University: Isfahan University - Faculty of Computer Engineering

Course: Fundamentals and Applications of Artificial Intelligence
Minimax Algorithm with alpha-beta pruning
Course Instructor: Dr. Hossein Karshenas

Student Name: **Sheida Abedpour**
Student ID: 4003623025

Team_id = 17

2023 January

Adversial Search:

Adversarial search in artificial intelligence is used to resolve two-person games in which one player seeks to maximize their score while the other tries to minimize it.

The development of algorithms and tactics for making decisions in competitive settings where numerous agents or players have competing interests is the focus of adversarial search in artificial intelligence. Finding the optimum movements or actions for a player while accounting for opponents' actions and prospective responses is the major objective of the adversarial search.

The objective of the adversarial search is to create intelligent algorithms that can choose the best course of action under conflicting circumstances while taking into account the moves of rivals and their potential retaliations. To find the optimum move or action to make, adversarial search algorithms often search through a **game tree**, which represents all feasible game states and their transitions.

Zero-Sum Game:

According to game theory, a zero-sum game is one in which each player's overall winnings or losses are equal to zero. In other words, every profit made by one player is offset by an equivalent loss suffered by another. The interests of the players are wholly at odds with one another in a zero-sum game, and the overall reward is fixed.

The reward matrix, which depicts each player's outcomes or payouts based on their strategies, has the property that the sum of payouts for all players at any given outcome is zero in a zero-sum game. This means that any increase in one player's payout must be offset by a corresponding decrease in another player's payoff.

Game Tree:

A **game tree** is a visual representation of every action and consequence that can be made in a two-player game when using adversarial search in artificial intelligence. The game tree begins with the beginning state of the game, and

each node in the tree represents a different state of the game. These states include details about which player is taking their turn, the layout of the board, and other pertinent information.

The game tree is frequently used in conjunction with search algorithms like **minimax** or **alpha-beta pruning** to determine a player's best course of action under the assumption that the other player would play similarly well. The search algorithm can decide the best move for a player by examining the game tree and considering all feasible moves and outcomes.

Minimax Algorithm:

The Minimax Algorithm is a recursive decision-making algorithm, which makes an optimal move for a player, assuming the opponent is playing optimally.

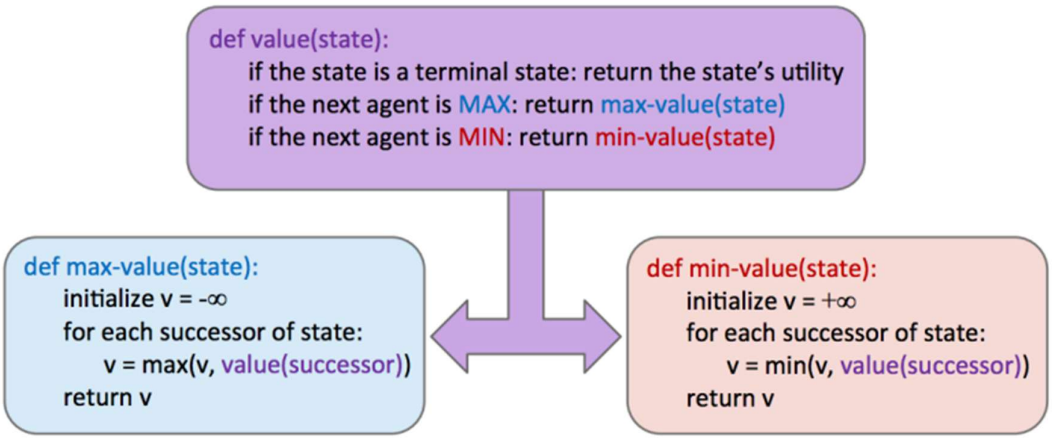
It makes 3 assumptions about the game in order to do so:

1. The game must be turn-based in order to properly generate a game tree.
2. The opponent must be attempting to play optimally. The algorithm becomes less effective if the opponent is not playing by a strategy.
3. The game must be purely strategic, involving no luck or secret information. While modifications on the minimax algorithm will allow it to function in the case of randomness, it will no longer be perfect.

