Name:Coffman,Ethan

Email: [ejc012@uark.edu](mailto:ejc012@uark.edu)

ID: 100468386

Late Days: 2

**Question 1:**

a)

def DFS(A):

target = (len(A)-1,len(A[0])-1)

def recurse(x,y,path):

if(A[x][y] == 0):

return -1

path.append((x,y))

if (x,y) == target:

return path

if x+1 < len(A) and not (x+1,y) in path:

result = recurse(x+1,y,path)

if result != -1:

return result

if y+1 < len((A)[0]) and not (x,y+1) in path:

result = recurse(x,y+1,path)

if result != -1:

return result

if x-1 > 0 and not (x-1,y) in path:

result = recurse(x-1,y,path)

if result != -1:

return result

if y-1 > 0 and not (x,y-1) in path:

result = recurse(x,y-1,path)

if result != -1:

return result

return -1

result = recurse(0,0,[])

if result == -1:

print(result)

output = ""

for i in range(len(result)):

output += f"{result[i]}"

if(i == len(result)-1):

print(output)

return

output += "->"

b)

def BFS(A):

queue = []

expended = []

fringe = []

target = (len(A)-1,len(A[0])-1)

queue.append((0,0))

fringe.append([(0,0)])

result = []

while len(queue) != 0:

node = queue.pop(0)

x,y = node

expended.append(node)

if(node == target):

print(node)

for path in fringe:

if path[-1] == target:

result = path

break

neighbors = []

if x+1 < len(A) and not (x+1,y) in expended and A[x+1][y] == 1:

queue.append((x+1,y))

neighbors.append((x+1,y))

if y+1 < len(A[0]) and not (x,y+1) in expended and A[x][y+1] == 1:

queue.append((x,y+1))

neighbors.append((x,y+1))

if x-1 > 0 and not (x-1,y) in expended and A[x-1][y] == 1:

queue.append((x-1,y))

neighbors.append((x-1,y))

if y-1 > 0 and not (x,y-1) in expended and A[x][y-1] == 1:

queue.append((x,y-1))

neighbors.append((x,y-1))

for path in fringe:

if path[-1] == node:

fringe.remove(path)

for neighbor in neighbors:

new\_path = path.copy()

new\_path.append(neighbor)

fringe.append(new\_path)

if result == []:

print(-1)

return

output = ""

for i in range(len(result)):

output += f"{result[i]}"

if(i == len(result)-1):

print(output)

return

output += "->"

**Question 2:**

a.1)

| Expended Node | Fringe/Frontier | Visited |
| --- | --- | --- |
|  | {[B]} | B |
| B | {[B,A],[B,E]} | E, A |
| A | {[B,A,D]} | D |
| D | {[B,A,D,G],[B,A,D,F]} | G, F |
| F |  |  |
| G | [B,A,D,G] |  |

a.2) [B,A,D,G]

a.3) B,A,D,F,G

a.4) 2+3+1 = 6

b.1)

| Expended Node | Fringe/Frontier | Visited |
| --- | --- | --- |
|  | {[B]} | B |
| B | {[B,A],[B,E]} | A, E |
| A | {[B,A,D]} | D |
| E | {[B,E,C],[B,E,F]} | C,F |
| D | {[B,A,D,G]} | G |
| C | {[B,E,C,A]} | A |
| F |  |  |
| G | [B,A,D,G] |  |

b.2) [B,A,D,G]

b.3) B,A,E,D,C,F,G

b.4) 2+3+1 = 6

c.1)

|  | [B,0] |
| --- | --- |
| B | [B,A,2],[B,E,2] |
| A | [B,A,D,5] |
| E | [B,E,F,4],[B,E,C,5] |
| F |  |
| D | [B,A,D,G,6] |
| C | [B,E,C,A,7] |
| G |  |

c.2) [B,A,D,G]

c.3) B,A,E,F,D,C,G

c.4) 2+3+1 = 6

**Question 3:**

a.1) [S,A,B,C,D,G]

a.2) 3+1+2+3+1=10

b.1) [S,A,D,G]

