 4

All parts are due Tuesday, November 15, 2016 at 11:59PM.

Name: Dimitris Koutentakis

Collaborators: Carl Unger, Driss Hafdi

Part A

Problem 4-1.

(a) We can modify the graph by transferring the value of the nodes to the weight of the edges that lead to those nodes. In other words, each edge is modified to have weight equal to the value of the node it leads to.

(b) We want to check if we can get from base A to base B in a total cost that is bigger than negative C (the initial amount of armor). That is we want:

$$\sum_{(u,v) in path} W(u,v) > -C$$

An easy way to do that would be to change the sign of all the edges of the graph and then check to see if the shortest path has weight smaller than C. If that is true, then there is at least one such path from base A to base B which speedy can traverse without having to call for help.

An easy way to check for the shortest path would be to just run Bellman Ford on the updated (weights of flipped sign) on the new graph. If the weight of the shortest path found by Bellman Ford, is less than C, then Speedy can traverse at least one path from A to B.

(c) If the robot Speedy has neither made it back, nor called for help, it means it is stuck somewhere. Moreover, since Speedy has not called for help, it means that Speedy has positive armor. Also, since Speedy doesn't exit the loop to continue his journey, it means that he doesn't gain armor, thus the loop is not a positive-weight loop. Thus the loop is a zero-weight loop.

(d) In order to find the set of vertices that Speedy could be, we need to find any non-negative weight loops in the original graph. That is, we can easily implement the Bellman Ford algorithm on the graph of (b) in order to find negative weight cycles.

However, our implementation should not simply stop once a negative weight cycle is found to exist. Once we find one edge in the negative weight cycle (through Bellman Ford), we should any vertices we found to be in a negative weight cycle, and repeat this procedure until there are no more negative-weight cycles.

Once we have a graph with no negative-weight cycles, we should run Bellman-Ford again in order to find the shortest path. Then, we should create a new graph that contains only those nodes on the shortest path and run DFS on this new graph. If the graph is found to have a back edge, that means that is has a cycle and since this cycle is not negative (we have already removed all negative weight cycles), it is a zero weight cycle. Then we should repeat this again, until there are no more cycles left.

Every node on in the zero-weight cycles found is a node that Speedy might be found. This algorithm will run in constant times BF time plus DFS time. Thus the overall runtime of the algorithm will be:

$$O(VE)$$

Problem 4-2.

(a) Since the graph of the nodes is a DAG, in order to find the least-effort path that Jerry must follow while guaranteeing that he will be not be seen, we basically need to do one iteration of Bellman-Ford in a topological sort order and make sure that every 3rd step away he is on a hiding tile.

Once we have a topological sort of all the vertices (just the order in which he can traverse them), we just have to relax each and every one of the edges, while making sure that we satisfy the hiding place requirement. The algorithm would look like this:

```
while x<X, or y<Y:
if (x+y+1=0mod3)and (x+1,y) not in H: pass
if (x+y+1=0mod3)and (x,y+1) not in H: pass
relax((x,y), (x+1,y), w)
relax((x,y), (x,y+1), w)
x=x+1, y=y+1
```

Thus, the time to run this whole process will be:

$$T(n) = O(V + E)$$

(b) In order to solve this problem, we must first create a graph that will easily allow us to extract the information we want. More specifically, we want to find the shortest path that allows Jerry to reach f by only being on H vertices on every 3rd minute. However the graph to use is not completely straightforward.

The graph we will use is analogous to that used in order to find paths of even length. In specific, we will copy the graph three times next to each other, once for 0(mod3), once for 1(mod3) and once for 2(mod3). Every node from the first column (0(mod3)) will be connected to its corresponding children of the second column (1mod3), since it takes one step to go from parent to child. Following the same logic, every node in the second column, will connect to its corresponding children in the third column, and every node in the third column to its corresponding children in the first column. Each of these edges should have the appropriate weight of going from one node to the other. Every node, should also have an edge leading to the same node in the next column with weight 0, signifying that Jerry can stay on the same node for a timestep. Then, we will remove all nodes from the first column (0(mod3)) that are not a hiding node (H) or the starting node (S).

The resulting graph is a graph of all the possible states Jerry can visit. Since we have this graph ready, we simply need to run Dijkstra's algorithm in order to find the least-effort path Jerry can traverse while guaranteeing that he will not be seen by Tom.

Since this graph is just three copies of the original one, the runtime will be the time of Dijkstra's times a constant.

Thus, the time to run this whole process will be:

$$T(n) = O(VlogV + E)$$

Problem 4-3.

(a) In order to find the shortest path to solve a Rubik's cube in initial configuration c, we just have to perform a Breadth First Search from c to a solved state.

Breadth First Search considers nodes level by level, and thus is guaranteed to return the minimum-length sequence that will produce a solved cube.

The time it takes for a BFS to run is simply O(VE).

(b) We should find a way to preprocess the graph so that we can find the shortest path from any vertex c_1 to any vertex c_2 in O(k) time, where k is the length of the sequence.

One way to pre-process such a graph, would be to run a BFS for every node as the starting node. Once this is done, we have all the shortest paths from all the nodes to all the nodes, that is the all pairs shortest paths. Thus we can find the shortest distance from any c_1 to any c_2. This would take time O(V(V+E)), but since V=O(E), the time of this pre-processing is:

$$O(VE)$$

.

Since we now know the shortest path from c_1 to c_2, we just need to output all the steps. As long as we store all the shortest paths, the time of outputting the steps, will take O(1) per step, but there are k steps to the sequence. Thus the time to retrieve the steps from c_1 to c_2, is just simply:

$$O(k)$$

Part B

Problem 4-4.

(a) *Submitted on alg.csail.mit.edu on Nov 15th*

(b) *Submitted on alg.csail.mit.edu on Nov 15th*

(c) Just do the same as for (b), but keep k dictionaries, and update all of them accordingly.