University of Alexandria
Faculty of Engineering
Computer and Systems Engineering Department

Artificial Intelligence

Assignment 2: Pacman

Names:

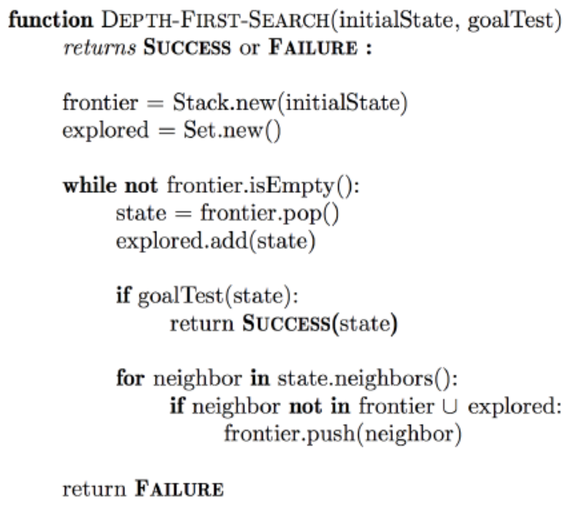
- Hazem Samir Sayed (#22)
- Amr Gamal Mohamed (#48)
- Mohamed Adel Abd El Fatah (#62)

Question 1 (Depth First Search):

First we initialize 3 data structures, frontier stack to hold the successors of explored states, explored set to hold the explored states and actions dictionary to hold the path used to reach each explored state.

Then we add the start state of the problem to the frontier stack and initialize the path to this state with empty list. The state is the coordinates of a point in the game board.

After that we loop over the frontier stack, pop its top state and add it to explored set. If the popped state is a goal state, then the path to reach this state is returned from the actions dictionary. Else we get the successors of this state and for each successor if it's not in the explored set, then it's added to the frontier stack and the path to reach this successor state is added to the actions dictionary and it's formed from the path to reach its parent and the path from its parent to it.



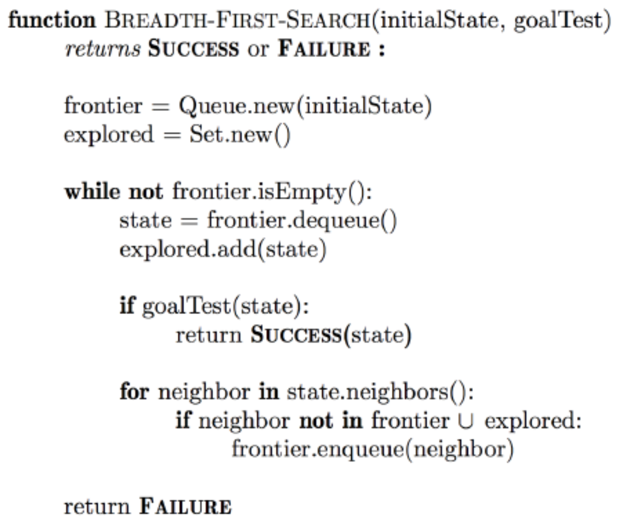
```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

Question 2 (Breadth First Search):

First we initialize 3 data structures, frontier queue to hold the successors of explored states, explored set to hold the explored states and actions dictionary to hold the path used to reach each explored state.

Then we add the start state of the problem to the frontier queue and initialize the path to this state with empty list. The state is the coordinates of a point in the game board.

After that we loop over the frontier queue and pop its first state. If the popped state is a goal state, then the path to reach this state is returned from the actions dictionary. Else we get the successors of this state and for each successor if it's not in the explored set, then it's added to the frontier queue and the explored set and the path to reach this successor state is added to the actions dictionary and it's formed from the path to reach its parent and the path from its parent to it.

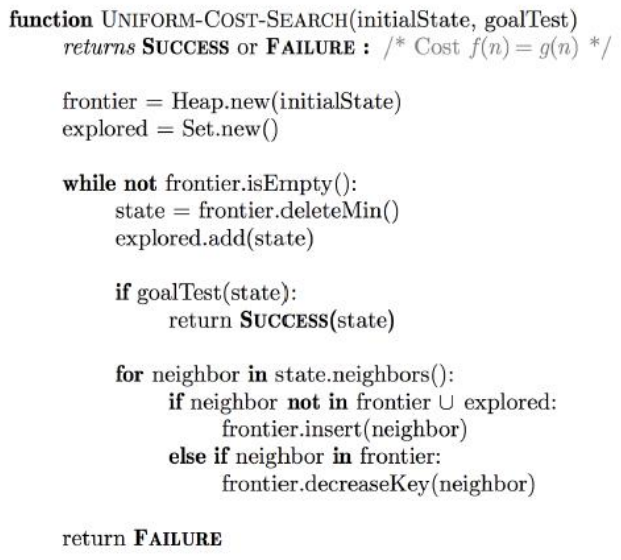


```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

Question 3 (Uniform Cost Search):

First we initialize 3 data structures, frontier heap (priority queue) to hold the successors of explored states, explored set to hold the explored states and actions dictionary to hold the path and the cost used to reach each explored state.

