**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**Department of Software and Automation Engineering**

**Laboratory work No. 2**

**(Intelligent) Searching Algorithms**

**Elaborated: Dan Hariton**

**st. FAF-211**

**Checked: Elena Graur**

**un. assistant.**

**Chişinău, 2024**

**1 THE TASK OF THE LABORATORY WORK**

Imagine being a local in Luna-City and living our best life there! All day long you are just ”hunting out” those awesome-tasting pallets and delight in those goodies! And now, this Ghostling Family decided to move to Luna-City! Everything would be fine if only they could control their little monsters children. They annoyingly follow you scaring you all around without leaving a chance to live your peaceful life and enjoy your much-loved treats! That’s exactly the case of the new client of ”HeinLeinAI”, who came demanding a solution that will help him at all costs avoid his new ”beloved” neighbours. Since you performed greatly on your previous task and know some characteristics of newbies in Luna-City, the company considered you the best fit for the task. Be aware! The solution you develop should grant the client both: avoiding Ghostling Family and gathering as many pallets as possible!

**2 THE PROGRESS OF THE WORK**

**2.1 Implement the MiniMax Algorithm with the following scoring function**

MiniMax is a decision-making algorithm used in adversarial environments, such as games. It recursively explores possible future states (moves) of both players (Pacman and the ghosts in this case) and evaluates them using a scoring function. Pacman tries to **maximize** the score, while the ghosts attempt to **minimize** it.

For Our Code we have this formula:



* **Pallet Score**: The distance to the nearest food (the closer, the better).
* **Ghost Danger**: The distance to the nearest ghost (the further, the better).

The core of MiniMax involves recursively exploring future game states. Pacman tries to maximize the score, while ghosts minimize it. Here’s a simplified version of how I implemented MiniMax:

class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        def minimax(agentIndex, depth, gameState):

            if depth == 0 or gameState.isWin() or gameState.isLose():

                return self.evaluationFunction(gameState)

            if agentIndex == 0:

                return max\_value(agentIndex, depth, gameState)

            else:

                return min\_value(agentIndex, depth, gameState)

        #code

        return bestAction

def max\_value(agentIndex, depth, gameState):

    #code

    return maxScore

def min\_value(agentIndex, depth, gameState):

    #code

    return minScore

**2.2 Implement Alpha-Beta Pruning**

Alpha-Beta Pruning is an optimization technique for the MiniMax algorithm. It reduces the number of nodes that need to be evaluated by "pruning" branches that cannot affect the final decision. The main goal is to skip unnecessary calculations and make the search process more efficient.

* Alpha: The best value that Pacman (the maximizing player) can guarantee so far.
* Beta: The best value that the ghosts (the minimizing players) can guarantee so far.
* Pruning: If Pacman finds a move with a score higher than the current best beta value (for  
  the ghosts), it knows that the ghosts will never allow this move. This branch is "pruned" (skipped).

class AlphaBetaAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        def alphaBeta(agentIndex, depth, gameState, alpha, beta):

            if depth == 0 or gameState.isWin() or gameState.isLose():

                return self.evaluationFunction(gameState)

            if agentIndex == 0:

                return max\_value(agentIndex, depth, gameState, alpha, beta)

            else:

                return min\_value(agentIndex, depth, gameState, alpha, beta)

        def max\_value(agentIndex, depth, gameState, alpha, beta):

                #code

                if maxScore >= beta:  # Beta cutoff (prune)

                    return maxScore

                alpha = max(alpha, maxScore)  # Update alpha

            return maxScore

        def min\_value(agentIndex, depth, gameState, alpha, beta):

            #code

                if minScore <= alpha:  # Alpha cutoff (prune)

                    return minScore

                beta = min(beta, minScore)  # Update beta

            return minScore

        alpha = float('-inf')

        beta = float('inf')

        return bestAction

**2.3 Implement an improved scoring (evaluation) method for MiniMax. For example,you could add values like MazeComplexity, PalletNumber per region or GhostVulnerability. Becreative!.**

In this task, I updated the basic Pallet Score - Ghost Danger evaluation and introduce additional factors to create a more sophisticated scoring function. The goal is to make Pacman smarter by taking into account more aspects of the game beyond just food and ghosts.