In order to make to make an optimal decision, the algorithm must evaluate the entirety of the game tree.

The game tree starts with the current state of the board, which forms the root node of the tree. Each possible move the next player can take is represented as a child of the root node, then each move that the other player can take is represented as a child of that new node, and so on. The result is a tree that contains *every possible combination of board states that are possible within N nodes, shown above as levels.*

For each of those child nodes, we can calculate it’s value: whether that position is good for any given player.



minimax algoritm pseudocode

Minimax is a simple enough algorithm that it can be theoretically implemented anywhere: however, the biggest risk to Minimax is Time and Space constraints.

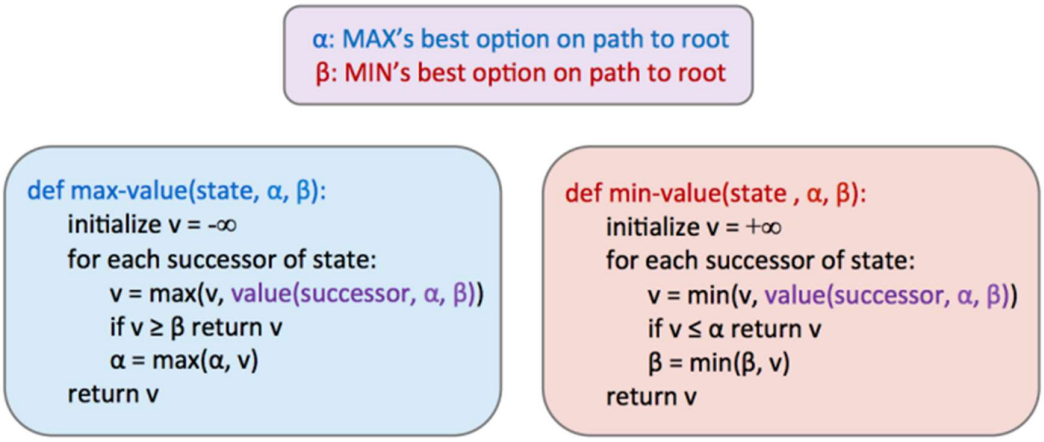
Minimax's *Time Complexity* is O(b^m), where b is branching factor of the game-tree, and m is the maximum depth of the tree. While methods such as **Alpha-Beta Pruning** can cut down on the time somewhat- if the task has too many options to run through quickly, or the Machine is too slow to run that many calculations, then Minimax can't be used effectively. While reducing the depth *m* is always an option- reducing it too low will cause the AI to make short-sighted decisions.

Similarly, Minimax has a *Space Complexity* of O(bm). This Makes Minimax a poor choice if Memory is a precious commodity. While this is often not the case in Games today, it is important to note when developing for old systems.

Alpha-Beta Pruning:

Minimax seems just about perfect - it's simple, it's optimal, and it's intuitive. Yet, its execution is very similar to depth-first search and it's time complexity is identical, a dismal O(b^m). This yields far too great a runtime for many games.

Conceptually, alpha-beta pruning is this: if you're trying to determine the value of a node n by looking at its successors, stop looking as soon as you know that n's value can at best equal the optimal value of n's parent. This is implemented as follows:



alpha-beta puring in minimax algorithm pseudocode

Evaluation Functions :

