UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Pacman

Artificial Intelligence

Authors: YOUSSEF ABANOUB
Group: 30216

Faculty of Automation
and Computer Science

2023-2024

Cuprins

1 Introduction

1.1 Context

Pac-Man is one of the most popular games in the world. The player controls Pac-Man, who must eat all the dots in a closed maze while avoiding the colorful ghosts. By consuming large food pellets, known as power pellets, the ghosts turn white, allowing Pac-Man to eat them for bonus points. The primary goal is to accumulate points by collecting all the food in the maze and avoiding the roaming ghosts. If a ghost captures Pac-Man, the game ends.

1.2 Motivation

We aim to design an intelligent Pac-Man agent that finds optimal paths through the maze to achieve the goal state—eating all the dots while avoiding the ghosts in the minimum number of steps. To achieve this, we implemented several search algorithms. These include uninformed search algorithms (DFS, BFS, UCS) and informed search algorithms (A* search). The key difference between uninformed and informed search lies in the use of problem-specific information; uninformed search algorithms have no additional problem information, while informed algorithms leverage such information. Additionally, we implemented multi-agent algorithms such as ReflexAgent, Minimax, and Alpha-Beta. Using these algorithms, the Pac-Man agent will attempt to evade ghost agents and consume all the food in the maze to win the game.

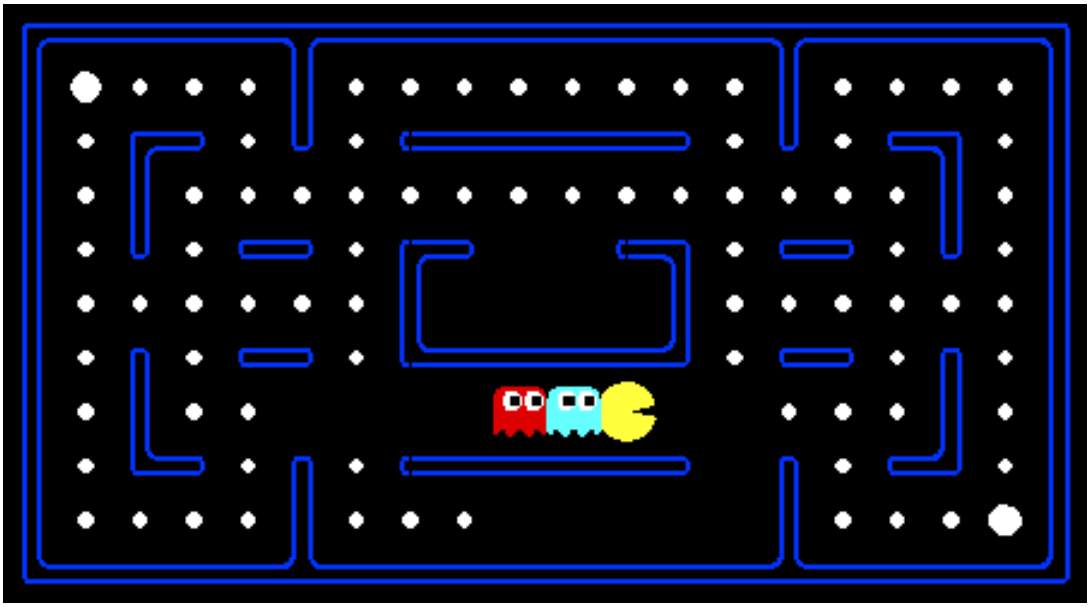

Figura 1: Pac-Man Interface

Figure 1 illustrates the Pac-Man game interface.

2 Uninformed Search

2.1 Depth-First Search

Depth-first search (DFS) always attempts to expand the deepest node in the search tree's stack. This algorithm is based on a stack data structure (LIFO - Last In, First Out). Using the stack, the most recently generated node is selected for expansion.

