Algorithms #6

Problem 1

Here the intuition is that we need to perform a modified depth first search. We need to modify it because we haven't established the absence of cycles in the path and thus a simple depth first search won't help us solve this problem.

For this algorithm we need auxiliary data structures: two stacks (stack1, stack2) and a maximum reachable arrays of size n (number of nodes). All nodes have a weight and a color field (which can be white, grey, or red). All nodes start out being colored white.

```
DFS-search(G)
  For each u ∈ G.V
    u.color = white
  For each u ∈ G.V
    if u.color is equal to white        ← MaxRed = 0
                                          currMax = 0
      DfS visit(G, u).
```

```
DFS-visit(G, v)
  if v.color == white
    v.color = grey.
  stack1.push(v)
  for each node adjacent to v
      if nodeAdjacent.color == white
          DFS-Visit(G, nodeAdjacent)
  If there are no nodes adjacent or all nodes adjacent are red (keeping track of max in MaxRed)
      v.color = red
      if v.weight > maxRed => maxRed = v.weight
      MaxReachable[v] = maxRed.
      stack1.pop
      return
  else if all nodes adjacent to v are grey or a mix of grey and red.
      stack1.pop
      stack2.push(v)
      currMax = max(max(maxRed, v.weight), currMax)
      if stack1 nonempty.
          DFS-visit(G, stack1.top)
      else
          set everything in stack2 to color red and its index in
          maximum reachable array to currMax. empty stack2.
```

Running Time.

We are executing a modified version of depth first search. In depth first search each node was explored exactly once and each edge was explore exactly once. However, in our algorithm we care about nodes reachable in cycles because this might lead to a maximum reachable node that we would not have discovered if we stopped the search when the children of a node were all grey or a combination of red and grey. Therefore we might have to explore each node twice meaning that we explore each of a nodes edges at most twice as well. This means that if we have a strongly connected graph were the max reachable nodes for every node is the same we must have explored every node twice and thus every edge twice. giving us a running time of $O(2n + 2m)$ which is $O(n+m)$.

Correctness.

Lets perform two proofs on node u to prove correctness. As we have in two situations that any given node U might be in when we visit it.

1. Node U has only red children or no children at all
2. Node U has only grey or red and grey children.

(1.) The base case is that u has no children at all, then clearly u cannot reach any other node except itself and we can assign MaxReachable[v] to v.weight and mark v as red.

Now assume that if u has children all of them are marked red. Because node U can reach its children, trivially, then it can reach whatever its children can reach. Thus the maximum of the MaxReachable of u's children (maxRed) is the maximum node that u can reach, not including itself.

Therefore the maximum node that u can reach is whatever is bigger between (maxRed) and u's weight itself. Thus we can assign this maximum to U and mark U red. Thus when a node becomes red we know for sure that its maximum node reachable is maxReachable[v].

But this does not cover cycles, instances when all given children of a node v are grey or grey and red.

(2.) As shown before we know that maxReachable for any given red node is without a doubt the value of the maximum reachable node from this node.

however when we can only reach grey and red children directly from u, this means we have traversed all the children of u at least once and all of them that could be made red were made red. Therefore, we are in a cycle and all currently grey nodes are a strongly connected component, thus we need to find the maximum weight between all grey children current grey nodes and all the red nodes connected to the grey nodes. This is why we use stack2. When a node is popped of stack 1 we take the maximum between its own weight and the maximum of all the red children thus giving the relative maxima of this grey node. If this maxima is bigger than currMax (the current maximum for all nodes in this strongly connected component) then we update currMax. And then we repeat on the stack1 until we exhaust it. When this happens it is now clear that we have looked at all possible reachable nodes from any node of this strongly connected component and thus currMax holds the absolute maximum reachable value from any given node in the strongly connected component via cycles which are all present in stack2.

Thus we can assign everything in the Stack2 a max reachable of currMax and mark them as red. Thus when we exhaust stack 2, node u will be marked red and have a valid maxreachable.

Since every node in the graph will ultimately be a part of either 1 or 2 case and we have proved that we assign a correct maxReachable for a node u that is in either case, then by the end of DFS-Search all nodes will be marked red and have a correct value in the maxReachable array.

Algorithms #6.

Problem 2

To solve this problem we will use the aid of three matrices of size $O(n^2)$. The first matrix is the adjacency matrix for the original graph G. The second matrix is a distance matrix of the shortest path between any given two nodes in the original graph. To construct this we would run a shortest path algorithm on every node which would run in $O(n^3)$. The third matrix would represent all possible pairs of the form (U, V) for which $U \neq V$ so it would be a matrix for which everything except the diagonal is set to 1. And we also have a queue

```
Construct Schedule Tree ((a,b), (c,d), r).
  enqueue(a,b)
while queue is not empty.
   (U,V) = dequeue
   ThirdMatrix[U,V] = 0
   For each possible pair U, V' st V'≠U and V'≠V
        if ThirdMatrix[U,V'] == 1 and SecondMatrix[U,V'] > r and FirstMatrix[U,V'] == 1.
            Make U, V' a node of the tree (children of U,V).
            Record its parent as U, V
            if (U,V) == (c,d)
               return U,V
            else enqueue U,V
   For each possible pair U', V st V' ≠ b and V' = a
        if ThirdMatrix[U',V] == 1 and SecondMatrix[U',V] ≥ r and FirstMatrix[U,U'] == 1
            Make U', V a node of the tree (child of U, V)
            Record its parent as U,V
            if (U',V) == (c,d)
               return U',V
            else
               enqueue U',V

return NULL.

MinSchedule((a,b)(c,d), r))
    Result = Construct Schedule tree ((a,b),(c,d),r)
    if Result == NULL
       retur false

    else
       Return backtrace from result.
```

Running time.