Though alpha-beta pruning can help increase the depth for which we can feasibly run minimax, this still usually isn't even close to good enough to get to the bottom of search trees for a large majority of games. As a result, we turn to evaluation functions, functions that take in a state and output an estimate of the true minimax value of that node. Typically, this is plainly interpreted as "better" states being assigned higher values by a good evaluation function than "worse" states. Evaluation functions are widely employed in depth-limited minimax, where we treat non-terminal nodes located at our maximum solvable depth as terminal nodes, giving them mock terminal utilities as determined by a carefully selected evaluation function. Because evaluation functions can only yield estimates of the values of non-terminal utilities, this removes the guarantee of optimal play when running minimax. A lot of thought and experimentation is typically put into the selection of an evaluation function when designing an agent that runs minimax, and the better the evaluation function is, the closer the agent will come to behaving optimally. Additionally, going deeper into the tree before using an evaluation function also tends to give us better results - burying their computation deeper in the game tree mitigates

the compromising of optimality. These functions serve a very similar purpose in games as heuristics do in standard search problems.

Minimax with alpha-beta pruning implemention in Python:

Minimax with alpha-beta pruning

```python
def max_level(gameState, depth, alpha, beta, agentIndex=0):

    if gameState.isWin() or gameState.isLose() or depth == self.depth:
        return scoreEvaluationFunction(currentGameState=gameState)

    max_value = float('-inf')

    legal_actions = gameState.getLegalActions(agentIndex=agentIndex)
    for action in legal_actions:
        successor = gameState.generateSuccessor(agentIndex=agentIndex, action=action)
        max_value = max(max_value, min_level(gameState=successor, depth=depth + 1,
                                             alpha=alpha, beta=beta, agentIndex=1))

        if max_value >= beta:
            return max_value
        alpha = max(alpha, max_value)

    return max_value

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

    if gameState.isWin() or gameState.isLose():
        return scoreEvaluationFunction(currentGameState=gameState)

    min_value = float('inf')

    legal_actions = gameState.getLegalActions(agentIndex=agentIndex)
    for action in legal_actions:
        successor = gameState.generateSuccessor(agentIndex=agentIndex, action=action)

        if agentIndex == (gameState.getNumAgents() - 1):
            min_value = min(min_value, max_level(gameState=successor, depth=depth, alpha=alpha, beta=beta))
        else:
            min_value = min(min_value, min_level(gameState=gameState, depth=depth, alpha=alpha, beta=beta,
                                                 agentIndex=(agentIndex + 1)))

        if min_value <= alpha:
            return min_value
        beta = min(beta, min_value)

    return min_value

agent_index = 0
ghost_index = 1
initial_depth = 0

alpha = float('-inf')  # max's best option on path to root
beta = float('inf')  # min's best option on path to root

scores_record = []
actions_record = []

legal_actions = gameState.getLegalActions(agentIndex=agent_index)
for action in legal_actions:
    successor = gameState.generateSuccessor(agentIndex=agent_index, action=action)
    score = min_level(successor, depth=initial_depth, alpha=alpha, beta=beta, agentIndex=ghost_index)

    if score >= alpha:
        alpha = score
        scores_record.append(score)
        actions_record.append(action)

best_actions = [index for index in range(len(scores_record)) if scores_record[index] == alpha]
return_action = actions_record[random.choice(best_actions)]

return return_action
```

- The **max_level** function represents the maximizing player's perspective. It evaluates the maximum possible score achievable for the current game state by exploring possible actions and the subsequent minimizing player's responses.
- **Parameters:**
    - **gameState**: The current state of the game.
    - **depth**: The current depth in the search tree.
    - **alpha**: The best score that the maximizing player has encountered so far on the path to the root.
    - **beta**: The best score that the minimizing player has encountered so far on the path to the root.
    - **agentIndex**: The index of the current agent (default is 0, representing the maximizing player).

- The **min_level** function represents the minimizing player's perspective. It evaluates the minimum possible score achievable for the current game state by exploring possible actions and the subsequent maximizing player's responses.
- **Parameters:**
    - Similar to **max_level**, with **agentIndex** being the index of the current agent.