#### Key Improvements in the Evaluation Function:

* **Remaining Pallets:** Reward Pacman more as the number of remaining food pallets decreases. This encourages Pacman to complete the game faster.
* **Capsule Distance:** Give Pacman a bonus for being close to power pellets (capsules) that allow him to scare ghosts.
* **Scared Ghost Vulnerability:** Encourage Pacman to chase scared ghosts by giving bonus points when ghosts are vulnerable.

def betterEvaluationFunction(currentGameState):

    pacmanPos = currentGameState.getPacmanPosition()

    foodList = currentGameState.getFood().asList()

    capsuleList = currentGameState.getCapsules()

    ghostStates = currentGameState.getGhostStates()

    ghostPositions = currentGameState.getGhostPositions()

    score = currentGameState.getScore()

  foodDistances = [manhattanDistance(pacmanPos, food) for food in foodList]

  capsuleDistances = [manhattanDistance(pacmanPos, capsule) for capsule in capsuleList]

    ghostDanger = 0

    scaredGhostBonus = 0

    score += scaredGhostBonus

    score -= ghostDanger

    return score

**2.4 Add at least one improvement to the MiniMax algorithm from the following list: Progressive Deepening, Transposition Tables, Opening Books, Move Ordering, Aspiration Window, etc .**

In this task, we implemented two key improvements to the MiniMax algorithm:

* **Progressive Deepening:** This technique allows the algorithm to incrementally increase the search depth, starting from a shallow depth and going deeper as time permits. If the algorithm runs out of time, it can return the best move found so far at the current depth.
* **Move Ordering**: This optimization orders the possible moves based on a heuristic evaluation. The best moves (those expected to lead to the best outcomes) are explored first.

class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        def minimax(agentIndex, depth, gameState, alpha, beta):

        def max\_value(agentIndex, depth, gameState, alpha, beta):

            maxScore = float('-inf')

            legalActions = self.getOrderedLegalActions(gameState, agentIndex)

 #Use ordered actions

            return maxScore

        def min\_value(agentIndex, depth, gameState, alpha, beta):

            minScore = float('inf')

            legalActions = self.getOrderedLegalActions(gameState, agentIndex)  # Use ordered actions

          return minScore

        # Progressive Deepening: Start with depth 1 and incrementally deepen the search

        bestAction = None

        startTime = time.time()

        maxDepth = self.depth  # The maximum depth we want to reach

        currentDepth = 1

        timeLimit = 2  # Set a reasonable time limit (in seconds) for searching

        while currentDepth <= maxDepth:

            legalActions = gameState.getLegalActions(0)

            bestScore = float('-inf')

            # Iterating through actions with the current depth

            for action in legalActions:

                successor = gameState.generateSuccessor(0, action)

                score = minimax(1, currentDepth, successor, float('-inf'), float('inf'))

        return bestAction

    def getOrderedLegalActions(self, gameState, agentIndex):

            # Sort actions by their evaluation scores in descending order

            actionScores.sort(reverse=True, key=lambda x: x[0])

            orderedActions = [action for score, action in actionScores]

            return orderedActions

**2.5 Improve the Path Finding algorithm for the Agent using the A-Star algorithm. Combine it with the implemented MiniMax algorithm.**

In this task, I combined the A pathfinding algorithm*\** with the MiniMax algorithm to help Pacman make more efficient and intelligent decisions. A\* allows Pacman to find the shortest path to food or capsules, while MiniMax allows Pacman to balance between avoiding ghosts and collecting food.

