Artificial intelligence - Project 1
- Search problems -

Vesa Bianca

4/11/2020

1 Uninformed search

1.1 Question 1 - Depth-first search

In this section the solution for the following problem will be presented:

*"In search.py, implement **Depth-First search(DFS) algorithm** in function depthFirstSearch. Don't forget that DFS graph search is graph-search with the frontier as a LIFO queue(Stack).".*

1.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  class Node:
2
3      def __init__(self, state, action, parent, path_cost=0):
4          self.state = state
5          self.action = action
6          self.parent = parent
7          self.path_cost = path_cost
8
9
10 def depthFirstSearch(problem):
11     adj = util.Stack()
12     explored = []
13     solution = []
14
15     start_node = Node(problem.getStartState(), None, None, 0)
16
17     adj.push(start_node)
18
19     while not adj.isEmpty():
20         current_node = adj.pop()
21
22         if current_node.state in explored: continue
23
24         if problem.isGoalState(current_node.state):
25             path_node = current_node
26
27             while path_node.state != start_node.state:
28                 solution.append(path_node.action)
29                 path_node = path_node.parent
30
31             solution.reverse()
32             return solution
33
34         explored.append(current_node.state)
35         successors = problem.getSuccessors(current_node.state)
36
```

```
37          for successor in successors:
38              if (not explored.__contains__(successor[0])) and (not adj.list.__contains__(successor)):
39                  new_node = Node(successor[0], successor[1], current_node, current_node.path_cost + succe
40                  adj.push(new_node)
```

Explanation:

- DFS function returns as a solution a sequence of actions that have to be taken in order to reach the goal state.
- For this implementation we are going to use the data structure Stack, from file util.py from this project, and define it in line 11. In lines 12 and 13 the explored set and solution are also initialized empty, for the initial state of the problem.
- The frontier is initialized in line 17 with the first node, and then we are going to loop over it, while it is not empty. If the frontier is empty, the empty initialized solution will be returned.
- In line 20, a leaf node is chosen, and extracted from the frontier. Because we want the graph-search version of the DFS problem, the current extracted node is searched for in the explored set, in line 22. If it has been explored, it will be skipped.
- The node's state is tested against the goal state in line 24. If the node contains a goal state, the method returns the corresponding solution, tracing back the search path (lines 27 to 29), with the help of the parameter 'parent' in Node structure. The solution is reversed before the return statement, because the obtained path has to be taken from start to goal and not viceversa.
- If the current node does not contain a goal state, it is marked as explored in line 34 and then expanded. Its successors are being provided in line 35. The resulting nodes are added to the frontier in lines 37 to 41.

Commands:

- -l tinyMaze -p SearchAgent -l mediumMaze -p SearchAgent -l bigMaze -z .5 -p SearchAgent

1.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1: When searching a state space for a path to a goal, depth-first search algorithm may not produce the most optimal solution. That is because the number of steps taken to reach the goal, and the cost spent in reaching it might be too high for certain state spaces (ex: if the search tree is not finite, it might never find a solution).
Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: 3/3

1.1.3 Personal observations and notes

There are not any additional observations.

1.2 Question 2 - Breadth-first search

In this section the solution for the following problem will be presented:

*"In **search.py**, implement the **Breadth-First search** algorithm in function breadthFirstSearch."*.

1.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  class Node:
2
3      def __init__(self, state, action, parent, path_cost=0):
4          self.state = state
5          self.action = action
6          self.parent = parent
7          self.path_cost = path_cost
8
9
10 def breadthFirstSearch(problem):
11     adj = util.Queue()
12     explored = []
13     solution = []
14
15     start_node = Node(problem.getStartState(), None, None, 0)
16
17     adj.push(start_node)
18
19     while not adj.isEmpty():
20         current_node = adj.pop()
21
22         if current_node.state in explored: continue
23
24         if problem.isGoalState(current_node.state):
25             path_node = current_node
26
27             while path_node.state != start_node.state:
28                 solution.append(path_node.action)
29                 path_node = path_node.parent
30
31             solution.reverse()
32             return solution
33
34         explored.append(current_node.state)
35         successors = problem.getSuccessors(current_node.state)
36
37         for successor in successors:
38             if (not explored.__contains__(successor[0])) and (not adj.list.__contains__(successor)):
39                 new_node = Node(successor[0], successor[1], current_node, current_node.path_cost + succe
40                 adj.push(new_node)
```

Explanation:

- BFS function returns as a solution a sequence of actions that have to be taken in order to reach the goal state. For this implementation we are going to use the data structure Queue, from file util.py

from this project, and define it in line 11. In lines 12 and 13 the explored set and solution are also initialized empty, for the initial state of the problem.