b.2) 3+5+1=9

c.1) [S,B,D,G]

c.2) 2+3+1=6

d.1) [S,B,D,G]

d.2) 2+3+1=6

**Question 4:**

| Path to State  Expanded | Length of  Path g(n) | Total Estimated  Cost f(n) | Expanded List |
| --- | --- | --- | --- |
| M | 0 | 5 | (M) |
| M,O | 3 | 4 | (M,O) |
| M,N | 1 | 5 | (M,O,N) |
| M,O,Q | 5 | 6 | (M,O,N,Q) |
| M,O,Q,P | 6 | 6 | (M,O,N,Q,P) |

**Question 5:**

**DFS:**

**def depthFirstSearch(problem):**

**stack = util.Stack()**

**stack.push(problem.getStartState())**

**fringe = [[problem.getStartState()]]**

**visited = []**

**result = []**

**while not(stack.isEmpty()):**

**node = stack.pop()**

**visited.append(node)**

**if problem.isGoalState(node):**

**for i in range(len(fringe)):**

**if fringe[-i][-1] == node:**

**result = fringe[-i]**

**break**

**options = problem.getSuccessors(node)**

**neighbors = []**

**for option in options:**

**loc = option[0]**

**if not problem.isWall(loc) and not loc in visited:**

**neighbors.append(loc)**

**stack.push(loc)**

**for path in fringe:**

**if path[-1] == node:**

**fringe.remove(path)**

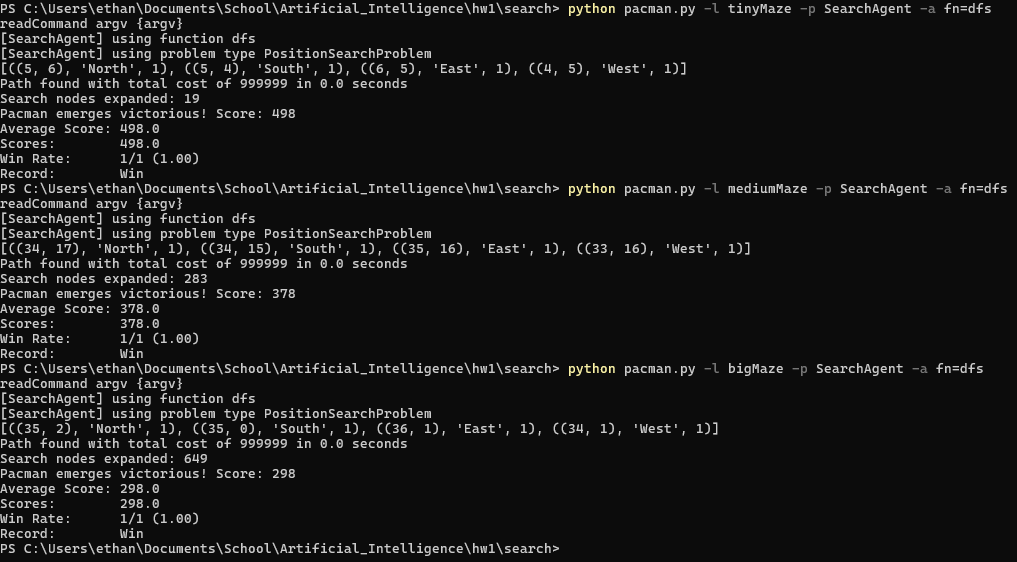
**for neighbor in neighbors:**

**new\_path = path.copy()**

**new\_path.append(neighbor)**

**fringe.append(new\_path)**

**return convert(problem,result)**

****

**BFS:**

**def breadthFirstSearch(problem):**

