Intelligent Systems Programming - Project 1
Spring 2018

Thor Olesen (tvao@itu.dk)
Daniel Hansen (daro@itu.dk)
Dennis Nguyen (dttn@itu.dk)

March 14, 2018

Contents

1 Search Algorithm

1.1 Minimax Algorithm

The following section describes our implementation of the adversarial min max search algorithm. The algorithm is implemented recursively as shown below:

Source Code 1: Mini Max Algorithm

```
private Position minimax(GameState state) {
        int depth = 0;
        float alpha = Float.MIN_VALUE;
        float beta = Float.MAX_VALUE;
        playerToMaximise = s.getPlayerInTurn();
        return maximum(state,alpha,beta,depth).finalPosition;
}
```

Initially, the algorithm will assign a value of positive or negative infinity to any position since the value of every position will be qual to some final winning or losing position. The first player that makes a move maximizes the minimum value of the position resulting from the opponent's possible following moves, hence the call to maximum:

Source Code 2: Maximum function

```
private Value maximum(GameState state, float alpha, float beta, int depth) {
        float bestScore = Float.MIN_VALUE;
        Value bestMove = null;

        if(terminalTest(state,depth)) return eval(state);
        if(state.legalMoves().isEmpty()) return minimum(state, alpha, beta, depth+1);

        for(Position action : state.legalMoves()) {
                float childScore =
                        minimum(performMove(state, action), alpha, beta, depth+1).Value;

                if(childScore > bestScore) {
                        bestScore = childScore;
                        bestMove = new Value(bestScore,action);
                }
                if (bestScore >= beta) return bestMove; //beta cutoff
                alpha = max(alpha, bestScore);
        }
        return bestMove;
}
```

As can be seen, the maximum function explores the nodes (i.e. positions) of the game tree recursively and uses a predefined depth denoting the amount of steps that are looked ahead.

In general, the number of nodes explored in the game tree increases exponentially due to its branching factor (i.e. number of children of each node) and the number of plies (i.e. depth of the tree), leading to O(b$\hat{d}$) nodes explored in a game tree of branching factor b and search depth of d plies. Thus, a fixed depth is used to avoid looking ahead for a solution too far ahead in the game tree.

1.2 Alpha Beta Pruning

The naive minimax algorithm has been improved by using alpha-beta pruning. Namely, the number of nodes in the search space are reduced by not evaluating any moves that prove to be worse than a previously examined move. In practice, branches of the search tree are eliminated by maintaining two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially, alpha is set to negative infinity and beta to positive infinity in code example 1 to indicate that both players start with their worst possible score. Whenever the maximum score that the minimizing player (beta) is assured of becomes less than the minimum score that the maximum player (alpha) is assured of (i.e. beta ¡= alpha), the maximum player disregards the descendants of that particular node, as it will never be reached:

Source Code 3: Minimum function

```
private Value minimum(GameState state, float alpha, float beta, int depth) {
        float bestScore = Float.MAX_VALUE;
        Value bestMove = null;

        if(terminalTest(state, depth)) return eval(state);
        if(state.legalMoves().isEmpty()) return maximum(state, alpha, beta, depth + 1);

        for(Position action :  state.legalMoves()) {
                float childScore =
                        maximum(performMove(state, action), alpha, beta, depth + 1).Value;

                if(childScore < bestScore) {
                        bestScore = childScore;
                        bestMove = new Value(bestScore, action);
                }

                if(bestScore <= alpha) return bestMove; // alpha cutoff
                        beta = min(beta, bestScore);
        }
        return bestMove;
}
```

As can be seen, the modified maximum and minimum methods ensure that the search algorithm excludes exploring nodes (i.e. moves) in the game tree that will not be taken by an optimally playing opponent. In other words, the alpha beta search algorithm uses pruning to avoid searching this part of the tree, which is captured by adding the tests **"if(bestScore ¿= beta) return bestMove;"** and **"if(bestScore ¡= alpha) return bestMove;"**. In this way, the maximizing player (alpha) disregards score values when bestScore (i.e. alpha max) is greater than beta min, vice versa. As a result, the modified minimax algorithm prunes away branches that do not influence the final decision move.

2 Evaluation and Cut Off Function

The evaluation function is used to compute a value for a given position that indicates how good it would be for a player to reach that position. Initially, a naive utility method was used calculating the value based on the amount of tokens on the board:

Source Code 4: Utility function

```
private Value utility(GameState state) {
        int[] tokens = state.countTokens();
        if(playerToMaximise == 1) return new Value(tokens[0],null); // Max
        else return new Value(tokens[1],null); // Min
}
```

The quality of this score has been improved dramatically by using another heuristic function, which evaluates the score based on the desirability of a given board position:

Source Code 5: Eval function

```
private Value eval(GameState state) {
        float score = 0.0f;
        int[][] board = state.getBoard();
        int width = board.length - 1;
        int height = board[0].length - 1;
        //check corners
        score +=checkCorner(board, 0, 0);
        score +=checkCorner(board, width, 0);
        score +=checkCorner(board, 0, height);
        score +=checkCorner(board, width, height);
        //check tokens on board of max (0) and min(1)
        int[] tokens = state.countTokens();
        score += playerToMaximise == 1 ? 0.1f * tokens[0] : 0.1f * tokens[1];
        return new Value(score,null);
}
```

By explanation, the new heuristic function weights the corners of the board very high, as disc tokens played in the corners cannot be flipped.

In terms of the cut off function, the terminal test below is used to terminate the search if the game is finished or a certain depth has been reached:

Source Code 6: Terminal-test (cut off function)

```
private boolean terminalTest(GameState state, int depth) {
        return state.isFinished() || depth >= MAX_DEPTH;
}
```

2.1 Future Improvements

Besides cutting off nodes using alpha beta pruning, the search algorithm will cut off the search based on a fixed depth d. In the future, we might improve is using iterative deepening to allow searches to successively deeper plies if there is time. Also, a transportation table or an online database might be used to lookup whole board game states in advance instead of always calculating the best decision move with the search algorithm. Finally, the heuristics used in the evaluation (i.e. utility) function may be improved.

Other heuristic measures that might be considered in the future may be based on the following statistics:

- the number of disc tokens that cannot be flipped for the rest of the game
- who is expected to make the last move of the game (zero weight in the opening, but increases towards endgame)
- Number of moves a player is able to make