**Machine Learning: Assignment 3**

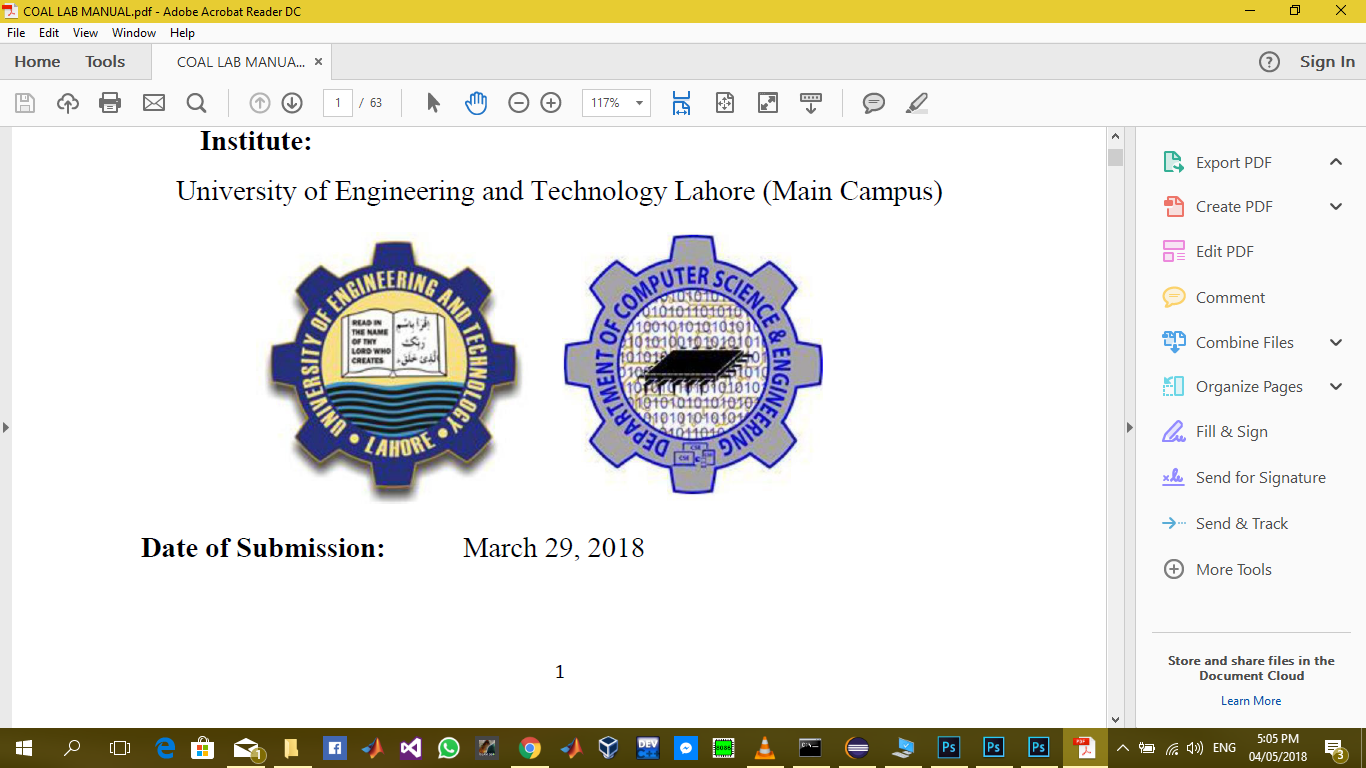
# Submitted to:

Dr. Awais Hassan

# Submitted by:

Komal Shehzadi (2020-MSCS-590)





**Assignment 3 Data Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Maze Name | Search Function | Nodes Expanded | Cost |  |
| Tiny Maze | BFS | 15 | 8 |  |
| Tiny Maze | DFS | 14 | 10 |  |
| Tiny Maze | UCS | 16 | 8 |  |
| Tiny Maze | AStarSearch | 8 | 8 |  |
| Medium Maze | BFS | 269 | 68 |  |
| Medium Maze | DFS | 144 | 130 |  |
| Medium Maze | UCS | 275 | 68 |  |
| Medium Maze | AStarSearch | 81 | 76 |  |
| Big Maze | BFS | 620 | 210 |  |
| Big Maze | DFS | 390 | 210 |  |
| Big Maze | UCS | 620 | 210 |  |
| Big Maze | AStarSearch | 466 | 210 |  |

**Challenge 1:**

DFS graph algorithm:

def isExistInQ(self, item):

    for p in self.list:

        if(item == p):

            return True

    return False

def depthFirstSearch(problem):

    """

    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the

    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to

    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())

    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))

    print("Start's successors:", problem.getSuccessors(problem.getStartState()))

    """

    currentState = problem.getStartState()

    F = util.Stack()

    E = []

    a = []

    paths= {}

    F.push(currentState)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

        E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            successors = problem.getSuccessors(currentState)

            #*successors.reverse()*

            for child in  successors:

                if (child[0] not in E) and (not isExistInQ(F, child[0])):

                    F.push(child[0])

                    a.append(child[1])

                    paths[child[0]] = path + a

                    a = []

    return []

**Challenge 2:**

**MediumMaze:**

* Adding in same order as provided by getSuccessors function.

