CISC 380 – Homework #4 Solutions

Colin Harthorn

April 13, 2023

Solution to problem 1: This is the solution to the Quoridor problem. In this problem we were assigned to complete the QBoard class in order to properly represent a game of Quoridor using an undirected graph. The methods implemented were canStillWin, getDistTo, and shortestPathTo. I will go over the strategy for each implementation in the order listed.

1. Here is the implementation of canStillWin:

```
def canStillWin(self):
    possible = self.graph.reachable_from((self.player))
    winning = {(0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0),(7,0),(8,0)}
    if(len(winning.intersection(possible)) == 0):
        return False
    return True
```

   We start by creating a set called 'possible' which is created by calling the UGraph method reachable_from, with the player position as input. This returns a set containing every space on the Quoridor board that the player can reach when considering walls. We then make the set 'winning' which contains all 9 positions on the opposite side of the board where the player will win if they reach any of the nodes in the set. We make one check to see if the length of the intersection of 'winning' and 'possible' is length zero. If this is true, then there is no way for the player to reach a winning tile, and we return False. If there is a winning tile the player can reach, we return True.

2. Here is the implementation of getDistTo:

```
def getDistTo(self,coord):
    S = {}
    self.graph.bfs(self.player, S)
    return S[coord]
```

   This function takes a desired end coordinate as input, and will return the distance from the player to that coordinate. We start by creating an empty set 'S' which will contain the set of distances from our breadth first search. We call bfs on self.graph, using the position of the player, and the empty set S as input. We then simply return the value in S at the position of our desired destination coordinate. This value is the distance from the player to the desired coordinate.

3. Here is the implementation of shortestPathTo:

```
def shortestPathTo(self,coord):
    parent = self.graph.bfs(self.player, {})
    tile = coord
    ans = []
    while (tile != None):
        ans.append(tile)
        tile = parent[tile]
    return ans
```

This function takes a desired end coordinate as input, and will return the shortest path from the player to that coordinate in the form of a list of nodes. We start by performing a breadth first search of the Quoridor board using the position of the player and an empty set as input. We then assign the the parent set from bfs to the 'parent' variable in our function. We assign the variable 'tile' to the desired coordinate that was given as input and create an empty list 'ans' to hold the answer. Due to the way a bfs is performed, the only value with a parent value of 'None' will be the input value which, in this implementation, is the player position. We start our while loop by appending the current value of 'tile' to the 'ans' list. We then reassign 'tile' to the parent of the current 'tile'. This continues until we reach the player position. We append the player position to 'ans', 'tile' is reassigned to None which breaks the loop, and we return 'ans' which is then used in QGui to draw the shortest path.

Solution to problem 2: This is the solution to the Speed problem. In this problem we were assigned to determine if there is an cycle that someone, perhaps Keanu Reeves, would be able to drive at the minimum speed or a greater speed indefinitely. The roads are represented with a directed graph containing nodes representing intersections and edge weights representing speeds. In order to determine if there is an indefinite path, we implemented the function find_indefinite_path which takes a DGraph, a start node, and a minimum speed, and returns the indefinite path or none if there is no such path. Here is the implementation of find_indefinite_path:

```
def find_indefinite_path(G, start_node, min_weight):

    edges = G.getEdgeSet()
    for edge in edges:
        if (G.getEdgeWeight(edge[0],edge[1]) < min_weight):
            G.removeEdge(edge[0],edge[1])

    ans = [start_node]
    back_edge = G.detect_cycle(start_node)

    i = 0
    while (back_edge[0] not in ans):
        ans.append(G.getNeighbors(ans[i])[0])
        i=i+1

    ans.append(back_edge[1])

    return ans
```

We start the function by assigning the variable 'edges' to the set of edges in the graph G given as input. We then check the weight of each edge in edges, and remove any edges from G with a weight less than the min_weight. After modifying G, we now have a Directed graph that only has valid edges we can drive on while maintaining minimum speed. We then assign 'ans' to a list containing only the starting node, which we will add to until we have our desired indefinite path.

We assign the variable 'back_edge' to the result of calling detect_cycle on our graph starting from the start node. In order to fill the answer list, we initialize a while loop that will continue as long as the first node from the back edge we found is not in the answer list. In the loop, we call getNeighbors on the graph G, with the value of 'ans' at position i as input. We then append the neighbor at position zero to the answer list and increment i by one. This loop continues until we add the first node of the back edge to the answer variable. Once we have added this value to 'ans', we append the second node that makes up the back edge to the end of the list. Due to the definition of a back edge, this value will already be in the answer list at a prior position, thus completing the indefinite path. We return the complete indefinite path which can then be followed indefinitely to maintain the minimum speed.