**queue = util.Queue()**

**queue.push(problem.getStartState())**

**fringe = [[problem.getStartState()]]**

**visited = []**

**result = []**

**while not(queue.isEmpty()):**

**node = queue.pop()**

**visited.append(node)**

**if problem.isGoalState(node):**

**for i in range(len(fringe)):**

**if fringe[i][-1] == node:**

**result = fringe[i]**

**break**

**options = problem.getSuccessors(node)**

**neighbors = []**

**for option in options:**

**loc = option[0]**

**if not problem.isWall(loc) and not loc in visited:**

**neighbors.append(loc)**

**queue.push(loc)**

**for path in fringe:**

**if path[-1] == node:**

**fringe.remove(path)**

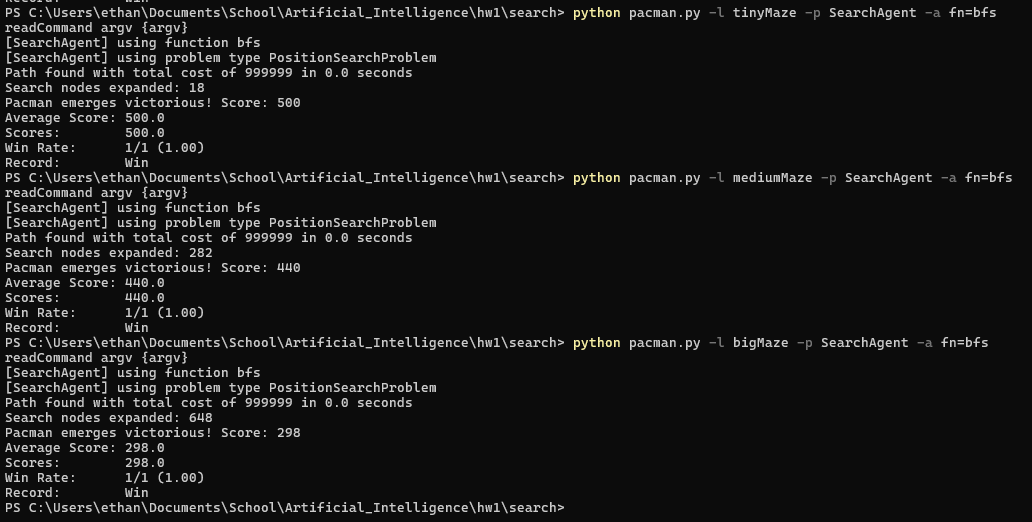
**for neighbor in neighbors:**

**new\_path = path.copy()**

**new\_path.append(neighbor)**

**fringe.append(new\_path)**

**return convert(problem,result)**

****

**UCS:**

**def uniformCostSearch(problem):**

**from game import Directions**

**n = Directions.NORTH**

**s = Directions.SOUTH**

**queue = util.PriorityQueue()**

**queue.push([problem.getStartState(),touch\_wall(problem),0],0)**

**visited = {}**

**while not queue.isEmpty():**

**node,actions,priority = queue.pop()**

**if not node in visited or priority < visited[node]:**

**visited[node] = priority**

**else:**

**continue**

**if problem.isGoalState(node):**

**return actions**

**for neighbor in problem.getSuccessors(node):**

**node,action,cost = neighbor**

**if problem.isWall(node):**

**continue**

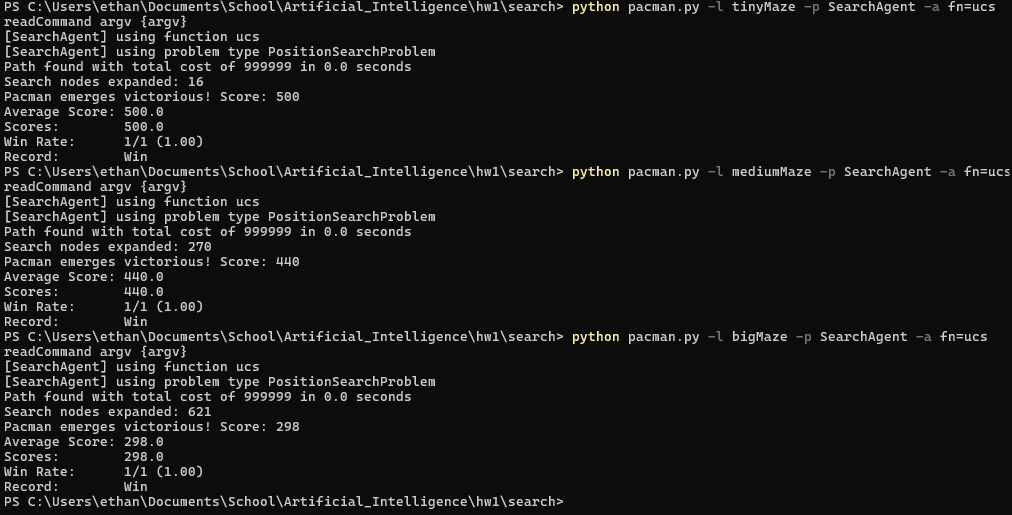
