VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

INTRODUCTION TO ARTIFICIAL INTELLIGENCE (CO3061)

Assigment 1

Developing search algorithms, with a reference framework from UCB

Advisor:	Le Thanh Sach	
Students:	Nguyen Tien Hung	- 2252280
	Nguyen Minh Khoi	- 2252377
	Thai Quang Phat	- 2252606

HO CHI MINH CITY, OCTOBER 2024

Contents

1 Introduction

Pac-Man, originally called Puck Man in Japan, is a 1980 maze video game developed and released by Namco for arcades. In North America, the game was released by Midway Manufacturing as part of its licensing agreement with Namco America. The player controls Pac-Man, who must eat all the dots inside an enclosed maze while avoiding four colored ghosts. Eating large flashing dots called "Power Pellets" causes the ghosts to temporarily turn blue, allowing Pac-Man to eat them for bonus points.

In this project, we try to teach pacman how to find the food in some particular algorithm (e.g, one or many foods in the maze) and may try to teach it to take as few steps as possible.

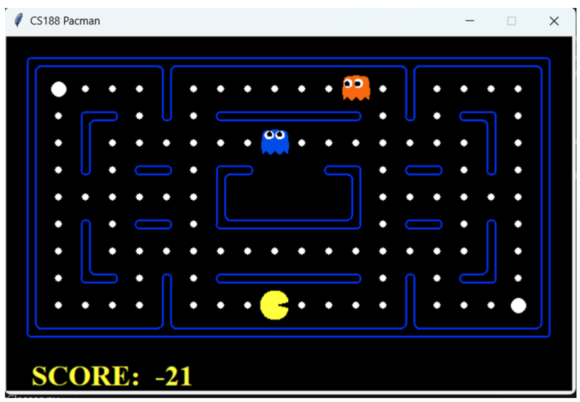

Figure 1: Game Pacman from the project UCB

The project UCB about Pacman can be divided into 2 parts: Implementing the search.py file (The first 4 questions about searching algorithm) and implementing the searchAgent.py file (The next 4 questions about Agent and implementing some heuristic function).

Files to Edit and Submit: We will have to fill in portions of search.py and searchAgents.py during the assignment. And we will not be allowed to modify any files.

2 General search algorithm

The next 4 questions from the project which are Depth-First-Search, Breadth-First-Search, Uniform-Cost-Search and Astar-Search actually have something in common. They just differ in how we choose the data-structure of the frontier or the fringe appropriately.

```
function GRAPH-SEARCH(problem,fringe) return a solution or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if EMPTY?(fringe) then return failure
        node ← REMOVE-FIRST(fringe)
        if GOAL-TEST[problem] applied to STATE[node] succeeds
                then return SOLUTION(node)
        if STATE[node] is not in closed then
                add STATE[node] to closed
                fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Figure 2: A general searching algorithm

```
function EXPAND(node,problem) return a set of nodes
    successors ← the empty set
    for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        STATE[s] ← result
        PARENT-NODE[s] ← node
        ACTION[s] ← action
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action,s)
        DEPTH[s] ← DEPTH[node]+1
        add s to successors
    return successors
```

Figure 3: A general expanding function for the searching algorithm

3 File search.py

3.1 Question 1 (3 points): Finding a Fixed Food Dot using Depth First Search

```python
fringe = Stack()
    start = problem.getStartState()
    fringe.push(start)
    parent_find = {start:None} # hashmap
    visited = {}

    while not fringe.isEmpty():

        currentState = fringe.pop()

        visited[currentState] = 1
        if problem.isGoalState(currentState):
            return_path = []
            while True:
                if parent_find[currentState] != None:
                    return_path.append(parent_find[currentState][1])
                    currentState = parent_find[currentState][0]
                else:
```

```
                    break

                return list(reversed(return_path))

            temp_list = problem.getSuccessors(currentState)
            for next_state,direction,cost in temp_list:
                if next_state in visited:
                    continue
                else:
                    fringe.push(next_state)
                    parent_find[next_state]=(currentState,direction)
```

In question 1 about the Depth-First-Search algorithm, we will use the Stack() data-structure for the fringe. Since we want to find as deep as we can. Whenever we cannot go further or find a solution. Backtrack to the closest node that we haven't searched for the other successor.
we am also using 1 more data structure than the set visited. Because we can see that the grandchild of the child node has a path to another child, it will modify the path (also the parent of that node). So the visited-set can be used to keep nodes that have already expanded and we will not handle those nodes again.

Table 1: Result for Depth-First-Search

Layout	Total Cost	Nodes Expanded	Score
tinyMaze	10	15	500
mediumMaze	130	146	380
bigMaze	210	390	300

3.2 Question 2 (3 points): Finding a Fixed Food Dot using Breadth First Search