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"
    stack  = Stack()
    start_state = problem.getStartState()# first point in my graph
    stack .push((start_state,[]))

    marked= set() #uniqe set for not adding the point more than one time
    while not stack .isEmpty():
        current_state, path = stack .pop()
        if current_state not in marked:
            marked.add(current_state)

            if problem.isGoalState(current_state):
                return path
            # Explore neighbors
            for successor, action, step_cost in problem.getSuccessors(current_state):
                if successor not in marked:
                    stack .push((successor, path + [action]))

    util.raiseNotDefined()
```

The 'depthFirstSearch' function implements the depth-first search algorithm to find the solution for a given problem. Steps involved:

1. A stack (state stack) is created to manage states.
2. The start state is added to the stack along with an empty list of actions (path) associated with that state.
3. A set (visited) is initialized to track visited states.
4. While the stack is not empty:

- The last element (popped element) is removed from the stack.
- The current state and the path associated with that state are obtained.
- If the current state is the goal state, the path found up to that point is returned.
- If the current state has not been visited, it is marked as visited, and successors for the current state are obtained and iterated through. If a successor has not been visited, it is added to the stack with the updated path (path + [direction]).

2.2 Breadth-First Search

The Breadth-First Search (BFS) algorithm starts by expanding the root node, then expands all children of the root node, followed by their successors, and so on. Here, all nodes are expanded level by level, meaning that all nodes at a certain level are expanded before moving to the next level. BFS uses a queue (FIFO - First In, First Out data structure). Breadth-First Search guarantees a solution with the lowest cost in terms of effort required by Pac-Man to reach the food point.

```
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"

    queue =Queue()
    start_state = problem.getStartState()# first point in my graph
    queue.push((start_state,[]))

    marked= set() #uniqe set for not adding the point more than one time
    while not queue.isEmpty():
        current_state, path = queue.pop()
        if current_state not in marked:
            marked.add(current_state)

            if problem.isGoalState(current_state):
                return path
            # Explore neighbors
            for successor, action, step_cost in problem.getSuccessors(current_state):
                if successor not in marked:
                    queue.push((successor, path + [action]))

    util.raiseNotDefined()
    util.raiseNotDefined()
```

This code implements the Breadth-First Search (BFS) algorithm to find the solution for a given problem. Steps involved:

1. Create a queue (state queue) to manage states.
2. Add the start state to the queue along with an empty list of actions (path) associated with that state.
3. Initialize a set (visited) to track visited states.
4. While the queue is not empty:
   - Remove the first element (popped element) from the queue.
   - Obtain the current state and the path associated with that state.

- If the current state is the goal state, return the path found up to that point.
- If the current state has not been visited, mark it as visited, obtain successors for the current state, and iterate through them. If a successor has not been visited, add it to the queue with the updated path (path + [direction]).

2.3 Uniform Cost Search

The main characteristic of the Uniform Cost Search (UCS) algorithm is that, instead of expanding the deepest node, it tries to expand the node with the lowest path cost.

This is achieved using a priority queue data structure, where elements are ordered based on cost.

```
def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"

    priority_queue = PriorityQueue()
    start_state = problem.getStartState()# first point in my graph
    priority_queue.push((start_state,[],0),0)

    marked= dict()
    while not priority_queue.isEmpty():
        current_state, path, current_cost = priority_queue.pop()

        if problem.isGoalState(current_state):
            return path

        if current_state not in marked or current_cost < marked[current_state]:
            marked[current_state] = current_cost
            # Explore neighbors
            for successor, action, step_cost in problem.getSuccessors(current_state):
                new_cost = current_cost + step_cost
                if successor not in marked or new_cost < marked[successor]:
                    priority_queue.push((successor, path + [action], new_cost), new_cost)

