Buffalo Hird
CS124
Homework 2

Problem 1:

For a d-ary heap, we can show how we implement decreaseKey(v, k), deleteMin():, insertKey(v, k) and their time complexity.

decreaseKey(v, k): we implement this by simply modifying the value at the pointer given and then swap this value with one of its parents if they are smaller than k until this no longer possible and the tree is now balanced. The first step is $O(1)$ and the next step is $log_d(n)$. This is true, because k will have up to $log_d(n)$ levels to propogate upward in a d-ary tree of size n. This is because the height of the tree is $log_d(n)$ so we can propogate upward no farther than this.

insertKey(v, k): We can accomplish this by traversing down the heap until we reach the last leaf. The depth of a d-ary tree with $|V|$ nodes should simply be $\frac{log|V|}{logd}$ so insert should take this amount of time + a constant $O(1)$ to do the insertion.

deletemin(): We accomplish this by traversing down the heap until we reach the last leaf and search for the minimum node by viewing each child of the vertices visited at each depth. This has runtime $d * \frac{log|V|}{logd}$ because we are repeating the same process as insertKey(v, k) but now must look at all child nodes s.t. we end up viewing all d nodes.

We want to optimize Dijkstra's Algorithm such that we minimize its runtime. We note that Djikstra's Algorithm contains at most m inserts and n deletes. Therefore since it uses d-ary trees, this takes time $mlog_dn+nlog_dn$. We therefore must choose choose a d such that we minimize this entire expression. We could do this by solving for 0 in the expression's derivative.

Problem 2:

We note that this problem is akin to finding the smallest path of a DAG $G-$ where our original graph is G and we find $G-$ by negating all edges of G. We discussed in class that Floyd Warshall can solve a shortest path in time $O(n^3)$. However, this is slower than we desire and we will therefore devise a new method.

We discussed in class that we can get a topological list ordering of a graph by running DFS on the graph. This is computed in time $O(|V| + |E|)$ and we are sure of its correctness.

We iterate through all edges in time $O(|V|)$ and if all are negative give intial values $-\infty$ to all nodes. Otherwise we can then iterate through this list in time $O(|V|)$ giving initial values 0 to node. These values represent the currently observed longest path to these nodes.

Proceeding this, we iterate (starting with the sources) through each node. For each node, we look at all outgoing edges, comparing the currently assigned longest path for each node with an incoming edge from the current node to the path of getting to the current node + the cost of travelling from the current node along the outgoing edge to the node said edge is incoming to. If for any of these succesive nodes the current cost + incoming edge cost is greater than its current longest path value, we assign its new longest path value to be this sum. So that we can retrieve our path after completing this search, we set the incoming node's parent to be our current node (essentially making a sort of doubly linking).

We repeat this process for each outgoing edge of a node and for each node through our topological list. Since our topological list contains all nodes and we look at all outgoing edges beginning with nodes with no incoming edges, we are guranteed to view each edge and vertex. This process therefore takes time $O(|V| + |E|)$.

Upon completing this, we can simply iterate through our topological list in time $O(|V|)$ and find the item with the highest maximum path value. Since we have iterated through each node topologically, we know this largest value in the list is the largest path value. Since we recorded each node's parent (item before it in the longest path), we can simply iterate through parents until reaching a source node. We record this path $V_n, V_{n-1}, ..., V_k$ to get the shortest path $V_k, ..., V_{n-1}, V_n$. Since there are no cycles there can be at most $|V|$ items in the path, so this process takes time $O(|V|)$. We then reverse this list to get the path from start to finish.

We note that we have the negative and mixed starting case s.t. if there are only negative values then summing proceeds normally. However, if we set initial values of 0, all negatives are less than 0. 0 is an apt start value in the mixed (or positive) case, however, because this precludes missing a path because of previous negative edges reaching a node. In this case, the path will be smaller than the current path of 0 and it will not be worth considering negative edges in our longest path. They are only worth examining where there are no positive (or 0) edges.