```
fringe = Queue()
    start = problem.getStartState()
    fringe.push(start)
    parent_find = {start:None} # hashmap

    while not fringe.isEmpty():

        currentState = fringe.pop()

         if problem.isGoalState(currentState):
            return_path = []
            while True:
                if parent_find[currentState] != None:
                    return_path.append(parent_find[currentState][1])
```

```
                    currentState = parent_find[currentState][0]
                else:
                    break

            return list(reversed(return_path))

        temp_list = problem.getSuccessors(currentState)
        for next_state,direction,cost in temp_list:
            if next_state in parent_find:
                continue
            else:
                fringe.push(next_state)
                parent_find[next_state]=(currentState,direction)
```

In question 2, with the Breadth-First-Search algorithm, we will use the Queue() data-structure for the fringe. Because we can clearly understand that. The BFS searches all the child nodes before continuing with the grand-child nodes, so using Queue() would be an excellent choice for us.
In this problem, differ from question 1, we don't use the visited-set algorithm any more. Because thinking that if the child-node expanded a grand-child node which was already in the queue() or handled. So we have to pass it because the old path will have lower cost. (Remember each step in the DFS and BFS is default 1)

Table 2: Result for Breadth-First-Search

Layout	**Total Cost**	**Nodes Expanded**	**Score**
tinyMaze	8	15	502
mediumMaze	68	269	442
bigMaze	210	620	300

The comparison with the DFS algorithm shows that the cost is reduced, but in return, BFS requires expanding more nodes, resulting in a lower score.

3.3 Question 3 (3 points): Varying the Cost Function

```
fringe = PriorityQueue()
    start = problem.getStartState()
    fringe.push((start,0),0)
    parent_find = {start:None} # hashmap
    while not fringe.isEmpty():

        (currentState,cost_current) = fringe.pop()
        if problem.isGoalState(currentState):
            return_path = []
            while True:
                if parent_find[currentState] != None:
```

```
                    return_path.append(parent_find[currentState][1])
                    currentState = parent_find[currentState][0]
                else:
                    break

            return list(reversed(return_path))

        temp_list = problem.getSuccessors(currentState)
        for next_state,direction,cost in temp_list:
            if next_state not in parent_find:
                fringe.update((next_state,cost+cost_current),cost+cost_current)
                parent_find[next_state]=(currentState,direction,cost+cost_current)
            elif parent_find[next_state] == None:
                continue
            elif parent_find[next_state][2] > cost+cost_current:
                fringe.update((next_state,cost+cost_current),cost+cost_current)
                parent_find[next_state]=(currentState,direction,cost+cost_current)
```

In question 3, with the Uniform-Cost-Search algorithm, we will use the PriorityQueue() data-structure for the fringe. Because we need to search from the lowest cost path to the highest cost path. If we use the vector or a simple array, when a new path is inserted. We have to sort the vector or array and lead to the time complexity nearly n^2 or $n.\log(n)$. The PriorityQueue() will cost the time $\log(n)$ whenever we need to insert a new path into the queue.

Table 3: Result for Uniform-Cost-Search

Layout	Total Cost	Nodes Expanded	Score
mediumMaze	68	269	442
mediumDottedMaze	1	186	646
mediumScaryMaze	68719479864	108	418

Seeing that the cost from the mediumDottedMaze and the mediumScaryMaze is very low and very high. Since we call the problem on the class StayEastSearchAgent, class StayWestSearchAgent. Since the first class using the function cost: 0.5^x and the second class using the function cost: 2^x.

```
class StayEastSearchAgent(SearchAgent):
    """
    An agent for position search with a cost function that penalizes being in
    positions on the West side of the board.

    The cost function for stepping into a position (x,y) is 1/2^x.
```

```
    """
    def __init__(self):
        self.searchFunction = search.uniformCostSearch
        costFn = lambda pos: .5 ** pos[0]
        self.searchType = lambda state: PositionSearchProblem(state, costFn
    , (1, 1), None, False)

class StayWestSearchAgent(SearchAgent):
    """
    An agent for position search with a cost function that penalizes being
    in
    positions on the East side of the board.