    util.raiseNotDefined()
```

Steps involved:

1. The function uses a priority queue (state queue) to prioritize nodes based on total cost. The priority is determined by the cumulative cost of reaching the current state from the start state.
2. The initial state (start state) is added to the priority queue with an empty path and zero cost.
3. The function maintains a set of visited states to avoid revisiting the same state.
4. The main loop continues until the priority queue is empty, indicating that all possible paths have been explored.
5. In each iteration of the loop, the state with the lowest total cost is removed from the priority queue.
6. If the removed state is a goal state, the function returns the corresponding path.

7. Otherwise, successors for the current state are obtained, and for each successor, if it has not been visited, a new state is added to the priority queue with an updated path and total cost.
8. The 'util.PriorityQueue' ensures that states with lower total costs are explored first.

3 Informed Search

3.1 A* Search Algorithm

The A* search algorithm is a type of informed search algorithm that uses information about the problem before beginning the search. It evaluates nodes by adding $x(n)$, the cost to reach the node, and $y(n)$, the estimated cost to reach the goal from the node: $f(n) = x(n) + y(n)$. Since $x(n)$ represents the cost from the start node to node n, and $y(n)$ is the cost of the optimal path from n to the goal, $f(n)$ represents the total cost of the cheapest solution through n.

Therefore, if we want to find the cheapest solution, we should consider the node with the smallest $f(n)$ value.

The only difference between A* Search and UCS is that UCS considers only the cost to reach the node, while A* Search considers the sum of the cost to reach the node and the estimated cost from that node to the target (x + y) instead of x. Here, the heuristic function used is the Manhattan distance heuristic. This heuristic determines which node or state is closest to the goal state.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) -> List[Directions]:
    """Search the node that has the lowest combined cost and heuristic first."""
    # Priority Queue to hold nodes and their cost
    priority_queue = PriorityQueue()
    start_state = problem.getStartState()
    # Initialize the priority queue with the start state, an empty path, and a cost of 0
    priority_queue.push((start_state, [], 0), heuristic(start_state, problem))

    # To keep track of visited nodes and the cost of reaching them
    marked = dict()

    while not priority_queue.isEmpty():
        # Pop the node with the lowest f value (f = g + h)
        current_state, path, current_cost = priority_queue.pop()

        # If the current state is the goal, return the path
        if problem.isGoalState(current_state):
            return path

        # Mark the node as visited only if we have not visited it before or we have a cheape
        if current_state not in marked or current_cost < marked[current_state]:
            marked[current_state] = current_cost

            # Explore neighbors
            for successor, action, step_cost in problem.getSuccessors(current_state):
                # Calculate the new cost to reach the successor
                new_cost = current_cost + step_cost
```

```
                # Calculate the heuristic value for the successor
                heuristic_cost = heuristic(successor, problem)
                # Calculate the total cost (f = g + h)
                total_cost = new_cost + heuristic_cost

                # Push the successor to the priority queue with its cost and updated path
                if successor not in marked or new_cost < marked[successor]:
                    priority_queue.push((successor, path + [action], new_cost), total_cost)

    # If no solution is found, raise an error (or return an empty list depending on the requ
    util.raiseNotDefined()
```

Steps:

1. The function uses a priority queue (state queue) to prioritize nodes based on total cost and a heuristic. Priority is determined by the sum of the accumulated cost to the current state and the heuristic value for the next state.
2. The initial state (start state) is added to the priority queue with an empty path and zero cost. The heuristic value for the start state can be specified via the heuristic argument.
3. The function maintains a set of visited states to avoid revisiting the same state.
4. The main loop continues until the priority queue is empty, indicating all possible paths have been explored.
5. In each iteration of the loop, the state with the lowest priority value is removed from the priority queue.
6. If the removed state is a goal state, the function returns the corresponding path.
7. Otherwise, successors for the current state are obtained, and for each successor, if it has not been visited, a new cost is calculated (accumulated cost to the successor), and it is added to the priority queue with an updated priority value.
8. The ‘util.PriorityQueue‘ ensures that states with lower total and heuristic costs are explored first.

