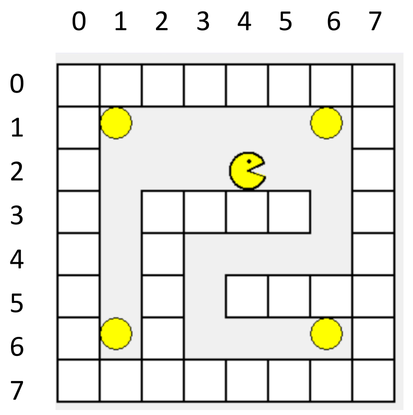
AI Assignment

In this assignment, we are going to experiment on the four corners problem. Pacman lives in a grid maze with dots on each corner as shown below:



The problem is to find a plan with the minimum number of steps to collect all four dots.

A state is defined as a tuple of Pacman position and remaining food positions in the maze. For example, the figure above has:

- Pacman Position=(4, 2),
- Remaining food positions = [(1, 1), (6, 1), (1, 6), (6, 6)]

So, the state is: ((4, 2), [(1, 1), (6, 1), (1, 6), (6, 6)]).

The goal state is when the remaining food is empty ([]).

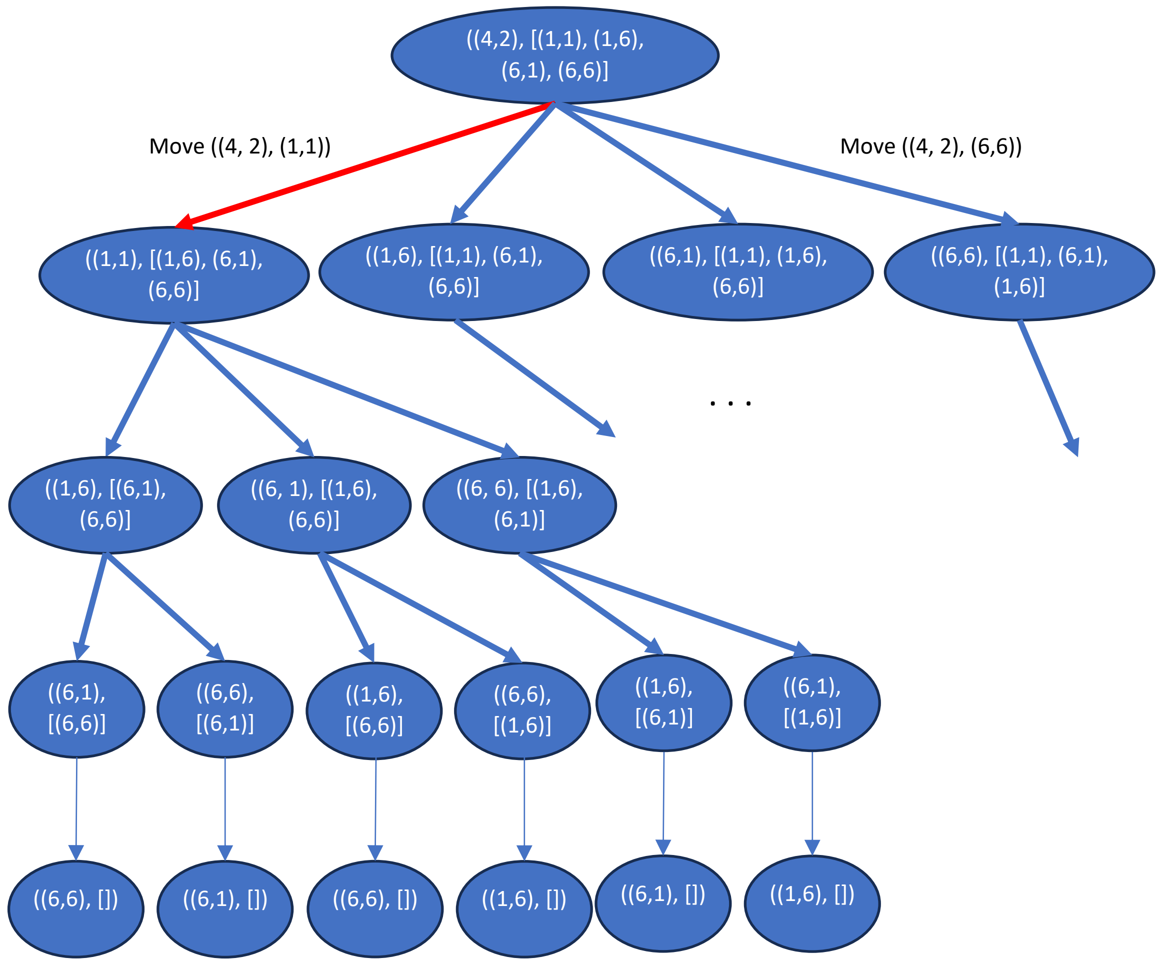
Think of the problem as the following states only:

Start state: ((4, 2), [(1, 1), (6, 1), (1, 6), (6, 6)])

If Pacman at location (1, 1) to eat the food at (1, 1), then the state could be any of the following:

- ((1, 1), [(6, 1), (1, 6), (6, 6)])
- ((1, 1), [(1, 6), (6, 6)])
- ((1, 1), [(6, 1), (6, 6)])
- ((1, 1), [(6, 1), (1, 6)])

The graph is as follows:



The above graph says that, the agent ‘Pacman’ is at (4, 2), if it moves to state (1, 1), then the agent is on state ((1,1), [(1,6),(6,1),(6,6)]), which is the state on the top left (red edge).

The cost of moving from state (4,2) to state (1, 1) is the number of steps needed to move from state (4, 2) to state (1, 1), this can be found if we apply Breadth-first search starting from state (4, 2) and the target state is (1, 1). In the fourCornersProblem, there is a function called BFS that takes input the start node and target node and return the minimum number of steps to move from start node to target node. The header function is:

```
def BFS(self, start_pos, target_pos):
```

This function returns the number of steps (actual cost) and the path from the start position to target position. You can call this function by:

cost, plan = BFS (start_pos, target_pos)

You are provided with the following programs:

fourCornerProblem.py	The file describes the problem. You have the initial (or start) state, the goal state, the transition, … etc.
fourConrners.py	The program describes the main program and the search function.

Let us look at the FourCornerProblem.py program, we have the class problem that defines the methods:

- **startState**: the start configuration of the problem,
- **isGoal**: checks that a state or node is a goal state, in this problem, the goal is when the remaining food list is empty.
- **transition**: returns the next states given a node or a state irrespective of the cost.
- **BFS**: finds the minimum number of steps to move from a start position to a target position.
- **nextState**: returns next state given a node or state with their cost. The cost is computed from the length of the plan returned by the BFS method.
- **h**: heuristic method that computes the heuristic of a node or a state. The heuristic h(current_state) is the cost to move from the current state to the goal state. The goal state is when Pacman eats all the foods in the maze. The algorithm to compute that is as follows:
    a) You have Pacman position and a list of the dots' positions. You can those from the current state = (pacman_position, foods_positions_list)
    b) compute the number of steps from Pacman position to every food and pick the smallest one. Call the food with smallest number of steps to reach from Pacman position f1.
    c) Then compute the number of steps from the food (f1) found in step (b) to every other food and pick the smallest. Call the new food found f2.

d) Then compute the number of steps from the food (f2) found in step (c) to every other food and pick the smallest.
e) Repeat that c and d until there is no more food left in the food list.

Here is a formal algorithm:

a) Define a list mst = {} and empty dictionary.
b) Set all mst for all foods position to False (mst[f] = False for all foods)
c) Define an accumulator sumSteps = 0
d) Loop until mst[f] all becomes true, meaning all foods are computed.
    a. Compute the minimum number of steps (s) needed from Pacman position to every food, ignore food with its mst is true.
    b. Set mst[f] = True, where f is the food with the minimum number of steps needed for Pacman to reach.
    c. Set Pacman position to f in step b.
    d. Add to sumSteps the s steps computed in a
    e. Go back to step a
e) After completing the loop return sumSteps as it is the minimum number of steps to reach the goal state from the current state. That is the heuristic.
f) Note: to compute the minimum number of steps between two positions (x1, y1) and (x2, y2), use the BFS method in the FourCornerProblem.py, it will compute the minimum distance.
g) Note: To speed up your program, create a lookup table for the number of steps needed between any two positions of Pacman and the foods and between any two foods positions. You can compute all number of steps between Pacman and any of the foods and between any two foods in advance in that lookup table before the search starts.

Answer the following questions:

a) Complete the code for heuristic method h, write down below the code for h:

```
def h(self, state):
        # Note: state is of the form (pacman_position, remaining_food_list)
        # e.g., state = ((4, 2), [(1, 1), (1, 6), (6, 1), (6, 6)])
```

Code is at bottom of doc

Let us look at the FourCorners.py program, that consists of the following functions:

- bfs: that do the search using breadth first search.
- ucs: that do the search using uniform cost search
- AStar: that do the search using A* search
- The main program that:
    1) Create an instance of the FourCornerProblem
    2) Call the AStar method
    3) Create an instance of the pacmanGraphics
    4) Setup the screen with a graphical maze.
    5) Execute the plan returned.