    The cost function for stepping into a position (x,y) is 2^x.
    """
    def __init__(self):
        self.searchFunction = search.uniformCostSearch
        costFn = lambda pos: 2 ** pos[0]
        self.searchType = lambda state: PositionSearchProblem(state, costFn
    )
```

3.4 Question 4 (3 points): A* search

```
fringe = PriorityQueue()
    start = problem.getStartState()
    fringe.push((start,0),0)
    parent_find = {start:None} # hashmap

    while not fringe.isEmpty():

        (currentState,cost_current) = fringe.pop()



        if problem.isGoalState(currentState):
            return_path = []
            while True:
                if parent_find[currentState] != None:
                    return_path.append(parent_find[currentState][1])
                    currentState = parent_find[currentState][0]
                else:
                    break

            return list(reversed(return_path))

        temp_list = problem.getSuccessors(currentState)
        for next_state,direction,cost in temp_list:


```

```
        heuristic_value = heuristic(next_state,problem)

        if next_state not in parent_find:
            fringe.update((next_state,cost+cost_current),cost+cost_current+heuristic_value)
            parent_find[next_state]=(currentState,direction,cost+cost_current)
        elif parent_find[next_state] == None:
            continue
        elif parent_find[next_state][2] > cost+cost_current:
            fringe.update((next_state,cost+cost_current),cost+cost_current+heuristic_value)
            parent_find[next_state]=(currentState,direction,cost+cost_current)
```

In question 4, upgrade the algorithm search UCS above, we estimate the whole length of the path using the already computed cost path plus the estimate from that point to the goal. Using the heuristic function as an argument of the function.
It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n). Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

Table 4: Show the different between UCS and Astar(Astar using manhattan heuristic function)

Algorithm	**Layout**	**Total Cost**	**Nodes Expanded**	**Score**
UCS	mediumMaze	68	269	442
Astar	mediumMaze	68	222	442
UCS	bigMaze	210	620	300
Astar	bigMaze	210	549	300

We can clearly see that the performance using the heuristic function will improve the algorithm in expanding fewer nodes and also lead to a better run time.

Table 5: Show the general 4 searching algorithm

Algorithm	**Layout**	**Total Cost**	**Nodes Expanded**	**Score**
DFS	bigMaze	210	390	300
BFS	bigMaze	210	620	300
UCS	bigMaze	210	620	300
Astar	bigMaze	210	549	300

4 File searchAgents.py

4.1 Question 5 (3 points): Finding All the Corners

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState: pacman.GameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        #self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))


        # EDIT
        self.startingPosition = (startingGameState.getPacmanPosition(),self
    .corners)
        # EDIT


        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded


    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman
    state
        space)
        """
        "*** YOUR CODE HERE ***"

        return self.startingPosition
        #util.raiseNotDefined()

    def isGoalState(self, state: Any):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        size = len(list(state[1]))
```

```python

        return size == 0

        #util.raiseNotDefined()

    def getSuccessors(self, state: Any):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
            For a given state, this should return a list of triples, (successor,
            action, stepCost), where 'successor' is a successor to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """

        currentPosition,tuple_corner = state

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]
            "*** YOUR CODE HERE ***"
            x,y = currentPosition
            dx,dy = Actions.directionToVector(action)
            nextx,nexty = int(x+dx),int(y+dy)
            hitsWall = self.walls[nextx][nexty]

            if not hitsWall:
                new_tuple = tuple(i for i in tuple_corner if i != (nextx, nexty))

                new_state = ((nextx,nexty),new_tuple)
                cost = 1
                successors.append((new_state,action,cost))


        self._expanded += 1 # DO NOT CHANGE
        return successors
```

Now this problem is the food is not only one but 4 (each in the corner of the maze). Since the 4 search algorithm is completed above, now we have to determine the state of the search problem, the rules, the actions and the getSuccessors also. To do that keeping in mind that the initial code of the project already has the tuple of 4 corners so we will use that information to do this question. Using the position plus the tuple of 4 corners, whenever we expand a node that has that corner, remove it from the tuple. And when there is nothing left in the tuple, that is the goal of the problem.

Table 6: Result for Quetion 5

Layout	Total Cost	Nodes Expanded	Score
tinyCorners	28	252	512
mediumCorners	106	1966	434
bigCorners	162	7949	378

4.2 Question 6 (3 points): Corners Problem: Heuristic

```python
def helper(pos,tuple_corner):
    result = 0
    tuple_corner = set(tuple_corner)


    while(len(list(tuple_corner)) != 0):
        min_distance_manhanttan = float('inf')
        corner_remove = None

        for i in tuple_corner:
            check_temp = abs(pos[0]-i[0]) + abs(pos[1]-i[1])

            if check_temp < min_distance_manhanttan:
                min_distance_manhanttan = check_temp
                corner_remove = i

        result += min_distance_manhanttan
        tuple_corner.remove(corner_remove)
        pos = corner_remove

    return result


def cornersHeuristic(state: Any, problem: CornersProblem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)
```

```

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible.
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    "*** YOUR CODE HERE ***"
    result = 0
    pos,tuple_corner = state

    return helper(pos,tuple_corner)

    #return 0 # Default to trivial solution
```

To solve this question, we have to design a function that will estimate the cost from the position to the goal in the most appropriate way. In this question, because the goal isn't simply a single food like the first 4 questions anymore. It now contains 4 foods in the 4 corners. So we decided to use the distance from pos to the closest corner, and then from that corner to the other corner, and so on until there is no corner left.