I have already discussed that the construction of the three matrices is $O(n^3)$ so we are good on that front.

Now the interesting question is how long does Construct Schedule Tree take to compute. In the worst case everything is one step away in the original graph meaning the solution (c,d) is two steps away in the solution tree this means that we would have to explore n-2 permutations of every node in the first loop and n-2 permutations of every node in the second for loop. n-2 because we dont want for a given node u,v to permute ✔ to u or to v (same for the other permutation). And because there are $\binom{n}{2} - n$ possible nodes u,v for which $u \neq v$ then we need to do this two for loops for $O(n^2)$ nodes. Therefore if we explore 2n-4 options for each of the possible $O(n^2)$ nodes our construction Schedule tree function runs in $O(n^3)$ time.

Correctness.

When we create the Third Matrix we assume all possible configurations that the robots can be at any given time (with the exception of the two robots being on the same node at the same time). The idea is that we start constructing the schedule tree from the starting configuration (a,b).

From here we explore all possible configurations that move one robot one step in the original graph with the restriction that that one step exists in the original graph and that the new configuration does not put the two robots within interference distance (r) of each other. and we do this for both robots.

We explore all options in a breadth first search manner as to find the least number of configuration steps (if any) that lead the robots from configuration (a,b) to configuration (c,d). Therefore there could be more than one branch of the tree that would reach (c,d) but the one that we care about is the first one, which by nature of the BFS we are running (by using the queue), is the shortest one.

Lastly by setting ThirdMatrix[u,v] = 0 when we explore u,v we ensure that if we have reached any u,v in n steps no other configuration will reach u,v in steps > n. This would not make sense because if we can reach configuration u,v in >n steps we could have just taken n steps to get to u,v in the first place.

Thus by running the construct schedule tree in a BFS manner with the aid of a queue, one of the computational branches will reach configuration (c, d) if any such branch exists. And if it does exist, this branch will be the shortest branch of computation that would make the robots go from the starting configuration a, b to the end configuration c, d. Therefore the backtrace from node c, d would exhibit the shortest schedule.

Algorithms #5

Problem 3

An arbitrage opportunity is defined as a sequence of multiplications such that

$$A[i_1, i_2] \cdot A[i_2, i_3] \cdots A[i_{k-1}, i_k] \cdot A[i_k, i_1] > 1.$$
(x_1, x_2, x_{n-1}, x_n)

so if we could devise an algorithm that could detect a cycle that has multiplied length greater than 1 we say that the graph has an arbitrage opportunity. otherwise it doesn't. However, what we have in our toolbox is an algorithm that can detect a cycle whose edge sum is negative, namely Bellman Ford. so if we can transform the above multiplication into a negative sum we can use Bellman Ford to solve our original question.

so suppose

$$x_1 \cdot x_2 \cdots x_n > 1$$

if that is the case we are saying

$$x_1 \cdot x_2 \cdots x_n = \frac{p}{q} > 1 \quad \text{so } p > q$$

and $$\frac{1}{\frac{p}{q}} = \frac{q}{p} < 1 \quad \text{because } p > q.$$

so $$\frac{1}{x_1} \cdot \frac{1}{x_2} \cdots \frac{1}{x_n} < 1.$$

and $$\log\left(\frac{1}{x_1} \cdot \frac{1}{x_2} \cdots \frac{1}{x_n}\right) < \log(1) = 0$$

so $$\log\left(\frac{1}{x_1}\right) + \log\left(\frac{1}{x_2}\right) + \cdots + \log\left(\frac{1}{x_n}\right) < 0$$

So if we take our original graph representation (an adjacency Matrix) and transform all edge weights by taking the log of its inverse. Then a negative weight cycle would represent that the graph has an arbitrage opportunity. This means that we can run bellman ford on this new graph, and use its capability as a negative cycle detector to answer if the graph has an arbitrage opportunity. And we have proven in class bellman ford can do this.

Running Time

if we have n nodes then our adjacency matrix is of size n^2 and it would take $O(n^2)$ to apply the proposed transformation.

In class we showed that bellman ford runs in $O(mn)$ time and here we have that m is $O(n^2)$ because we can convert any currency to any other currency, therefore making bellman ford run in $O(n^3)$ time.

Altogether this algorithm runs in $O(n^3)$ time, which is polynomial.