- The frontier is initialized in line 17 with the first node, and then we are going to loop over it, while it is not empty. If the frontier is empty, the empty initialized solution will be returned.
- In line 20, a leaf node is chosen, and extracted from the frontier. Because we want the graph-search version of the DFS problem, the current extracted node is searched for in the explored set, in line 22. If it has been explored, it will be skipped.
- The node's state is tested against the goal state in line 24. If the node contains a goal state, the method returns the corresponding solution, tracing back the search path (lines 27 to 29), with the help of the parameter 'parent' in Node structure. The solution is reversed before the return statement, because the obtained path has to be taken from start to goal and not viceversa.
- If the current node does not contain a goal state, it is marked as explored in line 34 and then expanded. Its successors are being provided in line 35. The resulting nodes are added to the frontier in lines 37 to 41.

Commands:

- -l mediumMaze -p SearchAgent -a fn=bfs -l bigMaze -p SearchAgent -a fn=bfs -z .5

1.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1: The solution found by breadth-first search is optimal, as long as all the costs in the state space are equal. Otherwise, it will return the shallowest path to a goal state in terms of steps taken, which might not be the most efficient one.
Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: 3/3

1.2.3 Personal observations and notes

There are not any additional observations.

1.3 Question 3 - Uniform-cost search

In this section the solution for the following problem will be presented:

*"In search.py, implement **Uniform-cost graph search** algorithm in uniformCostSearchfunction"*