4.3 Question 7 (3 points): Corners Problem: Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. The problem now will not only be 4 food in each corner, it now contains a random number of food located randomly in the maze. And we will use the command autograder.py -q q7 to evaluate.

```
def compute_mst(points):
    """
    Using Prim Algorithm
    """
    if len(points) <= 1:
        return 0

    # Priority queue for Prim's algorithm
    pq = PriorityQueue()
    visited = set()
    mst_cost = 0

    # Start with the first point
    start = points[0]
    visited.add(start)
```

```

    # Push initial edges to the priority queue
    for point in points[1:]:
        pq.push((start, point), manhattanDistance(start, point))

    # Process the MST
    while not pq.isEmpty():
        start, end = pq.pop()  # Pop the lowest-cost edge
        #print(cost)
        cost = manhattanDistance(start, end)
        if end not in visited:
            visited.add(end)
            mst_cost += cost

            # Add new edges from the newly visited point
            for other in points:
                if other not in visited:
                    pq.push((end, other), manhattanDistance(end, other))

    return mst_cost

    pacmanPosition, foodGrid = state
    foodList = foodGrid.asList()

    if not foodList:
        return 0  # No food left, heuristic is 0 at the goal state.

    # Compute the MST cost for the food points only
    mst_cost = compute_mst(foodList)

    # Add the distance from Pacman to the closest food point
    min_dist_to_food = min(manhattanDistance(pacmanPosition, food) for food
     in foodList)

    return mst_cost + min_dist_to_food
```

Result:

- Expanded node = 7209 nodes
- Time = 1.86 s

4.4 Question 8 (3 points): Suboptimal Search

Now the question will fill the maze with food. And our job is to make it play like a human(which is to make it come to the nearest food).

```
class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"
    def registerInitialState(self, state):
```

```python
        self.actions = []
        currentState = state
        while(currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState) # The missing piece
            self.actions += nextPathSegment
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception('findPathToClosestDot returned an illegal move: %s!\n%s' % t)
                currentState = currentState.generateSuccessor(0, action)
        self.actionIndex = 0
        print('Path found with cost %d.' % len(self.actions))

    def findPathToClosestDot(self, gameState: pacman.GameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        "*** YOUR CODE HERE ***"
        from search import depthFirstSearch
        from search import breadthFirstSearch
        from search import uniformCostSearch
        from search import aStarSearch

        #return depthFirstSearch(problem)
        #return breadthFirstSearch(problem)
        #return uniformCostSearch(problem)
        return aStarSearch(problem)
        util.raiseNotDefined()

class AnyFoodSearchProblem(PositionSearchProblem):
    """
    A search problem for finding a path to any food.

    This search problem is just like the PositionSearchProblem, but has a
    different goal test, which you need to fill in below.  The state space and
    successor function do not need to be changed.

```

```
    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.

    You can use this search problem to help you fill in the findPathToClosestDot
    method.
    """

    def __init__(self, gameState):
        "Stores information from the gameState.  You don't need to change this."
        # Store the food for later reference
        self.food = gameState.getFood()

        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT CHANGE

    def isGoalState(self, state: Tuple[int, int]):
        """
        The state is Pacman's position. Fill this in with a goal test that will
        complete the problem definition.
        """
        x,y = state

        "*** YOUR CODE HERE ***"

        #print(self.food)

        return self.food[x][y]
        util.raiseNotDefined()
```

It just simply uses which function that has been implemented from the first 4 questions. First, look at the result for each search algorithm

Table 7: Result for Quetion 8

Algorithm	Layout	Total Cost	Score
DFS	bigSearch	5324	-2614
BFS	bigSearch	350	2360
UCS	bigSearch	350	2360
Astar	bigSearch	350	2360

We can see that 3 algorithms BFS,UCS, Astar will all have results much better than DFS.

DFS cannot pass this question since is use too much cost and has a negative score.
Because of the DFS algorithm whenever pacman eats food,It will randomly generate the child node. Example there is a food in the next left side of the pacman, but it looks for the right hand side first. So it will not be correct in this question (which wants us to find the closest food).
The BFS algorithm will handle this question correctly since it begin to search all the node closest to it before going further.
UCS and Astar are actually the same because we call the Astar function without pass the heuristic function, so Astar is also the UCS. It had the result like the BFS but the way pacman goes to find food is not similar to BFS. We can understand this because UCS look for the shortest path first.

References

[1] *Project 1: Search* , 2024