The code includes alpha-beta pruning, which is a technique to reduce the number of nodes evaluated in the search tree. It maintains two values, **alpha** and **beta**, representing the best scores for the maximizing and minimizing players, respectively. The pruning occurs when a better option is found, and the search can be cut off.

Evaluation Function for pac-man game implementation in Python:

Minimax with alpha-beta pruning

```python
def betterEvaluationFunction(currentGameState: GameState):
    score_state = currentGameState.getScore()

    pacman_position = currentGameState.getPacmanPosition()

    ghosts_positions = currentGameState.getGhostPositions()
    ghosts_distance = []
    scared_ghosts_distance = []
    ghost_index = 1
    for ghost in currentGameState.getGhostStates():
        if ghost.scaredTimer == 0:
            ghosts_distance.append([manhattanDistance(pacman_position,
                                                      currentGameState.getGhostPosition(ghost_index))])
        else:
            scared_ghosts_distance.append([manhattanDistance(pacman_position,
                                                             currentGameState.getGhostPosition(ghost_index))])
        ghost_index += 1

    foods_available = currentGameState.getFood().asList()
    foods_distance = [manhattanDistance(pacman_position, food_position)
                      for food_position in foods_available]

    capsules_available = currentGameState.getCapsules()
    capsules_distance = [manhattanDistance(pacman_position, capsule_position)
                         for capsule_position in capsules_available]

    score_food = 0
    if len(foods_available) != 0:
        avg_foods_distances = sum(foods_distance) / len(foods_available)
        closest_food = min(foods_distance)
        score_food = 0.8 * closest_food + 0.2 * avg_foods_distances

    score_capsule = 0
    if len(capsules_distance) != 0:
        closest_capsule = min(capsules_distance)
        ghost_capsule_distances = [manhattanDistance(ghost_position, capsule_position)
                                   for ghost_position, capsule_position
                                   in zip(ghosts_positions, capsules_available)]
        avg_ghost_capsule_distance = sum(ghost_capsule_distances) / len(ghost_capsule_distances)
        score_capsule = 0.9 * avg_ghost_capsule_distance + 0.1 * closest_capsule

    closest_ghost = 0
    closest_scared_ghost = 0
    if len(ghosts_distance) != 0:
        closest_ghost = min([distance[0] for distance in ghosts_distance])
    if len(scared_ghosts_distance) != 0:
        closest_scared_ghost = min([distance[0] for distance in scared_ghosts_distance])

    score_ghost = 0.2 * sum([distance[0] for distance in ghosts_distance]) + 0.8 * closest_ghost
    score_scared_ghost = closest_scared_ghost

    sum_x_ghosts_position = 0
    sum_y_ghosts_position = 0
    for ghost in ghosts_positions:
        x, y = ghost
        sum_x_ghosts_position += x
        sum_y_ghosts_position += y
    avg_x_ghosts_position = sum_x_ghosts_position / len(ghosts_positions)
    avg_y_ghosts_position = sum_y_ghosts_position / len(ghosts_positions)

    x_pacman_position, y_pacman_position = pacman_position
    if abs(x_pacman_position - avg_x_ghosts_position) <= 2 \
            and abs(y_pacman_position - avg_y_ghosts_position) <= 2:
        score_ghost = float('-inf')

    features = [score_food,
                score_capsule,
                score_ghost,
                score_scared_ghost,
                score_state]

    weight_food = 5
    weight_capsule = 15
    weight_ghost = -200
    weight_scared_ghost = 200
    weight_state = 500

    weights = [weight_food,
               weight_capsule,
               weight_ghost,
               weight_scared_ghost,
               weight_state]

    estimate_score = sum([weight * feature for weight, feature in zip(weights, features)])

    return estimate_score
```

Refrences:

- github.com/karlapalem/UC-Berkeley-AI-Pacman-Project/blob/master/multiagent
- github.com/khanhngg/CSC665-multi-agent-pacman/blob/master/multiagent

Python Libreries used in project:

- sklearn