Project IV - Adversarial Search
Bomberman

Dăscălescu Dana
University of Bucharest
Faculty of Mathematics and Computer Science

April 27, 2023

Contents

Introduction

Bomberman is a widely known action-oriented video game in which the player navigates a labyrinthine setting, destroys obstacles, and defeats enemies by placing and detonating bombs. The original Bomberman was initially launched in 1983 for the MSX computer system and has since been released on many different platforms, including consoles and mobile devices. Throughout its history, the game has undergone various transformations and has been modified to encompass diverse configurations, such as individual and collaborative gameplay options.

In this particular project, Bomberman has been modified into a game that is deterministic, turn-based, two-player, and involves zero-sum outcomes with perfect information. Consequently, the game is now a two-player game wherein each player has comprehensive knowledge of the game's current state and all possible moves that can be made from those positions. The current gameplay involves a turn-based structure, where participants alternate in moving one tile and either planting an explosive device or triggering a previously installed one.

In order to solve this game, we have implemented Adversarial Search Algorithms, such as Minimax and Alpha-Beta Pruning. These algorithms enable us to find the optimal moves for each player, assuming that the opponent will also make their best possible move. By doing so, we can create an AI opponent that is capable of playing the game at a high level, making the game more challenging and engaging for players.

Algorithms overview

To represent the game, we will use a game tree where each node represents a state of play and each edge represents a possible move. The root node of the tree represents the initial state of the game, and the leaf nodes represent the terminal states of the game.

The game is turn-based, with the MAX player making the first move, followed by the MIN and MAX players taking turns until the end of the game. In the game tree, each level represents the moves of a specific player, starting with the MAX player at the root node, followed by the MIN player at the next level, and so on.

To determine the winner of the game, a utility function will be used to assign a numerical value to the outcome of the game. The utility function could provide points to the winning player or penalize the losing player. The aim of the MAX player is to maximize their score, while the MIN player's goal is to minimize the score of the MAX player. By using this approach, we can represent the game as a zero-sum game, where the total utility of both players is equal to zero.

The use of a game tree and a utility function is fundamental to implementing Adversarial Search Algorithms such as Minimax and Alpha-Beta Pruning. These algorithms enable us to determine the optimal moves for each player, given the assumption that the opponent will also make their best possible move.

Minmax

In order to implement the Minmax algorithm, the first step is to generate the entire game tree, up to the terminal states. Each of these terminal states is then evaluated using a utility function that assigns a numerical value to the outcome of the game.

Next, the algorithm proceeds to move back up the tree from the leaf nodes to the root nodes. At each level, the algorithm determines the values that represent the utility of the nodes at that level. The values are propagated to previous levels through successive parent nodes, with MAX nodes being assigned the maximum value of their child nodes, and MIN nodes being assigned the minimum value of their child nodes.

Finally, when the algorithm reaches the root node, it chooses the move that leads to the maximum value for MAX. This move is considered the optimal move for the player to make, assuming that the opponent will also make their best possible move. By repeating this process for each turn in the game, the Minmax algorithm is able to determine the best possible sequence of moves for the player to make in order to maximize their chances of winning.

!!! The main variation defines the optimal minmax game for both players. Empirical evidence suggests that the positions along the main variation exhibit a consistent value, implying that strategic moves that preserve the game's value are optimal.

Alpha-Beta Pruning

Alpha-beta pruning is a variation of the Minimax algorithm that can significantly reduce the number of nodes that need to be evaluated, by avoiding the evaluation of branches that are guaranteed to be suboptimal.

The Alpha-Beta Pruning algorithm involves the following steps:

1. The game tree is generated up to the terminal states.
2. The utility function is applied to each terminal state to obtain the value corresponding to the state.
3. As the tree is traversed from the leaf nodes to the root nodes, the algorithm prunes branches that cannot possibly affect the final decision. This is done by maintaining two values, alpha and beta, at each level of the tree:
   - Alpha represents the maximum value that the MAX player can guarantee.
   - Beta represents the minimum value that the MIN player can guarantee.
   - At each MAX node, the algorithm updates alpha as the maximum value found so far, and prunes branches whose values are lower than alpha.
   - At each MIN node, the algorithm updates beta as the minimum value found so far, and prunes branches whose values are higher than beta.
4. The values that represent the usefulness of the nodes are determined, propagating them back through successive parent nodes as in the Minimax algorithm.
5. The root node chooses for MAX that move which leads to the maximum value.

!!! The alpha value associated with the MAX type nodes can never decrease, and the beta value related to the MIN type node can never increase.