**Code:**

def isExistInQ(self, item):

    for p in self.list:

        if(item == p):

            return True

    return False

def depthFirstSearch(problem):

    """

    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the

    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to

    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())

    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))

    print("Start's successors:", problem.getSuccessors(problem.getStartState()))

    """

    currentState = problem.getStartState()

    F = util.Stack()

    E = []

    a = []

    paths= {}

    F.push(currentState)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

        E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            successors = problem.getSuccessors(currentState)

            for child in  successors:

                if (child[0] not in E) and (not isExistInQ(F, child[0])):

                    F.push(child[0])

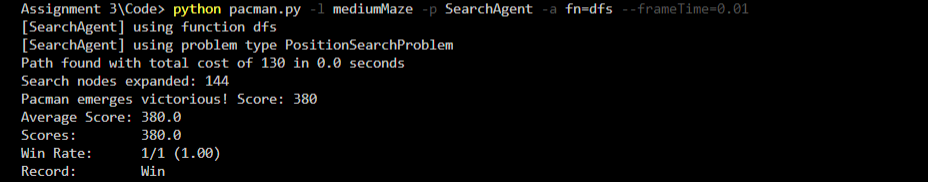
                    a.append(child[1])

                    paths[child[0]] = path + a

                    a = []

    return []

**Results:**



* Adding in reverse order.

**Code:**

def isExistInQ(self, item):

    for p in self.list:

        if(item == p):

            return True

    return False

def depthFirstSearch(problem):

    """

    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the

    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to

    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())

    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))

    print("Start's successors:", problem.getSuccessors(problem.getStartState()))

    """

    currentState = problem.getStartState()

    F = util.Stack()

    E = []

    a = []

    paths= {}

    F.push(currentState)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

        E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            successors = problem.getSuccessors(currentState)

            successors.reverse()

            for child in  successors:

                if (child[0] not in E) and (not isExistInQ(F, child[0])):

                    F.push(child[0])

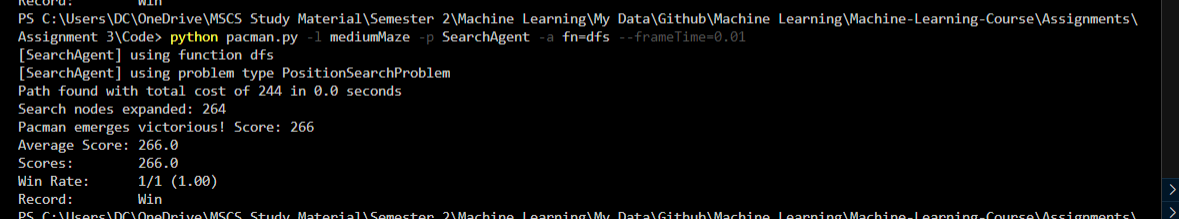
                    a.append(child[1])

                    paths[child[0]] = path + a

                    a = []

    return []

**Results:**



**Challenge 03:**

**Depth First Search(DFS):**

def isExistInQ(self, item):

    for p in self.list:

        if(item == p):

            return True

    return False

def depthFirstSearch(problem):

    """

    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the

    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to

    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())

    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))

    print("Start's successors:", problem.getSuccessors(problem.getStartState()))

    """

    currentState = problem.getStartState()

    F = util.Stack()

    E = []

    a = []

    paths= {}

    F.push(currentState)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

        E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            successors = problem.getSuccessors(currentState)

            for child in  successors:

                if (child[0] not in E) and (not isExistInQ(F, child[0])):

                    F.push(child[0])

                    a.append(child[1])

                    paths[child[0]] = path + a

                    a = []

    return []

**Breadth First Search(BFS):**

def breadthFirstSearch(problem):

    """Search the shallowest nodes in the search tree first."""

    currentState = problem.getStartState()

    F = util.Queue()

    E = []

    a = []

    paths= {}

    F.push(currentState)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

        E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            for child in  problem.getSuccessors(currentState):

                if (child[0] not in E) and (not isExistInQ(F, child[0])):

                    F.push(child[0])

                    a.append(child[1])

                    paths[child[0]] = path + a

                    a = []

    return []

**Comparison Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Maze Name | Search Function | Nodes Expanded | Cost |  |
| Tiny Maze | BFS | 15 | 8 |  |
| Tiny Maze | DFS | 14 | 10 |  |
| Medium Maze | BFS | 269 | 68 |  |
| Medium Maze | DFS | 144 | 130 |  |
| Big Maze | BFS | 620 | 210 |  |
| Big Maze | DFS | 390 | 210 |  |

**Challenge 04:**

UCS algorithm implemented in code, which finds a solution, which takes minimum cost and maximum score.

def isExistInPQ(self, item):

    for p in self.heap:

        if(item == p):

            return True

    return False

def uniformCostSearch(problem):

    """Search the node of least total cost first."""

    currentState = problem.getStartState()

    F = util.PriorityQueue()