A\* is an informed search algorithm used for finding the shortest path between two points. It uses:

* **g(n)**: The cost to reach the current node from the start (typically the distance).
* **h(n)**: A heuristic estimate of the cost to reach the goal from the current node (e.g., Manhattan distance).
* **f(n) = g(n) + h(n)**: The total estimated cost of reaching the goal through the current node.

A\* allows Pacman to plan the most efficient route toward food or capsules, while MiniMax helps Pacman make the best strategic decision based on the overall game state.

### How It Works:

* A Pathfinding*\*:* Pacman uses A\* to determine the best path to the nearest food or capsule. The result is a list of actions that lead Pacman to the goal.
* **MiniMax Decision Making**: After finding the path using A\*, MiniMax evaluates the current game state and decides whether following the A\* path is the best strategy or if other moves should be considered (e.g., if there is a nearby ghost, MiniMax may choose a safer move).
* **Fallback to MiniMax**: If A\* does not find a valid path (or if MiniMax finds a better option), Pacman will use the default MiniMax decision-making process.

**2.6 Combine it with the implemented Alpha-Beta Pruning algorithm.**

In this task, I combined the A pathfinding algorithm\* with the Alpha-Beta Pruning algorithm to further optimize Pacman’s decision-making. This integration leverages the benefits of both:

* A\* Pathfinding: Finds the optimal path to the nearest food or capsule.
* Alpha-Beta Pruning: Improves the search efficiency of the MiniMax algorithm by pruning branches that don’t need to be explored, ensuring better performance in adversarial situations (against ghosts).

### How It Works:

* A\* Pathfinding: Pacman first uses A\* to find the shortest path to the nearest food or capsule. The result is a list of actions leading to the goal.
* **Alpha-Beta Pruning**: After A\* provides the potential actions, Alpha-Beta Pruning evaluates the current game state, pruning branches that are irrelevant to the decision-making process. If the best action found by A\* is still optimal after evaluation, Pacman will follow the A\* path. If not, Alpha-Beta will choose a different action.
* **Pruning Unnecessary Branches**: Pacman explores fewer game states because Alpha-Beta Pruning skips branches that don’t improve the final decision (i.e., when a move is guaranteed to be worse than previously explored moves). This results in a more efficient search compared to standard MiniMax.

**CONCLUSIONS**

In this laboratory work, I successfully implemented several intelligent algorithms to enhance Pacman's decision-making in an adversarial game environment.

Starting with the implementation of the MiniMax algorithm, I defined a scoring function that prioritized Pacman’s movement toward food while penalizing close proximity to ghosts. Building on this, I implemented Alpha-Beta Pruning to optimize the MiniMax search process. By pruning branches that couldn't influence the final decision, significantly improved the efficiency of the search.

I enhanced Pacman’s scoring mechanism by adding new elements such as maze complexity, capsule proximity, and scared ghost vulnerability. These improvements provided Pacman with a more nuanced understanding of the game state, allowing it to make more informed decisions beyond basic food collection and ghost avoidance.

I further extended Pacman's decision-making by implementing Progressive Deepening and Move Ordering. These two improvements allowed Pacman to incrementally deepen its search while ensuring the most promising moves were explored first. This combination ensured that Pacman could always return the best possible move even within time constraints, while also improving search efficiency with better pruning in Alpha-Beta.

Lastly, I combined A pathfinding with both MiniMax and Alpha-Beta Pruning, giving Pacman the ability to efficiently plan its route to food and capsules while maintaining a strategic overview of the entire game state. The integration of A\* allowed Pacman to find optimal paths while Alpha-Beta ensured it avoided unnecessary risks, providing a balance between short-term gains and long-term strategy.

**BIBLIOGRAPHY**

**1**. Lague, Sebastian. “Algorithms Explained – Minimax and Alpha-Beta Pruning.” *YouTube*, 20 Apr. 2018, [www.youtube.com/watch?v=l-hh51ncgDI](http://www.youtube.com/watch?v=l-hh51ncgDI) . Accessed 4 Oct. 2024.