Lewis Arnsten, Collaborators: Brian McManus

I apologize for this homeworks lateness. Saturday morning I tested inconclusive for COVID and had to get tested again immediately. Today (sunday) I tested positive for COVID. This ordeal severely impacted my ability to do work this weekend.

Problem 1:

This problem is a weighted version of the hopping game. Thus, a greedy approach can no longer be used and we must consider a dynamic programming option. To do so, we will consider a k^{th} problem, which involves finding the maximum amount of reward you can pick up while restricted to the first k spaces ($k < n$). We define $OPT = OPT_n$ as the maximum amount of reward you can pick up while reaching space n, and therefore for the k^{th} problem we aim to find OPT_k. We can say that OPT_{k+1} should be V_{k+1} + best possible k restricted value (OPT_k) within 4 spaces of k+1. So we define our recurrence equation as follows:

$OPT_0 = V_0$

for $0 < k < n$, $OPT_{k+1} = V_{k+1} + \max\{OPT_{k-j}$ for j in [1,2,3,4]$\}$

Or we can write, $OPT_{k+1} = V_{k+1} + \max\{OPT_l\}$ where $l = \max\{OPT_j: k-3 \leq j \leq k\}$

This algorithm ensures that we are maximizing our rewards when hopping to space *k+1*, as given possible hops 1...4, we are choosing the hop for which the start location achieved a maximal score. This will necessarily be the hop that achieves a maximal score at space *k+1*, as for all possible hops, the end location value V_{k+1} is equal. This end location must be included as point n must be included.

```
max_reward = [0...n]
max_reward[0] ← V[0]
For i=1 to n:
        For j in [1,2,3,4]:
                If i - j ≥ 0:
                        max_reward[i] = v[i] + max(max_reward[i-j])
Return max_reward[n]
```

Given the implementation above, max_reward obeys the same base case and recurrence relation as OPT_n. Thus, we can use induction to show that all the values in these two sequences are identical and max_reward[n] = OPT_n = maximum possible reward.

This algorithm starts at the first index and stores each space's optimal cost in an array. When calculating the maximum reward for every possible space, we have to consider the four previous spaces. Thus the algorithm is O(4*n) or O(n) since 4 is a constant.

Problem 2:

a. Starting from node a, Dijkstra's algorithm finds the distance 3 for node c, while the true shortest distance is 2 along path a → e → c.

b.

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	∞	11	11	9	6
b	∞	9	9	8	5	5
c	∞	∞	7	4	4	1
d	∞	2	-1	-1	-4	-7
e	∞	4	4	1	-2	-2

Thus the shortest paths and distances of each node from t are:
a-e-c-d-e-t, length = 6
b-c-d-e-t, length = 5
c-d-e-c-d-t, length = 1
d-e-c-d-e-t, length = -7
e-c-d-e-t, length = -2

Problem 3:

A)

Algorithm/Correctness: To find the shortest such path, we will construct a graph G' such that G' has the same nodes as every G_i and consists of only the edges that occur in all $G_0,...,G_b$. Then, in G' we can find the shortest path from s to t by exploring outward from s in all possible directions one layer at a time (first layer consists of all nodes connected by an edge to s, second consists of any node connected to a node in first layer by an edge, and so on). This is breadth-first search. This algorithm is correct because we have chosen edges that occur in every one of $G_0,...,G_b$, thus if there exists a single path P that is an s-t path in each of the graphs $G_0,...,G_b$ then such a path must also exist in G'.

Polynomial time: when building G' we will check to see if each possible edge is in every E_i. At most we will have to check V(V-1)/2 edges in b graphs, resulting in an upper bound of $O(bV^2)$. To find the shortest path s-t we use breadth-first search, which has a complexity of O(V+E), meaning the total complexity is $O(bV^2)$.

B)

Algorithm/Correctness: Given graphs $G_0,...,G_b$ our goal is to find the set of paths $P_0,...,P_b$ of minimum cost where P_i is a s-t path in G_i for i = 0,..,b. First let's consider path P_b. If P_{b-1} is in G_b then we may use path P_{b-1} and add $l(P_{b-1})$ to the cost function. Otherwise, we consider the shortest s-t path in G_b, $(P_b)'$. Then we must add $l((P_b)')$ and the change cost K to the cost function. Now we will need dynamic programming to ensure that we use a path available in G_b for G_{b-1} to minimize the change penalty K. Thus, we will consider the i^{th} subproblem in graphs $G_0,...,G_i$ with optimal solution OPT_i. To find OPT_i we will find the last time our path changed. Say this change happened between G_i and G_{i+1}, the edges of some path P must be in every graph $G_{i+1},...,G_b$. Now, let G(i,n) $i \leq n$ be the graph with all of the common edges in $G_i,...,G_n$. Let P(i,n) be the length of the shortest path s-t in this graph G(i,n). If no such path s-t exists, then P(i,n) = ∞. If the last time our path changed was between G_i and G_{i+1} then we get that $OPT_b = OPT_i + (b-i)*P(i+1,b)+K$. If there are no changes $OPT_b = (b+1)*P(0,b)$.
Thus our recurrence relation is as follows:
$OPT_b = \min((b+1)*P(0,b), \min(OPT_i + (b-i)*P(i+1,b)+K))$
If we preprocess by computing the graphs G(i,n) and paths P(i,n) for n = i,...,b. Then our algorithm can be written as:

```
C ← list for optimal values
For i=0 to b:
        C[i] = min((i+1)*P(0,i), min(C_n+ (i-n)*P(n+1,i)))
Return C[b]
```

Polynomial time:
Our initial processing computes all graphs G(i,n) and paths P(i,n) for all $1 \leq i \leq n \leq b$. This is $O(b^2)$. Each subgraph has an upper bound of $O(n^2*b)$ as there could be $O(n^2)$ edges in b graphs. The shortest path can be found using breadth-first search, resulting in a total of $O(n^2*b^3)$.