3.2 Finding All the Corners

The ‘getSuccessors‘ function from the ‘CornersProblem‘ class generates successor states for a given state in the Corners Search problem.

```
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))
        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, (False, False, False, False)

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

    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
        """

        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 ***"
            pos, corners = state
            x, y = pos
            dx, dy = Actions.directionToVector(action)
            next_x, next_y = int(x + dx), int(y + dy)
            corners = list(corners)
            hitsWall = self.walls[next_x][next_y]
            if not hitsWall:
                successor = (next_x, next_y)
                for i in range(len(corners)):
                    if self.corners[i] == successor:
                        corners[i] = True
                successors.append(((successor, tuple(corners)), 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.
        """
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)

```

Steps:

1. Iterate through possible directions (NORTH, SOUTH, EAST, WEST).
2. For each direction, calculate the new x and y coordinates based on the direction.
3. Check if the new x and y coordinates do not intersect a wall (not self.walls[nextx][nexty]).
4. If there is no wall, add a successor state to the ‘successors‘ list. This includes the new x and y coordinates, the action required to reach there, and a cost of 1.
5. Update ‘self.expanded‘ to track expanded nodes.

3.3 Corners Problem: Heuristic

The ‘cornersHeuristic‘ function is a heuristic used in the Corners Search problem. It provides an estimated distance between the current state and the farthest unvisited corner. The main goal of this heuristic is to give a lower-bound (admissible) estimate of the shortest path from the current state to a goal state (all corners visited).

```
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 ***"
    pos, visited = state
    manhattan = []

    if visited == (True, True, True, True):
        return 0

    for i in range(4):
        if not visited[i]:
            manhattan.append(util.manhattanDistance(corners[i], pos))
    return max(manhattan)
```

Steps:

1. Check if there are no unvisited corners (`if len(unvisited_corners) == 0`). In this case, the estimated distance is 0, as the goal has already been achieved.
2. Initialize ‘farthest$_d$istance‘$to0.Thiswillbethevaluereturnedbythefunction.Foreachunvisitedcorner, calcu$
3. If the calculated distance is greater than ‘farthest$_d$istance‘, $update‘farthest_distance‘.Attheend, thefunctionretu$ $boundestimateofthedistancetothefarthestunvisitedcorner.$

3.4 Eating All the Dots

The ‘foodHeuristic‘ function is a heuristic used in the Food Search problem (FoodSearchProblem). This heuristic provides an estimated distance between the current state and the nearest uneaten food dot. The primary purpose of this heuristic is to provide a lower-bound (admissible) estimate of the shortest path from the current state to a goal (eating all the food).

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your search may have a but our your heuristic is not admissible!  On the
    other hand, inadmissible heuristics may find optimal solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
```

```
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    distanceList = []
    for each in foodGrid.asList():
        distanceList.append(mazeDistance(position, each, problem.startingGameState))

    return max(distanceList, default=0)
```

Steps:

4. Obtain the list of coordinates of uneaten food dots using the ‘asList()‘ method of the ‘foodGrid‘ object.
2. Check if there are no uneaten food dots (`if len(uneaten) == 0`). In this case, the estimated distance is 0, as the goal has already been achieved.
3. Initialize ‘farthest$_d$istance‘to0.$Thiswillbethevaluereturnedbythefunction.Foreachuneatenfooddot, calcu$
4. If the calculated distance is greater than ‘farthest$_d$istance‘, $update$‘$farthest_distance$‘.$Finally, thefunctionretur$
$boundestimateofthedistancetothefarthestuneatenfooddot.$

3.5 Suboptimal Search

The ‘findPathToClosestDot‘ function is a method of the ‘ClosestDotSearchAgent‘ class and is responsible for returning a path (a list of actions) to the nearest uneaten food dot, starting from the current game state of Pac-Man (‘gameState‘).

This method is used within the ‘registerInitialState‘ method, which builds a list of actions to collect all the food in the maze.