1.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  class Node:
2
3      def __init__(self, state, action, parent, path_cost=0):
4          self.state = state
5          self.action = action
6          self.parent = parent
7          self.path_cost = path_cost
8
9
10 def uniformCostSearch(problem):
11     adj = util.PriorityQueue()
12     explored = []
13     solution = []
14
15     start_node = Node(problem.getStartState(), None, None, 0)
16
17     adj.push(start_node, start_node.path_cost)
18
19     while not adj.isEmpty():
20         current_node = adj.pop()
21
22         if current_node.state in explored: continue
23
24         if problem.isGoalState(current_node.state):
25             path_node = current_node
26
27             while path_node.state != start_node.state:
28                 solution.append(path_node.action)
29                 path_node = path_node.parent
30
31             solution.reverse()
32             return solution
33
34         explored.append(current_node.state)
35         successors = problem.getSuccessors(current_node.state)
36
37         for successor in successors:
38             if (not explored.__contains__(successor[0])) and (not adj.heap.__contains__(successor)):
39                 new_node = Node(successor[0], successor[1], current_node, current_node.path_cost + succe
40                 adj.push(new_node, new_node.path_cost)
```

Explanation:

- UCS function returns as a solution a sequence of actions that have to be taken in order to reach the goal state.
- For this implementation we are going to use the data structure PriorityQueue, from file util.py from this project, and define it in line 11. In lines 12 and 13 the explored set and solution are also initialized empty, for the initial state of the problem.
- The frontier is initialized in line 17 with the first node and its corresponding cost, and then we are going to loop over it, while it is not empty. If the frontier is empty, the empty initialized solution will be returned.
- In line 20, a leaf node is chosen, and extracted from the frontier. Because we want the graph-search version of the DFS problem, the current extracted node is searched for in the explored set, in line 22. If it has been explored, it will be skipped.

- The node's state is tested against the goal state in line 24. If the node contains a goal state, the method returns the corresponding solution, tracing back the search path (lines 27 to 29), with the help of the parameter 'parent' in Node structure. The solution is reversed before the return statement, because the obtained path has to be taken from start to goal and not viceversa.
- If the current node does not contain a goal state, it is marked as explored in line 34 and then expanded. Its successors are being provided in line 35. The resulting nodes are added to the frontier in lines 37 to 41, updating their cost with the cost of the path taken so far, which can easily be calculated with the parameter 'path cost', that is stored in the Node structure.

Commands:

- -l mediumMaze -p SearchAgent -a fn=ucs -l mediumDottedMaze -p StayEastSearchAgent -l mediumScaryMaze -p StayWestSearchAgent

1.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Compare the results to the ones obtained with DFS. Are the solutions different? Is the number of extended (explored) states smaller? Explain your answer.

A1: The score returned by DFS is smaller than the one generated by UCS, because in this case, every step taken by Pacman will decrease its current score. That means we would prefer the path that gives us the highest score, which is the shortest one. The number of states explored by UCS is also higher, showing that it searches most of the state space in an effort to find the most optimal solution.

Q2: Consider that some positions are more desirable than others. This can be modeled by a cost function which sets different values for the actions of stepping into positions. Identify in **searchAgents.py** the description of agents StayEastSearchAgent and StayWestSearchAgent and analyze the cost function. Why the cost .5 ** x for stepping into (x,y) is associated to StayWestAgent.

A2: StayEastAgent and StayWestAgent are implemented like so, because the cost has to correspond to the x value of the position. Stepping into a position that is to the east from the current one will increase x's value, so the function corresponding to the east agent is 2^x, which will also increase. Moving west will have the opposite effect. X's value has to decrease, so we are going to apply a function with decreasing values, that is 0.5^x.

Q3: Run autograder *python autograder.py* and write the points for Question 3.

A3: 3/3

1.3.3 Personal observations and notes

There are not any additional observations.

1.4 References

2 Informed search

2.1 Question 4 - A* search algorithm

In this section the solution for the following problem will be presented:

*"Go to aStarSearch in search.py and implement **A* search algorithm**. A* is graphs search with the frontier as a priorityQueue, where the priority is given bythe function g=f+h".*

2.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:
Explanation:

- A* function returns as a solution a sequence of actions that have to be taken in order to reach the goal state. For this implementation we are going to use the data structure PriorityQueue, from file util.py from this project, and define it in line 11. In lines 12 and 13 the explored set and solution are also initialized empty, for the initial state of the problem.
- The frontier is initialized in line 17 with the first node and its corresponding cost. The heuristic is also taken into consideration, to help us estimate how close we are to the goal state. Then we are going to loop over the frontier, while it is not empty. If the frontier is empty, the empty initialized solution will be returned.
- In line 20, a leaf node is chosen, and extracted from the frontier. Because we want the graph-search version of the DFS problem, the current extracted node is searched for in the explored set, in line 22. If it has been explored, it will be skipped.
- The node's state is tested against the goal state in line 24. If the node contains a goal state, the method returns the corresponding solution, tracing back the search path (lines 27 to 29), with the help of the parameter 'parent' in Node structure. The solution is reversed before the return statement, because the obtained path has to be taken from start to goal and not viceversa.
- If the current node does not contain a goal state, it is marked as explored in line 34 and then expanded. Its successors are being provided in line 35. The resulting nodes are added to the frontier in lines 37 to 41, updating their cost with the cost of the path taken so far. This can easily be calculated with the parameter 'path cost', that is stored in the Node structure, also adding the heuristic of the current state.

Commands:

- -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

2.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Does A* and UCS find the same solution or they are different?
A1: A* can achieve a better solution, because it uses heuristics to guide its search. Combining them with choosing a cost effective path, it might generate an optimized one, compared to the UCS path.
Q2: Does A* finds the solution with fewer expanded nodes than UCS?

A2: Using heuristics, A* is trying to find the next state that is closer to the goal, therefore minimizing the number of nodes expanded.

Q3: Does A* finds the solution with fewer expanded nodes than UCS?

A3: Using heuristics, A* is trying to find the next state that is closer to the goal, therefore minimizing the number of nodes expanded.

Q4: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).

A4: 3/3

2.1.3 Personal observations and notes

There are not any additional observations.

2.2 Question 5 - Find all corners - problem implementation

In this section the solution for the following problem will be presented:

"Pacman needs to find the shortest path to visit all the corners,regardless there is food dot there or not. Go to ***CornersProblem*** *in searchAgents.py and propose a representation of the state of this search problem. It might help to look at the existing implementation for PositionSearchProblem. The representation should include only the information necessary to reach the goal. Read carefully the comments inside the class CornersProblem.".*

2.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
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):
        """
        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))
        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)
        """
        return self.startingPosition, []


    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """

        for corner in self.corners:
            if corner not in state[1]:
                return False
        return True

    def getSuccessors(self, state):
        """
        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
        """

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            x,y = state[0]
            corners_reached = state[1]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]

            if not hitsWall:
                if ((nextx, nexty) in self.corners) and ((nextx, nexty) not in corners_reached):
                    corners_reached = corners_reached + [(nextx, nexty)]
                    nextState = ((nextx, nexty), corners_reached)
                    successors.append((nextState, action, 1))
                else:
                    nextState = ((nextx, nexty), corners_reached)
                    successors.append((nextState, action, 1))


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

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
```

```
76          """
77          if actions == None: return 999999
78          x,y= self.startingPosition
79          for action in actions:
80              dx, dy = Actions.directionToVector(action)
81              x, y = int(x + dx), int(y + dy)
82              if self.walls[x][y]: return 999999
83          return len(actions)
```