Implementation details

The implementation adheres to the guidelines outlined in [1].

Algorithms

Application of Minimax algorithm:

```python
def mini_max(state: State) -> State:
    """
    Apply the minimax algorithm to find the best move for the current player.

    Args:
        state (State): The current state of the game.

    Returns:
        State: The best move for the current player.
    """
    # Check if we have reached a leaf of the tree
    if state.depth == 0 and state.game_configuration.final():
        # Calculate score of the leaf
        state.score = state.game_configuration.estimate_score(state.MAX_DEPTH)
        return state

    # Otherwise, calculate all possible moves from the current state
    state.possible_moves = state.state_possible_moves()

    # Apply the minimax algorithm on all possible moves (thus calculating their subtrees)
    moves_score = [mini_max(move) for move in state.possible_moves]

    if state.current_player == GameConfiguration.JMAX:
        # If it is the MAX player's turn, choose the son state with the maximum score
        state.chosen_state = max(moves_score, key=lambda x: x.score)
    else:
        # If it is the MIN player's turn, choose the son state with the minimum score
        state.chosen_state = min(moves_score, key=lambda x: x.score)

    # Update the score of the "father" = the score of the chosen "son"
    state.score = state.chosen_state.score
    return state
```

Application of Alpha-beta algorithm:

```python
def alpha_beta(alpha: int, beta: int, state: State):
    """
    Alpha-beta pruning algorithm.
    Args:
        alpha: The best score that the maximizer can guarantee at that level or above.
        beta: The best score that the minimizer can guarantee at that level or above.
        state: The current state of the game.
    """

    # We check if the current state is a leaf node (final game state) or if the maximum depth has
        been reached
    if state.depth == 0 or state.game_configuration.game_over():
        state.score = state.game_configuration.estimate_score()
        return state

    # Prune the tree branches that do not need to be explored
    if alpha >= beta:
        return state

    # Calculate all the possible moves from the current state
    state.possible_moves = state.state_possible_moves()

    if state.current_player == GameConfiguration.PMAX:
        current_score = float('-inf')

        # For each possible move for MIN, we calculate the score of the successor state
        for move in state.possible_moves:
            new_state = alpha_beta(alpha, beta, move)

            # Try to maximize the score of PMAX knowing that PMIN will play optimally
            if current_score < new_state.score:
                state.chosen_state = move
                current_score = new_state.score

            if alpha < new_state.score:
                alpha = new_state.score
                if alpha >= beta: # Prune the tree branches that do not need to be explored
                    break # beta cut-off

    else:
        current_score = float('inf')

        # For each possible move for MAX, we calculate the score of the successor state
        for move in state.possible_moves:
            new_state = alpha_beta(alpha, beta, move)

            # Try to minimize the score of PMAX knowing that it will play optimally
            if current_score > new_state.score:
                state.chosen_state = new_state
                current_score = new_state.score

            if beta > new_state.score:
                beta = new_state.score
                if alpha >= beta: # check the pruning condition
                    break # alpha cut-off
```

```
55
56      # Update the score of the parent state
57      if state.chosen_state != None:
58          state.score = state.chosen_state.score
59
60      return state
```

Heuristic function

The heuristic function is used to evaluate the quality of different game states and determine the best move to make. The function takes the player symbol as input and returns a numeric value representing the utility of the current state.

The function first determines the player and opponent objects based on their symbols. It then checks if the opponent has placed a bomb in the same row or column as the player. If so, the function calculates the distance between the player and the bomb and returns a negative value if the player is within range of the explosion. This suggests that the player should move away from the bomb.

If the opponent has not placed a bomb or if it is not in the same row or column as the player, the function calculates the distance between the player and the opponent and the distance between the player and the closest item that can be collected. It returns a weighted average of these distances, suggesting that the player should prioritise getting closer to the opponent and collecting items that provide protection.

Overall, the heuristic function seems to be more focused on the offensive aspects of the game but not entirely (getting closer to the opponent and placing bombs) than on defence, as it only penalises the player for being within range of the opponent's bomb and does not consider the player's own bombs as a defensive tool.

User interaction

The process consists of interacting with a command-line-based game engine that allows the user to choose a difficulty level and player. The user will be prompted to input their preferred difficulty level and opponent at the beginning of the programme. The game engine will then determine the depth of the search tree based on the user's selected difficulty level. A deeper search tree allows for a more precise prediction of the opponent's future moves, but it requires more computational power and time.

If the user inputs an incorrect response, the query will be repeated until a valid response is given. This ensures that the game engine is initialised with the correct difficulty level and opponent and that the subsequent gameplay is as intended.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.