**new\_actions = actions.copy()**

**new\_actions.append(action)**

**new\_priority = priority + cost**

**queue.push([node,new\_actions,new\_priority],new\_priority)**

**util.raiseNotDefined()**

****

**A\*:**

**def aStarSearch(problem, heuristic=nullHeuristic):**

**from game import Directions**

**w = Directions.WEST**

**e = Directions.EAST**

**queue = util.PriorityQueue()**

**queue.push([problem.getStartState(),touch\_wall(problem),0],0)**

**visited = {}**

**while not queue.isEmpty():**

**node,actions,priority = queue.pop()**

**if not node in visited or priority < visited[node]:**

**visited[node] = priority**

**else:**

**continue**

**if problem.isGoalState(node):**

**return actions**

**for neighbor in problem.getSuccessors(node):**

**node,action,cost = neighbor**

**if problem.isWall(node):**

**continue**

**new\_actions = actions.copy()**

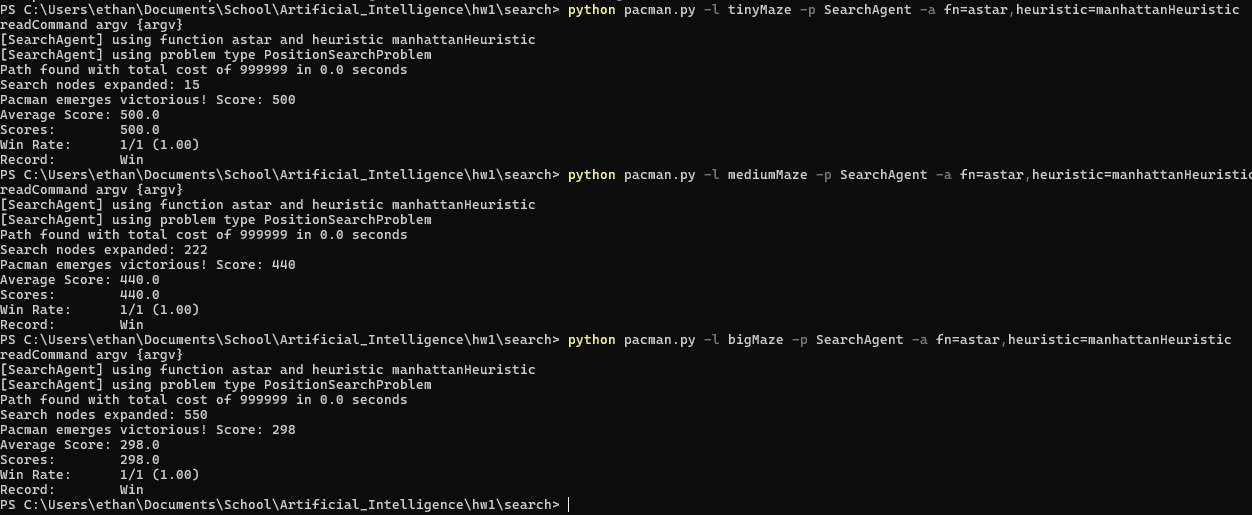
**new\_actions.append(action)**

**path\_cost = priority + cost**

**new\_priority = priority + heuristic(node,problem)**

**queue.push([node,new\_actions,path\_cost],new\_priority)**

**util.raiseNotDefined()**

****