```
def findPathToClosestDot(self, gameState: pacman.GameState):
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)
        return search.bfs(problem)
```

Steps:

5. **Input:**
    - ‘gameState‘: The current game state of Pac-Man, containing information about Pac-Man’s position, the food map, and other data.

2. **Method:**
   - Start by obtaining Pac-Man's current position using 'gameState.getPacmanPosition()'.
   - Then, obtain the food map using 'gameState.getFood()'.
   - Initialize an instance of the 'AnyFoodSearchProblem' class with the current game state. This search problem aims to find a path to any remaining food dot.
   - Use the BFS (breadth-first search) algorithm to find a path to the nearest food dot. Call 'search.bfs(problem)', where 'problem' is the search problem instance.
   - The method returns the found path.

4 Adversarial Search

4.1 Improve the ReflexAgent

Here, we create a ReflexAgent that, at each step, chooses a random legal action from those available. This is different from a random search agent, as a reflex agent does not build a sequence of actions but instead chooses and executes a single action. A capable reflex agent must consider both the food locations and ghost locations to perform well. We implemented the 'evaluationFunction' method in the 'ReflexAgent' class.

```
    def evaluationFunction(self, currentGameState: GameState, action):
        """
        Design a better evaluation function here.

        The evaluation function takes in the current and proposed successor
        GameStates (pacman.py) and returns a number, where higher numbers are better.

        The code below extracts some useful information from the state, like the
        remaining food (newFood) and Pacman position after moving (newPos).
        newScaredTimes holds the number of moves that each ghost will remain
        scared because of Pacman having eaten a power pellet.

        Print out these variables to see what you're getting, then combine them
        to create a masterful evaluation function.
        """
        # Useful information you can extract from a GameState (pacman.py)
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
        capsules = successorGameState.getCapsules()

        "*** YOUR CODE HERE ***"

        evaluation = successorGameState.getScore()

        # Distance to closest food
        foodDistances = [manhattanDistance(newPos, food) for food in newFood.asList()]
        closestFoodDistance = min(foodDistances) if foodDistances else 0  # Zero if no food
```

```
31
32          # Distance to ghosts
33          ghostDistances = [manhattanDistance(newPos, ghostState.getPosition()) for ghostState
34          closestGhostDistance = min(ghostDistances)
35
36          # Ghost behavior: scared vs active
37          for ghostState, scaredTime in zip(newGhostStates, newScaredTimes):
38              ghostDistance = manhattanDistance(newPos, ghostState.getPosition())
39              if scaredTime > 0:
40                  # Encourage eating scared ghosts
41                  evaluation += max(10, 200 - ghostDistance)  # Higher reward for closer scare
42              elif ghostDistance <= 1:
43                  # Penalize proximity to active ghosts
44                  evaluation -= 500
45
46          # Reward for eating capsules
47          if newPos in capsules:
48              evaluation += 500
49
50          # Reward for proximity to food
51          if closestFoodDistance > 0:
52              evaluation += 10 / closestFoodDistance
53
54          # Add penalties for remaining food (encourage clearing the board)
55          evaluation -= len(newFood.asList()) * 10
56
57          return evaluation
```

The ‘evaluationFunction‘ is used to evaluate a specific game state and a proposed action in the Pac-Man game. The steps are as follows:

1. Initialize variables and extract information from the current state:

- ‘successorGameState‘: Generates the game state after a given proposed action.
- ‘newPos‘: Gets Pac-Man’s new position from the successor state.
- ‘newFood‘: Obtains information about the remaining food in the successor state.
- ‘newGhostStates‘: Retrieves information about the ghosts’ states from the successor state.

2. Evaluate the successor state:

- **‘closestFoodDistance‘**: Initialized to infinity to calculate the distance to the nearest food dot.
- **‘closestGhostDistance‘**: Initialized to infinity to calculate the distance to the nearest ghost.
- **Iterate through food dots:** For each food dot in ‘newFood‘, calculate the distance to it and update ‘closestFoodDistance‘ with the smallest distance found.
- **Iterate through ghost states:** For each ghost in ‘newGhostStates‘, calculate the distance to it and update ‘closestGhostDistance‘ with the smallest distance found.
- **Overall evaluation:**
  - Initialize ‘evaluation‘ with the successor state’s score.
  - Subtract a significant penalty (-500) if Pac-Man is within one unit of the nearest ghost to avoid collisions.

- Add a value proportional to the inverse of the distance to the nearest food dot. The closer it is, the higher the evaluation.
- Add a bonus (+500) if Pac-Man is on a special pill.

3. Return the evaluation: The function returns the resulting evaluation of the successor state based on the above factors. This evaluation will be used to select the best action for Pac-Man at that moment in the game.

4.2 Minimax

The Minimax algorithm is a decision-making method used in two-player games where one player tries to maximize their gain, and the other player tries to minimize the first player's gain. We implemented the 'getAction' function in the 'MinimaxAgent' class.