Time complexity: We noted in the explanation that this method utilizes multiple steps of time $O(|V|+|E|)$, $O(|V|)$, or $O(|E|)$. Since these are summed, we are only left with some combination of $a*O(|V|)+b*O(|E|)+c*O(|V|+|E|)$ s.t our algorithms runtime is $O(|V| + |E|)$. This is definitely efficient as it is of the same time complexity magnitude of BFS and DFS.

Proof of correctness: We can inductively prove this algorithm. For 1 or 2 nodes the algorithm works by returning 0 or the path between the two nodes. We then must show it works in the $k + 1$ nodes case given it works for k nodes. If it works for k we can return the longest path for k nodes. We can therefore for $k + 1$ return (where F(k) = longest path of k and E(k) = all edges to k) $max(F(k), F(k) + max(E(k + 1)))$. Essentially all this means is either $k + 1$ hasn't produced a longer path, in which case just return the same one as before. If it has produced a longer path that extended $F(k)$, add it to $F(k)$ and return this value. Therefore if we have calculated $F(k)$, we can easily calculate $F(k+1)$ by checking if this occurs.

Problem 3:

We can write this algorithm by essentially modifying BFS such that we iterate through all enemies for each player to see if we can assign teams such that no two enemies are on the same team. We accomplish this by picking an arbitrary initial player and assigning them to one of the teams and add them to our queue. While our queue is not empty we pop off a player (will be already assigned a team) and view its outgoing edge destination nodes. For each of these nodes, if we have already seen it and if so we see if it has been given the same team as the current node and therefore is enemy. If so, we return impossible. If not yet seen, we assign this node to the other team such that enemy links are satisfied in this step. We then mark this node as visited such that we do not need to assign it a team again and add it to the queue. By iterating through this process for each player and its enemy links, we will eventually give a team assignment for all players. Since edges are undirected and we compare all players' team to their enemies' teams, we will see in our BFS modification if we ever make an impossible assignment.

As this algorithm involves viewing each edge and node as does BFS we have a runtime of $O(|V| + |E|)$. We do other small operations such as compare teams, but these operations should be constant time. We can show the correctness of this algorithm by noting that the algorithm will place every two directly connected nodes on opposing teams since it assigns the opposite team to any directly connected edge. Any graph including a cycle or other structure such that a student's team is predefined and this definition conflicts with any other possible arrangement, then we know we would require an additional team for this partitioning to be possible. This therefore precludes all cases where a satisfying two team assignment could not be found. Since this algorithm always finds a correct assignment or returns impossible if more than 2 teams are required, we have successfully reasoned about its correctness.

Problem 4:

We want to make Dijkstra's Algorithm run in time $O(|E| + |V| * l)$ where l is the length of the largest edge in the graph. We can accomplish this by changing the heap sructure the algorithm utilizes such that we are changing the time complexity of its subprocesses. We note that these subprocesses are $|E|$ inserts and $|V|$ deletes. To therefore execute in time $O(|E| + |V| * l)$, we need inserts to be constant such that $|E| * O(1) = O(|E|)$ and all deletes take in total $O(|V| * l)$ time. We find l quickly in time $O(|E|)$. We can then create a list of length $|V| * l$. We do this because then for any path with length n, we can add this path at index n in the list. This is guaranteed to exist, because our list length is the same as there being $|V| * l$, whereas if the longest path was every node connected by the largest edge, the largest path would be $(|V| - 1) * l$. If we had $|V|$ edges in this path, we would then have a cycle.

This structure gives us constant time insert and update as we can index directly into position n for a path of length n without having to maintain any heap invariants or structural costs. Deletemin() is then just linear, executed by iterating through our list until we find the smallest node. This has to be the min node because edges are non-negative such that adding any additional edges to the node will increase its path length. Therefore, we only have to iterate through the list once, giving a runtime of $L * |V|$ for deleteMin.

Therefore pluggin in these values we get runtime $O(|E| + |V| * l)$ as desired.