    E = []

    a = []

    paths= {}

    F.push(currentState, 0)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

        E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            for child in  problem.getSuccessors(currentState):

                if (child[0] not in E) and (not isExistInPQ(F, child[0])):

                    F.push(child[0], child[2])

                    a.append(child[1])

                    paths[child[0]] = path + a

                    a = []

                elif isExistInPQ(F, child[0]):

                    F.update(child[0], child[2])

    return []

**Challenge 05:**

Astar heuristic algorithm implemented with absolute distance heuristic.

def isExistInPQ(self, item):

    for p in self.heap:

        if(item == p):

            return True

    return False

def manHattanHeuristic(state, problem=None):

    goalState = problem.goal

    i = abs(state[0][0] - goalState[0]) + abs(state[0][1] - goalState[1])

    p = state[2] + i

    return p;

def aStarSearch(problem, heuristic=nullHeuristic):

    """Search the node that has the lowest combined cost and heuristic first."""

    currentState = problem.getStartState()

    F = util.PriorityQueue()

    E = []

    a = []

    paths= {}

    goalState = problem.goal

    i = abs(currentState[0] - goalState[0]) + abs(currentState[1] - goalState[1])

    F.push(currentState,  i)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            for child in  problem.getSuccessors(currentState):

                if (child[0] not in E) and (not isExistInPQ(F, child[0])):

                    #*using absolute distance heuristic*

                    p = manHattanHeuristic(child, problem);

                    F.push(child[0], p)

                    a.append(child[1])

                    paths[child[0]] = path + a

                    a = []

                elif isExistInPQ(F, child[0]):

                    F.update(child[0], child[2])

    return []

**Challenge 06:**

1. **Fill following table with the information**.

|  |  |  |
| --- | --- | --- |
|  | **A\* Heuristic** | **UCS** |
| **Total Cost** | 210 | 210 |
| **Nodes Expanded** | 620 | 466 |
| **Score** | 300 | 300 |

1. **Why Node Expanded is Greater in UCS and Less in A\* Heuristic?**

UCS is only focusing on cost effective path whilst A\* algorithm keeps cost and distance both into account by updating the path by absolute distance factor.

1. **Compare other parameters and give reason why they greater/less/equal/.**

Both algorithms are focusing on cost and score equally that’s why they are equal and nodes expanded are lesser for A\* algorithm because of distance heuristic we are using in it.

**Challenge 07:**

**Euclidean Code:**

def EuclideanHeuristic(state, problem=None):

    goalState = problem.goal

    i = math.sqrt((state[0][0] - goalState[0])\*\*2 + (state[0][1] - goalState[1])\*\*2)

    p = state[2] + i

    return p;

def aStarSearch(problem, heuristic=nullHeuristic):

    """Search the node that has the lowest combined cost and heuristic first."""

    currentState = problem.getStartState()

    F = util.PriorityQueue()

    E = []

    a = []

    paths= {}

    goalState = problem.goal

    #*i = abs(currentState[0] - goalState[0]) + abs(currentState[1] - goalState[1])*

    i = math.sqrt((currentState[0] - goalState[0])\*\*2 + (currentState[1] - goalState[1])\*\*2)

    F.push(currentState,  i)

    paths[currentState] = []

    while not F.isEmpty():

        currentState = F.pop()

        path = paths[currentState]

        E.append(currentState)

        if problem.isGoalState(currentState):

            return path

        else:

            for child in  problem.getSuccessors(currentState):

                if (child[0] not in E) and (not isExistInPQ(F, child[0])):

                    #*using absolute distance heuristic*

                    #*p = manHattanHeuristic(child, problem);*

                    #*using Euclidean Heuristic*

                    p = EuclideanHeuristic(child, problem)

                    F.push(child[0], p)

                    a.append(child[1])

                    paths[child[0]] = path + a

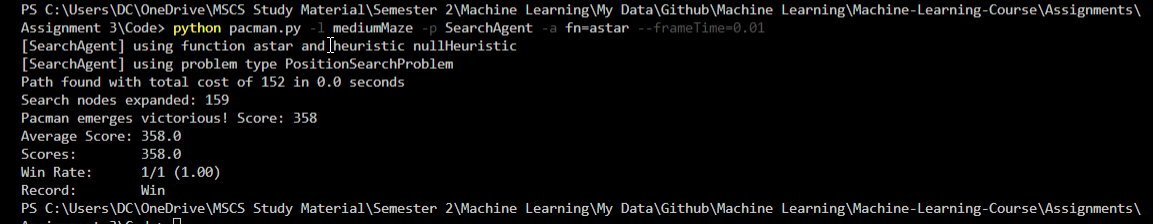
                    a = []

                elif isExistInPQ(F, child[0]):

                    F.update(child[0], child[2])

    return []

**mediumMaze Results::**



**Comparison Table**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Euclidean** | **Manhattan** | **UCS** |
| **Total Cost** | 152 | 76 | 68 |
| **Nodes Expanded** | 159 | 81 | 275 |
| **Score** | 358 | 434 | 442 |

Manhattan distance is better for mediumMaze problem states. Distance heuristc varies their efficiency based on the problem states provides that is why the results are different.