Tiange Wang
3717659

CS165A MP2 Report

This AI for dots and boxes was implemented via a gradually increasing level of intelligence. For the minimax algorithm, the root act the beginning of the board where the MAX player(me) makes the first move. The root's children includes all possible moves that the MAX player can make on the next move using the helper function list_possible_moves(). For each child, a different move is selected and thus a different game world is generated. The game world is shown through the next_state() function. After the MAX player make its first move, the MIN player has to make the second move. More possible game worlds are generated using the same method for each of the game world generated from the first step. This process repeats with alternating MIN and MAX and the tree grows exponentially. For my implementation, I went with a tree of depth 2 as results have shown that a depth of 2 is sufficient enough to beat player 1. Additional parameters included in my minimax algorithm include a next_move parameter which is used when the base case is reached, the horizontal/vertical parameters indicating the current (or calculated) board state, and the max_player Boolean variable indicating who is the current maximizing player. For the base case of the method, it is only run when the depth is 0 (leaf) or there are no more possible moves left on the board. This code block simply returns next_move, the move that was used to arrive at this game world. For the recursive case, the code block executed depends on the current player. If the max_player variable is true, it means that the MAX player is executing a move. For the MAX player, the goal is to find a move that result in the highest evaluation score. This is done by iterating through all of the node's children in order to find out which path should the MAX player follow to in order to maximize the final score. However, in each iteration, in order to find out the evaluation of the current node, the node's children have to be evaluated first. The node's children for a MAX player would be a MIN player, and MIN player's goal would be to minimize the evaluation score. This alteration of MAX and MIN player goes all the way down to the base case. The implementation of the alteration is done by calling the minimax function again but with the opposing player's Boolean value in the method parameter and a depth of the original depth minus 1. After the base case is reached, the code would propagate back up to the original caller and return evaluation score of the move. This evaluation score is calculated for all of the

children of the caller node in order to find out which children would give the best score for the caller. Before iterating through all children, a variable max_eval is made for the MAX player to store the max move and min_eval is made for the MIN player to store the min move. They are initialized to the first available move at the beginning. For each iteration, each move will be checked against this variable. If the evaluation score of a child is larger (for max_eval)/smaller (for min_eval) than the move stored in the variable, then the variable would store this new move. At the end the max move would be returned for MAX player and the min move would be returned for the MIN player.

Throughout the minimax algorithm, the evaluation score is mentioned multiple times as an index of how "good" the move is. As mentioned at the beginning of the report, this evaluation strategy was implemented via a gradually increasing level of intelligence. At the beginning, the evaluation function was simply random. The random function would return a score of 1 as long as a move is valid. This strategy is obviously not good enough for the program as the opponent player always find opportunity to fill up a box and always end up scoring more points than my naïve AI. For the next level of intelligence, the evaluation score is higher for moves that can complete the fourth side of a box. If the move can complete 2 boxes at the same time, the score is doubled. This strategy performs better than the original random strategy, however it is still not enough to beat player 1. In order to improve the AI, player 1's moves are also analyzed. For this level, the previous level's strategy of completing boxes are kept. In addition to that, the evaluation function also explores one level deeper to discover the next move. If on player 2(me)'s turn it did not complete a box, then the next move would be made by the player 1 and if player 1 can complete a box due to my move, scores would be subtracted, indicating it is not a good move by player 2. If on player 2's turn it can complete a box, then the next move is still made by me and if I can complete another box due to my previous move, scores would be added, indicating it is a good move by player 2. For player 1, the strategy is the same, however its goal would be the minimize the evaluation function rather than maximizing it. To translate this strategy into code, my final evaluation function takes in 4 parameters. The first parameter move is the move that will be analyzed by the function. The second and third parameters horizontal and vertical are the board states to be analyzed. The

last parameter max_player is a boolean variable used to determine who is the current maximizing player. If the value is true, then the final score doesn't change. If the value score is false, then the final score would be multiplied by -1. This step is implemented at the beginning of the evaluation method by creating a variable named multiplier initialized to 1. If the value of max_player is false, then the value of multiplier is changed to -1. After creating this variable, by using the move, horizontal and vertical parameters the next state is determined using the next_state() function. Another variable named eval_result is also created and set to the value of increment_score(), which indicate possible point increase due to this move. Then, we look through all children of this move and try to determine what next move can result in maximum points after making this move, since either player would try to aim for this move afterwards. After finding out this max score, the final score is determined through the code block as follows:

```
if eval_result > 0:
        return (eval_result * 2 - max_score) * multiplier
else:
        return (-max_score) * multiplier
```

If eval_result is greater than 0, then it means that the first move the player makes result in completing the fourth side of a box. Thus, this score is multiplied by 2(first move values more than second move) and max_score(determined by analyzing all children of this move, as the next move might also grant the current player with more boxes) is subtracted from it since the next move is always made by the opponent, and finally multiplied by the multiplier(negative for MIN, positive for MAX). If eval_result is less than 0, then it means that the first move the player makes cannot result in completing the fourth side of a box, which means the next move would be made by the opposing player. Thus, the max_score would be first multiplied by -1 as an indication of benefiting the opponent, then multiplied by the multiplier again. By combining the use of the minimax algorithm and this evaluation strategy, player 2 is able to beat player 1 with a score line of 29-7.

To reduce time taken by the minimax algorithm, the alpha-beta pruning strategy is used. The alpha-beta pruning function is implemented on top of the implementation of the minimax algorithm, as the strategy of searching is the same. In addition to the function parameters

passed in the minimax algorithm, 2 additional function parameters alpha and beta are added. They are also passed into the function when examining moves and are updated each time after a move has been evaluated. If at any point the beta value is less than or equal to the alpha value, then it means this path would never be picked and thus the function and simply return. Based on the alpha and beta values, some moves are simply ignored to save computation time. The depth of algorithm is also changed to 1 to observe different result based on different inputs. The result is rather interesting as reducing the depth with alpha-beta pruning actually increased player 2's performance from 29-7 in the previous round to 32-4, an even bigger score difference. / Example of alpha & beta value: alpha starts at -infinity and beta starts at infinity. The algorithm goes all the way to the bottom without changing the alpha/beta value. Once it reaches the bottom, the evaluation function is executed, and the alpha is updated to 2. Now it recursive goes back up a step and goes into another branch and acquire the beta value of 2. Since now alpha is equal to beta, no more children of their parent node need to be checked and the algorithm simply returns.

The time taken to make a move are printed onto the console.