```
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """

    def getAction(self, gameState: 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
        """
        "*** YOUR CODE HERE ***"
        def minimax(state, depth, agentIndex):
            # Base case: If depth is 0 or the game is in a terminal state
            if depth == 0 or state.isWin() or state.isLose():
                return self.evaluationFunction(state), None

            # Get legal actions for the current agent
            legalActions = state.getLegalActions(agentIndex)
```

```
37
38              # If no legal actions, return the evaluation of the current state
39              if not legalActions:
40                  return self.evaluationFunction(state), None
41
42              # Pacman's turn (maximize)
43              if agentIndex == 0:
44                  bestValue = float("-inf")
45                  bestAction = None
46                  for action in legalActions:
47                      successor = state.generateSuccessor(agentIndex, action)
48                      value, _ = minimax(successor, depth - 1 if agentIndex == state.getNumAge
49                                         (agentIndex + 1) % state.getNumAgents())
50                      if value > bestValue:
51                          bestValue, bestAction = value, action
52                  return bestValue, bestAction
53
54              # Ghosts' turn (minimize)
55              else:
56                  bestValue = float("inf")
57                  bestAction = None
58                  for action in legalActions:
59                      successor = state.generateSuccessor(agentIndex, action)
60                      value, _ = minimax(successor, depth - 1 if agentIndex == state.getNumAge
61                                         (agentIndex + 1) % state.getNumAgents())
62                      if value < bestValue:
63                          bestValue, bestAction = value, action
64                  return bestValue, bestAction
65
66          # Start the minimax algorithm with Pacman (agentIndex = 0) and initial depth
67          _, bestAction = minimax(gameState, self.depth, 0)
68          return bestAction
```

The **'getAction'** function is the main function for Minimax agents. It calculates the best action using the Minimax algorithm, based on the given depth ('self.depth') and evaluation function ('self.evaluationFunction'). The **'minimax'** function is the recursive implementation of the Minimax algorithm.

1. If the depth is 0 or the state is a win/loss, return the evaluated value and no action ('self.evaluationFunction(state), None').
2. **Maximization and minimization:**
   - If it is Pac-Man's turn (maximizing player), initialize 'bestValue' to $-\infty$ to find the highest possible value.
   - If it is a ghost's turn (minimizing player), initialize 'bestValue' to $+\infty$ to find the lowest possible value.
3. **Iterate through legal actions:**
   - For each legal action of the current player, generate the successor state and recursively calculate its value using the 'minimax' function.
   - Update 'bestValue' and 'bestAction' based on the calculated value for the successor state.

4. At the end, the ‘minimax‘ function returns the best value and the action associated with it.

Calling the ‘minimax‘ function: In ‘getAction‘, the ‘minimax‘ function is called with the current game state (‘gameState‘), depth, and agent index (0 for Pac-Man). The final result of the Minimax algorithm is the optimal action for the given state, which is returned and used as the recommended action for the Minimax agent.

4.3 Alpha-Beta Pruning

The Alpha-Beta Pruning algorithm is an enhancement of the Minimax algorithm, designed to reduce the number of nodes explored in a game tree without affecting the final decision made by the algorithm. It achieves this by updating the alpha and beta values to exclude certain branches from exploration. Consequently, the Alpha-Beta Pruning algorithm can be significantly more efficient than the Minimax algorithm in exploring the state space.

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"
        def alphaBeta(state, depth, alpha, beta, agentIndex):
            # Base case: stop recursion if depth is 0, or the game state is a win/lose state
            if depth == 0 or state.isWin() or state.isLose():
                return self.evaluationFunction(state), None  # Return the evaluation score

            # Pacman's turn (maximizing agent)
            if agentIndex == 0:
                bestValue = float("-inf")  # Initialize the best value as negative infinity
                for action in state.getLegalActions(agentIndex):  # Loop through all legal
                    successor = state.generateSuccessor(agentIndex, action)  # Generate the
                    # Recursively call alphaBeta for the next agent
                    value, _ = alphaBeta(successor, depth - 1, alpha, beta, (agentIndex + 1)
                    if value > bestValue:  # Update the best value and action if this actio
                        bestValue = value
                        bestAction = action
                    if bestValue > beta:  # Beta cut-off: stop exploring further as it won't
                        return bestValue, bestAction
                    alpha = max(alpha, bestValue)  # Update alpha for the maximizing player
                return bestValue, bestAction  # Return the best value and corresponding acti

            # Ghost's turn (minimizing agent)
            else:
                bestValue = float("inf")  # Initialize the best value as positive infinity
                for action in state.getLegalActions(agentIndex):  # Loop through all legal
```

```
                    successor = state.generateSuccessor(agentIndex, action)  # Generate the
                    # Recursively call alphaBeta for the next agent
                    value, _ = alphaBeta(successor, depth - 1, alpha, beta, (agentIndex + 1)
                    if value < bestValue:  # Update the best value and action if this actior
                        bestValue = value
                        bestAction = action
                    if bestValue < alpha:  # Alpha cut-off: stop exploring further as it wor
                        return bestValue, bestAction
                    beta = min(beta, bestValue)  # Update beta for the minimizing player
                return bestValue, bestAction  # Return the best value and corresponding acti

        # Start the alpha-beta pruning algorithm
        _, bestAction = alphaBeta(gameState, self.depth * gameState.getNumAgents(), float("-
        return bestAction  # Return the best action for Pacman
```