Then we add the start state of the problem to the frontier heap with priority zero and initialize the path to this state with empty list and zero cost. The state is the coordinates of a point in the game board and the priority is the cost to reach this state.

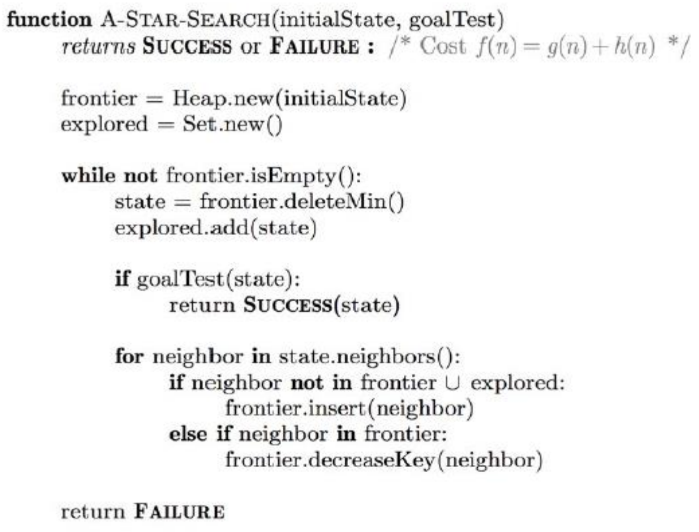


```
function UNIFORM-COST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost $f(n) = g(n)$ */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

After that we loop over the frontier heap and pop the state with the least priority. If the popped state is a goal state, then the path to reach this state is returned from the actions dictionary. Else we get the successors of this state and for each successor if it's not in the explored set and not in the frontier heap, then it's added to the explored set. The path to reach that successor state and its cost are calculated and the successor state is added to the heap with priority equal to that cost and the path to reach that state is added to the actions dictionary with the cost of this path. Else if the successor state is in the frontier heap and the cost to reach that state from this path is smaller than the stored one, then we update the priority of this state in the frontier heap and the path to reach this state along with its cost in the actions dictionary.

Question 4 (A*):

The A* Algorithm is almost the same as the Uniform cost search Algorithm implemented in Question 3. The only difference is that we store as Priority with each state in the heap the cost of the path to reach that node plus the value returned from the heuristic function for this state.



```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost $f(n) = g(n) + h(n)$ */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Question 5 (Finding All The Corners):

In the previous Problem, the goal is just a single point but in The Corners Problems the goal is 4 corner points which we need to pass by all of them to reach the goal. So we need to change the state used.

In the previous single goal point, the state was a tuple of 3 attributes which are the point coordinates, the path moved to reach that point and the cost of the path. Now we need to change the first attribute to be composed of the point coordinates along with another tuple contains the corner (goal) points passed by the agent. The three implemented functions in this Problem are getStartState, isGoalState and getSuccessors.

1- getStartState(): returns a tuple contains the starting position of the agent along with an empty tuple.
2- isGoalState(state): loops over all the corner points in the board and if all of them are in the points passed by the agent in the given state, then it returns True, otherwise it returns False.
3- getSuccessors(state): loops over the four possible directions for the agent to move to (four neighbors) and if the neighbor is not a wall, then we check if it is a corner point which is not already passed by the agent, then it's added to the corner points passed by the agent and finally the successor is added to the list of the successors of the given state.

Question 6 (Corners Problem: Heuristic):

In order to return a heuristic which is admissible and consistence, we decided to use the Manhattan distance needed to be moved by the agent in order to pass by the corner points not passed by the agent.

First we get the remaining corner points that are needed to be passed by the agent in order to reach the goal by removing the set of states passed by the agent from the set of corner points. Then we loop over these corner points and find the smallest Manhattan distance between the current position of the agent and the corner points. After that we add that distance to the heuristic, update the current position of the agent to that found corner point and remove the reached corner from the set of remaining corners. After all the corner points are visited, the found heuristic (total Manhattan distance) is returned.

Question 7 (Eating All The Dots):

Many Heuristics were tried in order to reach the best one:

- The simplest one is the number of food dots in the grid. It's the simplest both admissible and consistent heuristic but it expands 12517 nodes ($2/4$ grade).
- Then we used the same heuristic as in *Corners Problem* but to generalize it to build a path of all the food dots not only the corners. It failed the Admissibility test as it greedily builds the full path so at some cases it overestimates the cost.
- Then we tried using the most distant food dot from the current agent position, it produces better estimate and expands 9551 nodes ($3/4$ grade).
- Finally we decided to find the two most distant food dots (independent from the current position). The heuristic cost then will be the Manhattan distance of the minimum path from the current position of the agent and passing through these two food dots. It expands only 7459 nodes ($4/4$ grade).

Question 8 (Suboptimal Search):

Our problem is to get the path to the closest dot. In order to find the closest dot to the agent, the most suitable type of search to be used is the Breadth First Search because it returns the shallowest solution. AnyFoodProblem is used with its goal defined to be when a dot food is reached.