Explanation:

- This search problem finds a path that visits all four corners of the Pacman game layout.
- For solving it, getStartState, isGoalState and getSuccessors methods needed to be implemented. Starting with getStartState, in line 27, it returns the starting state of the game. This state takes the form of a tuple, with the first element being the starting position of Pacman, and the second element being a list (initially empty), that is going to store the corners that have already been visited, so that we can keep track correctly of the explored states.
- The goal state of the problem is when all 4 corners have been visited, so what the method isGoalState does is checking the list that was previously mentioned (lines 35 to 37), that can be found in the state tuple.
- For a given state, the successor states are needed. Therefore, the method getSuccessors returns a list that contains them. This list is build in the following way. For each action that can be taken (line 52), variables x, y and 'corners reached' store the coordinates of the current state and the list of already visited corners (lines 53 and 54). The successor's coordinates are being calculated in lines 55 and 56. If the action is valid, and it does not lead us to the walls of the layout (lines 57 and 59), the successor is searched for in the corners list. If it is indeed a corner and it hasn't been visited yet, the successor's coordinates will be added to the already visited corners list (lines 61) and the successor's entire state will be appended to the list of successors (line 62 and 63). Otherwise, the successor will be added to the list of successor, without updating its corners list (lines 65 and 66).

Commands:

- -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

2.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: For mediumCorners, BFS expands a big number - around 2000 search nodes. It's time to see that A* with an admissible heuristic is able to reduce this number. Please provide your results on this matter. (Number of searched nodes).

A1: Search nodes expanded for mediumCorners: BFS - 2448, A* - 901.

2.2.3 Personal observations and notes

There are not any additional observations.

2.3 Question 6 - Find all corners - Heuristic definition

In this section the solution for the following problem will be presented:

"Implement a consistent heuristic for CornersProblem. Go to the function ***cornersHeuristic*** *in searchAgent.py.".*

2.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  def cornersHeuristic(state, problem):
2      corners = problem.corners # These are the corner coordinates
3      walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
4
5
6      unvisited = []
7      visited = state[1]
8      current_position = state[0]
9      heuristic = 0
10
11     for corner in corners:
12         if not corner in visited:
13             unvisited.append(corner)
14
15     while unvisited:
16         distances = []
17         for unvisited_corner in unvisited:
18             distances.append((util.manhattanDistance(current_position, unvisited_corner), unvisited_corn
19
20         min_dist,corner = min(distances)
21
22         heuristic += min_dist
23         current_position = corner
24         unvisited.remove(corner)
25
26     return heuristic # Default to trivial solution
```

Explanation:

- In order for the Corners problem to work properly, we need to implement an admissible and consistent heuristic.
- The heuristic that I found to be most suitable is the sum of manhattan distances between the unvisited corners, starting with the closest one.
- It starts with obtaining the visited corners from the corners already visited list, and also the position of Pacman, which are stored in state (lines 7 and 8).
- The unvisited corners list is build in lines 11 to 13, with the corners that are not in the visited list. Looping over the unvisited list, and starting with the current position as Pacman's coordinates, as long as there are unvisited corners (lie 15), we add the minimum manhattan distance to the next corner (lines 17 and 18) in the variable 'heuristic' (line 22). After that, we move to the next postion, that is the closest corner (line 23) and remove the corner that has just been visited from the unvisited list.
- Admissibility is obtained even when all the corners have already been visited, because the heuristic has been initialized with value 0.

Commands:

- -l mediumCorners -p AStarCornersAgent -z 0.5

2.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with on the mediumMaze layout. What is your number of expanded nodes?
A1: Search nodes expanded for mediumCorners: 901. Search nodes expanded for mediumMaze: 1304.

2.3.3 Personal observations and notes

There are not any additional observations.

2.4 Question 7 - Eat all food dots - Heuristic definition

In this section the solution for the following problem will be presented:

"Propose a heuristic for the problem of eating all the food-dots. The problem of eating all food-dots is already implemented in ***FoodSearchProblem*** *in searchAgents.py.".*

2.4.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  def foodHeuristic(state, problem):
2      position, foodGrid = state
3      food_list = foodGrid.asList()
4      heuristic = 0
5      current_position = position
6
7      while food_list:
8          distances = []
9
10         for food in food_list:
11             distances.append((mazeDistance(current_position, food, problem.startingGameState), food))
12
13         max_dist, food_pos = max(distances)
14
15         if max_dist > heuristic:
16             heuristic = max_dist
17
18         current_position = food_pos
19         food_list.remove(food_pos)
20
21     return heuristic
```