The **getAction** function is the primary function of Alpha-Beta agents. It calculates the best action using the Alpha-Beta Pruning algorithm, based on the given depth (`self.depth`) and the evaluation function (`self.evaluationFunction`). The **alphaBeta** function is the recursive function that implements the Alpha-Beta algorithm.

Steps:

1. **Base Case:** If the depth is 0 or the state is a win or loss, return the evaluated value and no action (`self.evaluationFunction(state)`, None).
2. **Maximization and Minimization with Alpha-Beta Cuts:**
   - **For Pac-Man (Maximizing Player):** Attempt to maximize the value. Iterate through legal actions and calculate their associated values, updating alpha and beta accordingly. For each action, if the value is greater than `bestValue` (the best value found so far), update `bestValue` and `bestAction`. Perform a Beta cut-off if the value exceeds beta, and return the value and action.
   - **For Ghosts (Minimizing Players):** Similarly, attempt to minimize the value. Iterate through legal actions and calculate their associated values, updating alpha and beta accordingly. For each action, if the value is less than `bestValue`, update `bestValue` and `bestAction`. Perform an Alpha cut-off if the value is less than alpha, and return the value and action.
3. **Final Return:** The `alphaBeta` function returns the best value and its associated action, with Alpha-Beta cuts applied to reduce unnecessary node exploration.
4. **Calling the `alphaBeta` Function:** In `getAction`, the `alphaBeta` function is called with the current game state (`gameState`), the depth, and the initial alpha and beta values. The final result of the Alpha-Beta algorithm is the optimal action for the given state, which is returned and used as the recommended action for the Alpha-Beta agent.

Steps:

1. Calculate the minimum distance to a food dot to emphasize the importance of moving closer to the food source.
2. Iterate through all ghosts, and if a ghost is in immediate proximity (at a distance of 1), subtract a significant score (-500) to avoid encountering them.
3. Otherwise, add a score based on the inverse of the distance to the nearest food dot. The closer Pac-Man is to the food, the higher the score.
4. Add a significant score (+500) if Pac-Man's current position is in a capsule, indicating a strategic advantage when Pac-Man eats a capsule and becomes capable of confronting the ghosts.