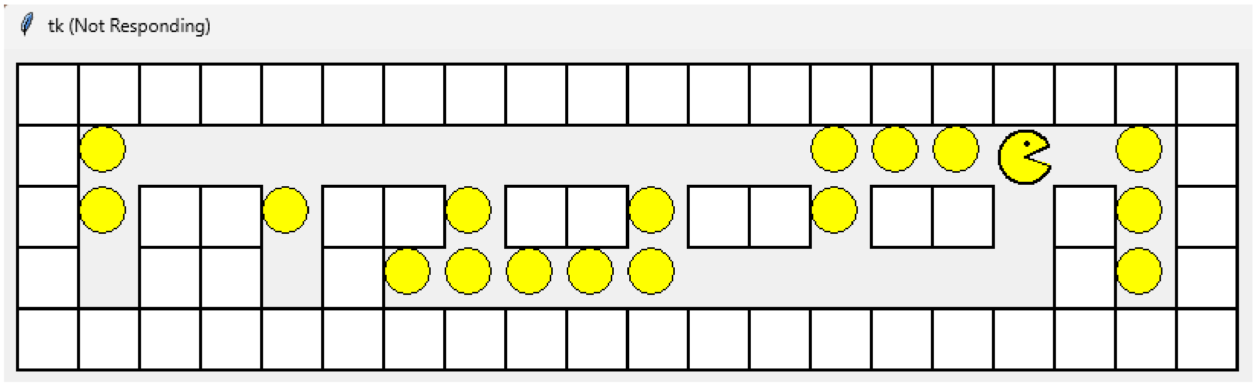
a) Complete the code for the main where it says complete the code here.
b) Complete the part in AStar where it see # Complete your code here.
c) Run the program on the file name 'tinyCorners.txt' and record the following:
    - BFS:
        a) Number of nodes explored: 251
        b) Time to execute algorithm: 3.55 ms
        c) Plan length: 28
    - UCS:
        a) Number of nodes explored: 21
        b) Time to execute algorithm: 4.34 ms
        c) Plan length: 28
    - A*:
        a) Number of nodes explored: 4
        b) Time to execute algorithm: 4.36 ms
        c) Plan length: 28
d) Change the file name to 'mediumCorners.txt', and record the following:
    - BFS:
        a) Number of nodes explored: 1972
        b) Time to execute algorithm: 44.2 ms
        c) Plan length: 106
    - UCS:
        a. Number of nodes explored: 23

b. Time to execute algorithm: 52.08 ms
c. Plan length: 106

- A*:
  a. Number of nodes explored: 6
  b. Time to execute algorithm: 52.09 ms
  c. Plan length: 106

Part 2:

The second part of the assignment, is using the A* to collect the dots in a maze Given a maze as follows:



Try the running the same program on the maze: 'tinySearch.txt'. Record the following:

- BFS:
  a. Number of nodes explored: 5312
  b. Time to execute algorithm: 252 ms
  c. Plan length: 27
- UCS:
  d. Number of nodes explored: 3726
  e. Time to execute algorithm: 56 ms
  f. Plan length: 27
- A*:
  g. Number of nodes explored: 12
  h. Time to execute algorithm: 17 ms

i. Plan length: 27

Try running the same program on the maze: ‘smallSearch.txt”. Record the following:

- BFS:
  a) Number of nodes explored: 69189
  b) Time to execute algorithm: 58248 ms
  c) Plan length: 34
- UCS:
  a) Number of nodes explored: 417717
  b) Time to execute algorithm: 25307 ms
  c) Plan length: 34
- A*:
  a) Number of nodes explored: 26
  b) Time to execute algorithm: 52 ms
  c) Plan length: 34

Create a github repository then upload the completed programs FoutCornerProblem.py, FourCorners.py and this document “FourCornersAssignment.docx”, then post the link to the repository into Canvas assignment.

```
if not remaining_dots:
  return 0
mst = {dot: False for dot in remaining_dots}
total_steps = 0
current_position = current_pos
while any(not visited for visited in mist.values()):
  min_steps = float('inf')
  nearest_dot = None
  for dot in remaining_dots:
    if not mst[dot]:
      if current_position == self.pacman:
        plan = self.dist[(current_position, dot)]
      else:
        plan = self.dist[(current_position, dot)]
      steps = len(plan)
      if steps < min_steps:
        min_steps = steps
        nearest_dot = dot
  if nearest_dot:
    mst[nearest_dot] = True
    total_steps += min_steps
    current_position = nearest_dot
return total_steps
```