Explanation:

- In order for the FoodSearch problem to work properly, we need to implement an admissible and consistent heuristic.

- The heuristic that I found to be most suitable is the maximum maze distance between the foods that have to be eaten, starting with the closest one.
- It starts with obtaining the foods that have not been eaten, and also the position of the Pacman, which are stored in state (lines 2 and 3).
- For every food that has not been eaten, the maze distance between the current position and the next food is calculated (lines 10 and 11).
- If the distance is bigger than the current heuristic, the value of the heuristic is updated (lines 15 and 16), and then the Pacman is moved in the next state, line 18.
- Admissibility is obtained even when all the foods have already been eaten, because the heuristic has been initialized with value 0.

Commands:

- -l testSearch -p AStarFoodSearchAgent

2.4.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with autograder *python autograder.py*. Your score depends on the number of expanded states by A* with your heuristic. What is that number?
A1: Expanded nodes: 1130.

2.4.3 Personal observations and notes

There are not any additional observations.

2.5 References

```python
class Node:

    def __init__(self, state, action, parent, path_cost=0):
        self.state = state
        self.action = action
        self.parent = parent
        self.path_cost = path_cost


def aStarSearch(problem, heuristic=nullHeuristic):
    adj = util.PriorityQueue()
    explored = []
    solution = []

    start_node = Node(problem.getStartState(), None, None, 0)

    adj.push(start_node, start_node.path_cost + heuristic(start_node.state, problem))

    while not adj.isEmpty():
        current_node = adj.pop()

        if current_node.state in explored: continue

        if problem.isGoalState(current_node.state):
            path_node = current_node

            while path_node.state != start_node.state:
                solution.append(path_node.action)
                path_node = path_node.parent

            solution.reverse()
            return solution

        explored.append(current_node.state)
        successors = problem.getSuccessors(current_node.state)

        for successor in successors:
            if (not explored.__contains__(successor[0])) and (not adj.heap.__contains__(successor)):
                new_node = Node(successor[0], successor[1], current_node, current_node.path_cost + succe
                adj.update(new_node, new_node.path_cost + heuristic(new_node.state, problem))
```

Listing 1: Solution for the A* algorithm.

3 Adversarial search

3.1 Question 8 - Improve the ReflexAgent

In this section the solution for the following problem will be presented:

"Improve the ReflexAgent such that it selects a better action. Include in the score food locations and ghost locations. The layout testClassic should be solved more often.".

3.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class ReflexAgent(Agent):

    def getAction(self, gameState):
        # Collect legal moves and successor states
        legalMoves = gameState.getLegalActions()

        # Choose one of the best actions
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        "Add more of your code here if you want to"

        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState, action):
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

