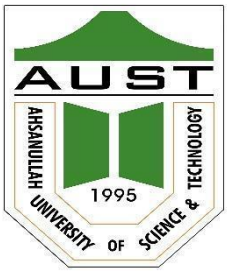


# Ahsanullah University of Science & Technology

## Department of Computer Science & Engineering

**Course No** : **CSE4108**
**Course Title** : **Artificial Intelligence Lab**
**Assignment No** : **02**

**Date of Performance** : **19.12.2022**
**Date of Submission** : **29.12.2022**

**Submitted To** : **Mr. Faisal Muhammad Shah & Mr. Raihan Tanvir**

**Submitted By-**
**Group** : **$C_2$**
**Name** : **Ashiqul Islam**
**Id** : **190104140**
**Section** : **C**

**1) Define a recursive procedure in Python to find the length of a path between two vertices of a directed weighted graph.**

**Solution:**

```
path = []
length = 0

def findPath(graph, source, destination):
    global length
    path.append(source)
    if source == destination:
        return True

    for neighbor in graph[source]:
        if neighbor not in path:
            if(findPath(graph, neighbor, destination)) :
                length += graph[source][neighbor]
                return True
    path.pop()
    return False

graph = {
  'I': {'A': 35, 'B': 45},
  'A': {'C': 22, 'D': 32},
  'B': {'D': 28, 'E': 36},
  'C': {'D': 31, 'G': 47},
  'D': {'G': 30},
  'E': {'G': 26}
}

findPath(graph, 'I', 'G')
print("Path between I to G : " + str(path))
print("Length of the path: " + str(length))
```

**Explanation:** `findPath()` is a recursive function that takes a graph adjacency list, the start node and the destination node between which we have to find the length of the path. The base condition of this recursive function is when the source and the destination node become same, that means we have found the final node of our path while searching through the neighbors of the start node and visiting their neighbor nodes recursively. While doing that if current visiting node is not in the `path` list we add that to the list and add the length of the edge to the global variable `length`. If the end node is not found we pop the last node that we added to the `path` list.

**2) Modify the Python and Prolog codes demonstrated above to find $h_2$ and $h_3$ discussed above.**

**Solution:**

```
# H2 function using Manhattan Distance of tiles in 8-puzzle Problem
def h2():
    goalPos=[(1,1,1), (2,1,2), (3,1,3), (4,2,3), (5,3,3), (6,3,2), (7,3,1), (8,2,1)]
    currPos=[(1,1,2), (2,1,3), (3,2,1), (4,2,3), (5,3,3), (6,2,2), (7,3,2), (8,1,1)]
    h_value = 0

    for i in range(8):
        if(currPos[i] != goalPos[i]):
            h_value += abs(goalPos[i][1] - currPos[i][1]) + abs(goalPos[i][2] -
currPos[i][2])

    print("Heuristic Value (h2) of this instance of 8-puzzle: {0}".format(h_value))
h2()
```

**Explanation:** If the any tile position is not the same as the goal tile position we calculate the Manhattan Distance by taking absolute difference of the goal and the current X and Y coordinate respectively and add their sum to the `h_value` variable. After checking for all the tile position we get the h2 heuristics value for the given instance of 8-puzzle problem.

```
# H3 function of number of attacking pairs of 8-Queen Problem
def h3(state):
    queenPos = [6,1,5,7,4,3,8,1]
    h_value = 0

    for i in range(1, 9):
        for j in range(i+1 , 9):
            # Checking for attacking position face to face
            if(queenPos[i-1] == queenPos[j-1]):
                h_value += 1
            # Checking for attacking position diagonally
            if(abs(i - j) == abs(queenPos[i-1] - queenPos[j-1])):
                h_value += 1

    print("Heuristic value (h3) for this instance of 8-Queen Problem:
{0}".format(h_value))

h3()
```

**Explanation:** The position of each queen is given in the `queenPos` list. We run a nested loop to simulate the chess board as a matrix. Here, `i` denotes the X coordinate in the board (or the queen no. if we follow the context of the given material). So, `queenPos[i]` will give the Y coordinate of ith queen (or the row where ith queen is if we follow the context of the given material). To check if a pair of queens are in face to face attacking position, we will just have to check if they are in the same row or not. Again, for a pair in attacking position diagonally we have to check if the difference of X coordinate and the difference of Y coordinate are same of not. As in a diagonal of a matrix, the absolute difference of row number and absolute difference of column number are same.