CSCI-6511 TeamTikiTaki TeamID:1387
Artificial Intelligence
Team 4: C. Norton, M. Kotur, M. Therupalli

Project 3: Generalized Tic Tac Toe

Introduction

Given board size (*n*) and target (*m*), Generalized Tic Tac Toe is a game between two players played on an *n*n* board. The player who plays first uses the *'O'* symbol, and the other player uses the *'X'* symbol. Each player takes turns placing their symbol on the board. A player wins when they are able to place the *m* consecutive symbols in a contiguous sequence (row, column or diagonal). The game may end in a draw when no one wins.

The goal of this project is to develop an agent that effectively employs an adversarial search algorithm to play Generalized Tic Tac Toe. Our agent, written in Java, will compete with other agents with the same purpose. Using a predetermined API, games will be played interactively with other teams from the class and points will be rewarded to the winning teams. The API allows the following actions: Create a Team, Add a Team Member, Get a Team List, Remove a Team Member, Create a game, Make a move, Get the Move List, Get Board String, and Get Board Map.

Evaluation Function

The evaluation function used in our project calculates the utility of each game state during a Minimax search. A move's utility is evaluated e during the Minimax search algorithm in the evaluate() method of the Node class. The function assigns a score to a given game state based on the number of consecutive symbols (horizontal, vertical, and two diagonal directions). It then uses the scores to compute a total score for the board state. The total score is higher if the player has more pieces in a row in any direction.The evalScore array in the Game class holds the scores for the number of consecutive symbols needed to win the game, and it is used in the evaluate() method to calculate the score of each possible move. Furthermore, different weights are assigned based on the number of pieces in a row. The weights are set based on the number of pieces needed to win the game. The more pieces needed to win, the higher the weight assigned to a configuration of pieces in a row.

Moreover, our evaluation function calculates The function returns a score that indicates how good a state is for the player that is about to make a move.

Key Points About Algorithm

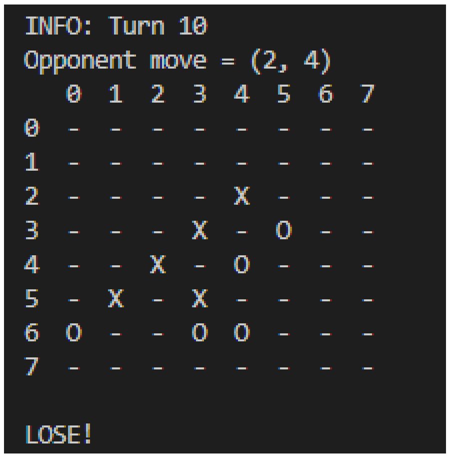
- Our project employs the minimax search algorithm discussed in lecture 5. It is a recursive algorithm that is used for game playing and decision making in situations where there are two players with opposing goals, such as in games like chess or tic-tac-toe. This allows us to not only maximize the utility of our moves, but minimize the utility of our opponent's moves.
- The algorithm works by assuming that one player will always try to maximize their score or outcome, while the other player will always try to minimize the score or outcome.
- The algorithm explores the game tree by recursively evaluating each possible move (at a depth of 3) and its consequences for both players, assuming that both players will play optimally.
- The algorithm returns the best possible move for the maximizer, assuming that the minimizer will play optimally to try to minimize the score.
- The minimax function checks whether the game has ended or the maximum depth of the search tree has been reached (Max depth is 3). If so, it returns the score of the current board position.
- If the game has not ended and the maximum depth has not been reached, the function recursively evaluates each possible move and returns the best score found.
- The minimax function alternates between maximizing and minimizing the score at each level of the tree, depending on whether the current player is the maximizer or not.
- The main find_best_move( ) function calls the minimax function for each possible move and returns the move that leads to the best score for the maximizer.

Tricks Used To Improve Algorithm Performance

- *Alpha-Beta Pruning*
    - Used to minimize the number of possible moves being evaluated by the minimax algorithm in the search tree.
    - Here the alpha-beta pruning algorithm is implemented by recursively exploring the game tree and maintaining two values: alpha and beta. Alpha represents the maximum lower bound of the possible scores for the maximizing player, while beta represents the minimum upper bound of the possible scores for the minimizing player. These scores are calculated using the evaluate() function described earlier.
    - During the search, if a node's score is found to be worse than the current alpha, then that node's branch can be discarded because the maximizing player will

never choose it. Therefore those node leaves are pruned. Similarly, if a node's score is found to be better than the current beta, then that node's branch can be discarded because the minimizing player will never choose it. By doing this, we can prune large parts of the search space and reduce the number of nodes that need to be evaluated.

- Other Tricks
  - For the early first moves, we settled that the AI would pick the center as this is the move that gives the most options later on. The second and third moves would be 1 unit from the center to incentivise the AI to complete the target as soon as possible. However this could be inefficient as it might create gaps when playing defensively or if the other AI is able to block your path early on. The other moves would be to search with multiple threads to create different node states and run the min and max values.
  - In order to counteract the lack of defensive traits in the AI, we installed a function checkEnd() and winorloseimmediatly variable to check that if the AI sees that the other opponent will win, it will block their winning move. However this can be counterted in the case that the opponent has more than one winning move. This is such a case:



```
INFO: Turn 10
Opponent move = (2, 4)
   0  1  2  3  4  5  6  7
0  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -
2  -  -  -  -  X  -  -  -
3  -  -  -  X  -  O  -  -
4  -  -  X  -  O  -  -  -
5  -  X  -  X  -  -  -  -
6  O  -  -  O  O  -  -  -
7  -  -  -  -  -  -  -  -

LOSE!
```

We have tried playing with the depth to increase speed but also intelligence to prevent this case.