        "*** YOUR CODE HERE ***"

        minFoodDist = 100000
        minGhostDist = 100000

        for food in newFood.asList():
            minFoodDist = min(minFoodDist, manhattanDistance(newPos, food))

        for ghost in successorGameState.getGhostPositions():
            minGhostDist = min(minGhostDist, manhattanDistance(newPos, ghost))

        if len(newFood.asList()) == 0:
            minFoodDist = 1
```

```
37
38          stop_pen = 0
39          if action == Directions.STOP and minGhostDist > 5:
40              stop_pen = -100
41
42          return successorGameState.getScore() + minGhostDist / minFoodDist + stop_pen
```

Explanation:

- The evaluation function for the reflex agent is based on the next formula: game score + smallest distance to a ghost / smallest distance to the next food + stop penalty.
- Smallest distance to the next food is calculated in lines 29 and 30, computing the manhattan distances, and the smallest distance to a ghost is found in lines 32 and 33, computed in the same way.
- In order for Pacman to make the best decision, we want the distance to the closest ghost to be big, and the distance to the closest food to be small. In order to obtain a value that pleases both situations, we need to divide the two of them. I also added a stop penalty for the case in which the ghost is far enough for Pacman to continue eating the closest foods (lines 38 to 40).

Commands:

- -p ReflexAgent -l testClassic

3.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your agent on the openClassic layout. Given a number of 10 consecutive tests, how many types did your agent win? What is your average score (points)?
A1: The agent won 10 times, and the average score is 563.

3.1.3 Personal observations and notes

There are not any additional observations.

3.2 Question 9 - H-Minimax algorithm

In this section the solution for the following problem will be presented:

" Implement H-Minimax algorithm in MinimaxAgentclass from multiAgents.py. Since it can be more than one ghost, for each max layer there are one ormore min layers. ".

3.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):
            """
            Returns the minimax action from the current gameState using self.depth
            and self.evaluationFunction.

            Here are some method calls that might be useful when implementing minimax.

            gameState.getLegalActions(agentIndex):
            Returns a list of legal actions for an agent
            agentIndex=0 means Pacman, ghosts are >= 1

            gameState.generateSuccessor(agentIndex, action):
            Returns the successor game state after an agent takes an action

            gameState.getNumAgents():
            Returns the total number of agents in the game

            gameState.isWin():
            Returns whether or not the game state is a winning state

            gameState.isLose():
            Returns whether or not the game state is a losing state
            """

            value, action = self.minimaxDecision(gameState, 0, 0)
            return action


    def minimaxDecision(self, gameState, player, depth):

        legalActions = gameState.getLegalActions(player)

        #cutoff-test
        if len(legalActions) == 0 or depth == self.depth:
            return self.evaluationFunction(gameState), None

        if player == 0:
            return self.agentValue(gameState, player, depth)
        else:
            return self.ghostValue(gameState, player, depth)


    def agentValue(self, gameState, player, depth):
        legalActions = gameState.getLegalActions(player)

        max_values = []

        for legalAction in legalActions:
            successor = gameState.generateSuccessor(player, legalAction)

            max_values.append((self.minimaxDecision(successor, player + 1, depth)[0], legalAction))

```

```
55          return max(max_values)
56
57      def ghostValue(self, gameState, player, depth):
58          legalActions = gameState.getLegalActions(player)
59          last_ghost = gameState.getNumAgents()
60          min_values = []
61
62          for legalAction in legalActions:
63              successor = gameState.generateSuccessor(player, legalAction)
64              next_player = player + 1
65
66              if next_player == last_ghost:
67                  next_player = 0
68                  min_values.append((self.minimaxDecision(successor, next_player, depth + 1)[0], legalActi
69              else:
70                  min_values.append((self.minimaxDecision(successor, next_player, depth)[0], legalAction))
71
72          return min(min_values)
```

Explanation:

- Pacman needs to evaluate every action that is given as parameter in method getAction in an optimal way. To do so, the minimax algorithm is implemented in the following way.
- A pair of value and action is returned by minimaxDecision method, out of which, the action is taken by Pacman (lines 27, 28). Min agent (Pacman) is considered to be the player with index 0 and max agents (the ghosts) have index >= 1.
- The cutoff test for depth limit is implemented in line 36 and then, the corresponding player to the index given is called, in lines 39 to 42. If the cutoff test is not passed, that means we have to return the evaluation function of the current state. What a Pacman agent has to do is: for each legal action (line 50) , get the successor corresponding to it (line 51), and take the minimax decision for the next player, the ghost (line 53). The max player will want the maximum value out of all the values returned by the min agents (line 55).
- On the other hand, the ghosts will do the next steps: for each legal action (line 50) , get the successor corresponding to it (line 51), and take the minimax decision for the next player. If the next player is Pacman, the depth will increase, otherwise the depth will stay the same until the last ghost has made its decision (lines 64 to 70). The min player will want the minimum value out of all the values returned by the max agents (line 55).

Commands:

- -p MinimaxAgent -l minimaxClassic -a depth=4

3.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test Pacman on trappedClassic layout and try to explain its behaviour. Why Pacman rushes to the ghost?

A1: Pacman rushes to the ghost because Minimax algorithm considers playing against optimal agents. In this case, if the blue ghost agent is optimal, it will try to kill Pacman. Knowing the ghost will kill it under these conditions, it will try to minimize the decrease in its score, that appears for every state taken, so Pacman will end the game as soon as possible.

3.2.3 Personal observations and notes

There are not any additional observations.

3.3 Question 10 - Use $\alpha - \beta$ pruning in AlphaBetaAgent

In this section the solution for the following problem will be presented:

*" Use alpha-beta prunning in **AlphaBetaAgent** from multiagents.py for a more efficient exploration of minimax tree."*.

3.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
1  class AlphaBetaAgent(MultiAgentSearchAgent):
2      """
3      Your minimax agent with alpha-beta pruning (question 3)
4      """
5
6      def getAction(self, gameState):
7          """
8          Returns the minimax action using self.depth and self.evaluationFunction
9          """
10         alpha = -1000000
11         beta = 1000000
12         return self.alphaBetaPrunning(gameState, 0, 0, alpha, beta)[1]
13
14     def alphaBetaPrunning(self, gameState, player, depth, alpha, beta):
15         legalActions = gameState.getLegalActions(player)
16
17         if len(legalActions) == 0 or depth == self.depth:
18             return self.evaluationFunction(gameState), None
19
20         if player == 0:
21             return self.agentValue(gameState, player, depth, alpha, beta)
22         else:
23             return self.ghostValue(gameState, player, depth, alpha, beta)
24
25
26     def agentValue(self, game_state, player, depth, alpha, beta):
27
28         legalActions = game_state.getLegalActions(player)
29         max_value = -1000000
30         max_values = []
31         action = None
32
```

```
33          for legalAction in legalActions:
34              successor = game_state.generateSuccessor(player, legalAction)
35
36              max_values.append((self.alphaBetaPrunning(successor, player + 1, depth, alpha, beta)[0], leg
37
38              max_value, action = max(max_values)
39              alpha = max(alpha, max_value)
40
41              if max_value > beta:
42                  return max_value, action
43
44          return max_value, action
45
46
47      def ghostValue(self, gameState, player, depth, alpha, beta):
48          legalActions = gameState.getLegalActions(player)
49
50          last_ghost = gameState.getNumAgents()
51          min_value = 1000000
52          min_values = []
53          action = None
54
55          for legalAction in legalActions:
56              successor = gameState.generateSuccessor(player, legalAction)
57              next_player = player + 1
58
59              if next_player == last_ghost:
60                  next_player = 0
61                  min_values.append((self.alphaBetaPrunning(successor, next_player, depth + 1, alpha, beta
62              else:
63                  min_values.append((self.alphaBetaPrunning(successor, next_player, depth, alpha, beta)[0]
64
65              min_value, action = min(min_values)
66              beta = min(beta, min_value)
67
68              if min_value < alpha:
69                  return min_value, action
70
71          return min_value, action
```

Explanation:

- Alpha-beta prunning algorithm has the same steps as minimax algorithm, with the addition that we introduce two parameters that help us reduce the number of game states in the game tree.
- Alpha will represent the value of the best choice there is so far at any given state for a max agent, and beta the best choice for a min agent. Having these two numbers, the agents will test to see if the next actions are worth exploring. For max agent, in lines 39 to 42, the alpha factor(the best choice so far) is updated if it is the case, and then drops the evaluation of the next action if the current value is bigger than beta.
- Same thing happens for min agent, that updates his best choice accordingly, and drops the evaluation of the next action it it doesn't reach the factor alpha.

Commands:

- -p AlphaBetaAgent -a depth=3 -l smallClassic

3.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your implementation with autograder **python autograder.py** for Question 3. What are your results?
A1: 5/5

3.3.3 Personal observations and notes

There are not any additional observations.

3.4 References

4 Personal contribution

4.1 Question 11 - Define and solve your own problem.

In this section the solution for the following problem will be presented:

4.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:
Explanation:

-

Commands:

-

4.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

4.1.3 Personal observations and notes

4.2 References